ADOBE® DIRECTOR® 11
Scripting Dictionary

Adobe® Director® 11 software Getting Started Guide for Windows®

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Contents

Chapter 9: Constants

Chapter 10: Events and Messages

Chapter 11: Keywords

Chapter 12: Methods

Chapter 14: Properties

Chapter 15: Physics Engine

Chapter 1: Introduction

This reference provides conceptual and how-to information about scripting in Adobe® Director® 11, and also provides reference descriptions and examples for the scripting application programming interfaces (APIs) that you use to write scripts.

The scripting APIs are the means by which you access the functionality of Director through script to add interactivity to a movie. By using these APIs, you can create interactive functionality that is identical to that provided by the prewritten behaviors that are shipped with Director, in addition to functionality that is more powerful and more varied than that provided by the prewritten behaviors.

The prewritten behaviors enable you to add basic interactive functionality to a movie, such as moving the playhead to a frame number or marker, or zooming in when a user clicks a sprite. They also enable non-interactive functionality such as sprite animation, media loading, and frame navigation. The scripting APIs enable you to extend and customize these types of functionality.

Intended audience

This reference is intended for you if you want to do any of the following:

- Extend the existing functionality of prewritten behaviors by using script.
- Add functionality to a movie by using script instead of prewritten behaviors.
- Add more powerful, varied, and custom functionality to a movie than that provided by prewritten behaviors.

This reference is intended to provide all the information, from basic to advanced, you need to add interactivity to your movies by using script. Therefore, you do not need to have any prior scripting experience to write effective scripts in Director.

Regardless of your experience level with Director, Lingo, or JavaScript™ syntax, take a few moments to browse "Director Scripting Essentials" on page 4 and "Writing Scripts in Director" on page 41 before you begin writing scripts. Like any product, Director has its own unique set of scripting conventions, types of data, and so on. You will need to be comfortable with these unique characteristics of Director before you can write effective scripts.

What's new with Director scripting

If you have written scripts in previous versions of Director, you should note some new and important changes about scripting in this latest release.

Adobe Director supports scripting in Unicode.

Limitations of Unicode support in Director

- Languages written right-to-left are not supported.
- File I/O Xtra functions such as readchar(), getLength(), and getPosition() work only with a one-byte character input. To read two- or three-byte Unicode characters, read the entire file into a string object using the readFile() method. Then, use the char...of method to read the Unicode character.

- Components of Director Physics do not support Unicode.
- Unicode names for HTTP paths are not supported.
- You cannot name a scripting Xtra as a Unicode string using the 'kMoaMmDictType_MessageTable' registry key. Also, you cannot expose Lingo functions named in Unicode using the scripting Xtras.
- Scripting Xtras expose Lingo functions. These function names exposed through the Xtras are not supported in Unicode.
- 3D model names are not supported in Unicode.

The Script window has an Explorer panel in addition to the Script Editor. By default, the Explorer panel appears to the left of the Script Editor. You can view the Explorer panel in the Dictionary view or the Script Browser view.

Dictionary view

The Dictionary view displays a list of built-in Lingo script/JavaScript functions arranged in a tree structure. The functions are classified based on their category, and alphabetically as an index.

Use the Dictionary view to do the following:

- Browse built-in functions for Lingo scripts and JavaScripts.
- Use the built-in functions to create scripts.

Script Browser view

The Script Browser view displays the scripts and associated handlers that have been used in the movie. You can create new scripts and handlers in this view.

Use the Script Browser view to do the following:

- Browse scripts and handlers in the current movie as a tree or a list.
- Sort scripts based on their script name, cast name, cast number, or script type.
- Locate a handler in the Script Editor.
- Create scripts under each script type.

What's new in this documentation

If you learned how to script in previous versions of Director, you should know about some changes in the scripting documentation for this release. The *Director Scripting Reference* takes the place of the *Lingo Dictionary* that was shipped with previous versions of Director. This reference is organized differently than the *Lingo Dictionary*.

In the *Lingo Dictionary*, information about the scripting model was organized by feature. For example, if you wanted to learn how to work with sprites in script, you looked up that information in one of the sections under the Sprites heading, such as Dragging Sprites, Sprite Dimensions, and so on. In addition, all of the scripting APIs were listed in a single alphabetical list, which meant that all functions, properties, events, and so on were mixed together alphabetically.

In the *Director Scripting Reference*, information about the scripting model is organized by object. This organization closely reflects the organization of the actual scripting objects that you use in your scripts. For example, if you want to find out how to work with sprites in script, you should look for the Sprite section in the chapter, Director Core Objects.

The scripting APIs are still listed in alphabetical order, but they are categorized by API type. For example, all methods are listed alphabetically under the Methods heading, all properties are listed alphabetically under the Properties heading, and so on.

Finding information about scripting in Director

With its new organization, the *Director Scripting Reference* contains the following topics:

Director Scripting Essentials Provides information about the basic scripting concepts and components you will use when scripting in Director.

Writing Scripts in Director Provides information about the Director scripting environment in addition to advanced scripting concepts and techniques.

Debugging Scripts in Director Provides information about how to find problems in your scripts when they are not performing as expected.

Director Core Objects Provides a list of the objects and APIs that you use to access the core functionality and features in Director, such as the Director player engine, movie windows, sprites, sounds, and so on.

Media Types Provides a list of the media types and APIs that you use to access the functionality of the various media types in Director, such as RealMedia, DVD, Animated GIF, and so on, that are added to movies as cast members.

Scripting Objects Provides a list of the scripting objects, also known as Xtra extensions, and APIs that you use to extend core Director functionality. Xtra extensions provide capabilities such as importing filters and connecting to the Internet.

3D Objects Provides a list of the objects you use to add 3D functionality to a movie.

Constants Provides a list of the constants that are available in Director.

Events and Messages Provides a list of the events that are available in Director.

Keywords Provides a list of the keywords that are available in Director.

Methods Provides a list of the methods that are available in Director.

Operators Provides a list of the operators that are available in Director.

Properties Provides a list of the properties that are available in Director.

Chapter 2: Director Scripting Essentials

If you are new to scripting in Director®, you may want to take some time to learn the basic scripting concepts that are essential to understanding how to script in Director before you begin. Some of these essentials include definitions of important terms, syntax rules, available data types, and information about the basic elements of scripting in Director—for example, variables, arrays, operators, and so on.

Types of scripts

A Director movie can contain four types of scripts: behaviors, movie scripts, parent scripts, and scripts attached to cast members. Behaviors, movie scripts, and parent scripts all appear as independent cast members in the Cast window. A script attached to a cast member is associated with that cast member in the Cast window and does not appear independently.

- **Behaviors** are scripts that are attached to sprites or frames in the Score, and are referred to as sprite behaviors or frame behaviors. The Cast window thumbnail for each behavior contains a behavior icon in the lower-right corner.

  When used in the *Director Scripting Reference*, the term *behavior* refers to any script that you attach to a sprite or a frame. This differs from the behaviors that come in the Director Library Palette. For more information on these behaviors, which are built into Director, see the Using Director topics in the Director Help Panel.

  All behaviors that have been added to a cast library appear in the Behavior inspector's Behavior pop-up menu. Other types of scripts do not appear there.

  You can attach the same behavior to more than one location in the Score. When you edit a behavior, the edited version is applied everywhere the behavior is attached in the Score.

- **Movie scripts** contain handlers that are available globally, or on a movie level. Event handlers in a movie script can be called from other scripts in the movie as the movie plays.

  A movie script icon appears in the lower-right corner of the movie script's Cast window thumbnail.

  Movie scripts are available to the entire movie, regardless of which frame the movie is in or which sprites the user is interacting with. When a movie plays in a window or as a linked movie, a movie script is available only to its own movie.

- **Parent scripts** are special scripts that contain Lingo that is used to create child objects. You can use parent scripts to generate script objects that behave and respond similarly yet can still operate independently of each other. A parent script icon appears in the lower-right corner of the Cast window thumbnail.

  For information on using parent scripts and child objects, see "Object-oriented programming with Lingo syntax" on page 47.

  JavaScript syntax does not use parent scripts or child objects; it uses regular JavaScript syntax-style object-oriented programming techniques. For information on object-oriented programming in JavaScript syntax, see "Object-oriented programming with JavaScript syntax" on page 56

- **Flash® member components** placed on the stage (Flash sprites) that are invisible can be accessed only by using the Member object. Using a Sprite object for a Flash sprite with an invisible property will result in an error.

- **Scripts** attached to cast members are attached directly to a cast member, independent of the Score. Whenever the cast member is assigned to a sprite, the cast member's script is available.

Unlike behaviors, movie scripts, and parent scripts, cast member scripts do not appear in the Cast window. However, if Show Cast Member Script Icons is selected in the Cast Window Preferences dialog box, cast members that have a script attached display a small script icon in the lower-left corner of their thumbnails in the Cast window.

Scripting terminology

Both Lingo and JavaScript syntax use some terms that are specific to each language, in addition to some terms that are shared between each language.

Important scripting terms are listed here in alphabetical order. These terms are used throughout the *Director Scripting Reference*, so it will help to understand these terms before moving forward.

- **Constants** are elements whose values do not change. For example, in Lingo, constants such as `TAB`, `EMPTY`, and `RETURN` always have the same values and cannot be modified. In JavaScript syntax, constants such as `Math.PI` and `Number.MAX_VALUE` always have the same values and cannot be modified. You can also create your own custom constants in JavaScript syntax by using the keyword `const`.

  For more information on constants, see "Constants" on page 13.

- **Events** are actions that occur while a movie is playing. Events occur when a movie stops, a sprite starts, the playhead enters a frame, the user types at the keyboard, and so on. All events in Director are predefined, and always have the same meaning.

  For more information on events, see "Events" on page 26.

- **Expressions** are any part of a statement that produces a value. For example, `2 + 2` is an expression.
- **Functions** refer to either top-level functions or specific types of JavaScript syntax code.

  A *top-level function* instructs a movie to do something while the movie is playing or returns a value, but it is not called from any specific object. For example, you would call the top-level `list()` function by using the syntax `list()`. Like a function, a *method* also instructs a movie to do something while the movie is playing or returns a value, but it is always called from an object.

  A function is used in JavaScript syntax to represent an event handler, a custom object, a custom method, and so on. The use of JavaScript functions in these cases is described in the applicable topics later in this reference.

- **Handlers**, or event handlers, are sets of statements within a script that run in response to a specific event and subsequent message. When an event occurs, Director generates and sends a corresponding message to scripts, and a corresponding handler runs in response to the message. The names of handlers are always the same as the events and messages they respond to.

  Note: *Although in JavaScript syntax an event is actually handled by a function, the term* handler *is used generically throughout this reference to refer to both Lingo handlers and JavaScript syntax functions that handle events.*

  For more information on handlers, see "Handlers" on page 28.

- **Keywords** are reserved words that have a special meaning. For example, in Lingo, the keyword `end` indicates the end of a handler. In JavaScript syntax, the keyword `var` indicates that the term following it is a variable.

- **Lists** (Lingo) or **Arrays** (JavaScript syntax) are ordered sets of values used to track and update an array of data, such as a series of names or the values assigned to a set of variables. A simple example is a list of numbers such as `[1, 4, 2]`.

  For more information on using lists in both Lingo and JavaScript syntax, see "Linear lists and property lists" on page 31.

  For more information on using JavaScript syntax arrays, see "JavaScript syntax arrays" on page 37.

- **Messages** are notices that Director sends to scripts when specific events occur in a movie. For example, when the playhead enters a specific frame, the `enterFrame` event occurs and Director sends an `enterFrame` message. If a script contains an `enterFrame` handler, the statements within that handler will run, because the handler received the `enterFrame` message. If no scripts contain a handler for a message, the message is ignored in script.

  For more information on messages, see "Messages" on page 27

- **Methods** are terms that either instruct a movie to do something while the movie is playing or return a value, and are called from an object. For example, you would call the `insertFrame()` method from the Movie object, using the syntax `_movie.insertFrame()`. Although similar in functionality to top-level functions, methods are always called from an object, and top-level functions are not.

- **Operators** are terms that calculate a new value from one or more values. For example, the addition (`+`) operator adds two or more values together to produce a new value.

  For more information on operators, see "Operators" on page 18.

- **Parameters** are placeholders that let you pass values to scripts. Parameters only apply to methods and event handlers, and not to properties. They are required by some methods and are optional for others.

  For example, the Movie object's `go()` method sends the playhead to a specific frame, and optionally specifies the name of the movie that frame is in. To perform this task, the `go()` method requires at least one parameter, and allows for a second parameter. The first required parameter specifies what frame to send the playhead to, and the second optional parameter specifies what movie the frame is in. Because the first parameter is required, a script error will result if it is not present when the `go()` method is called. Because the second parameter is optional, the method will perform its task even if the parameter is not present.

- **Properties** are attributes that define an object. For example, a sprite in a movie has specific attributes, such as how wide it is, how tall it is, its background color, and so on. To access the values of these three specific attributes, you would use the Sprite object's `width`, `height`, and `backColor` properties.

  For more information on assigning properties to variables, see "Storing and updating values in variables" on page 15.

- **Statements** are valid instructions that Director can execute. All scripts are made up of sets of statements. The following Lingo is a single complete statement.:

  ```
  _movie.go("Author")
  ```

  For more information on writing script statements, see "Scripting in dot syntax format" on page 42

- **Variables** are elements used to store and update values. Variables must start with a letter, an underscore (_), or the dollar sign (`$`). Subsequent characters in a variable name can also be digits (0-9). To assign values to variables or change the values of many properties, you use the equals (`=`) operator.

  For more information on using variables, see "Variables" on page 14.

Scripting syntax

The following are general syntax rules that apply to Lingo and JavaScript syntax:

- Comment markers vary between Lingo and JavaScript syntax.

  All Lingo comments are preceded by double hyphens (`--`). Each line of a comment that covers multiple lines must be preceded by double hyphens.

  ```
  -- This is a single-line Lingo comment

  -- This is a
  -- multiple-line Lingo comment
  ```

  JavaScript syntax comments on a single line are preceded by a double-slash (`//`). Comments that cover multiple lines are preceded with a `/*` and followed by a `*/`.

  ```
  // This is a single-line JavaScript syntax comment

  /* This is a
  multiple-line JavaScript syntax comment */
  ```

  You can place a comment on its own line or after any statement. Any text following the comment markers on the same line are ignored.

  Comments can consist of anything you want, such as notes about a particular script or handler or notes about a statement whose purpose might not be obvious. Comments make it easier for you or someone else to understand a procedure after you've been away from it for a while.

  Adding large numbers of comments does not increase the size of your movie file when it is saved as a compressed DCR or DXR file. Comments are removed from the file during the compression process.

  Comment markers can also be used to ignore sections of code you want to deactivate for testing or debugging purposes. By adding comment markers rather than removing the code, you can temporarily turn a section into comments. Select the code you want to turn on or off, and then use the Comment or Uncomment buttons in the Script window to quickly add or remove comment markers.

- Parentheses are required after all method and function names. For example, when calling the Sound object's `beep()` method, you must include the parentheses after the word beep. Otherwise, a script error will occur.

  ```
  // JavaScript syntax
  _sound.beep(); // this statement will work properly
  _sound.beep; // this statement will result in a script error
  ```

  When you call a method, function, or handler from within another method, function, or handler, you must include parentheses in the calling statement. In the following example, the `modifySprite()` method contains a call to a `spriteClicked` handler. The call to the `spriteClicked` handler must include parentheses; otherwise, a script error occurs:

  ```
  // JavaScript syntax
  function modifySprite() {
      spriteClicked(); // this call to the handler will work properly
      spriteClicked; // this call to the handler results in a script error
  }
  function spriteClicked() {
      // handler code here
  }
  ```

You can also use parentheses to override the order of precedence in math operations, or to make your statements easier to read. For example, the first math expression below yields a result of 13, while the second expression yields a result of 5:

```
5 * 3 - 2 -- yields 13
5 * (3 - 2) -- yields 5
```

- Event handler syntax varies between Lingo and JavaScript syntax. In Lingo, handlers use the syntax `on handlerName`. In JavaScript syntax, handlers are implemented as functions and use the syntax `function handlerName()`. For example, the following statements make up a handler that plays a beep when the mouse button is clicked:

```
-- Lingo syntax
on mouseDown
    _sound.beep()
end
// JavaScript syntax
function mouseDown() {
    _sound.beep();
}
```

- Event handler parameter syntax can vary between Lingo and JavaScript syntax. Both Lingo and JavaScript syntax support enclosing parameters passed to a handler within parentheses. If more than one parameter is passed, each parameter is separated by a comma. In Lingo, you can also pass parameters that are not enclosed by parentheses. For example, the following `addThem` handler receives the two parameters `a` and `b`:

```
-- Lingo syntax
on addThem a, b -- without parentheses
    c = a + b
end

on addThem(a, b) -- with parentheses
    c = a + b
end

// JavaScript syntax
function addThem(a, b) {
    c = a + b;
}
```

- The `const` keyword can be used in JavaScript syntax to specify a constant whose value does not change. Lingo has its own predefined set of constants (`TAB`, `EMPTY`, and so on); therefore, the keyword `const` does not apply to Lingo.

  For example, the following statement specifies a constant named `intAuthors` and sets its value to 12. This value will always be 12, and cannot be changed through script:

```
// JavaScript syntax
const intAuthors = 12;
```

- The `var` keyword in JavaScript syntax can be placed in front of a term to specify that the term is a variable. The following statement creates a variable named `startValue`:

```
// JavaScript syntax
var startValue = 0;
```

  Note: *Although using `var` in JavaScript syntax is optional, it is recommended that you always declare local JavaScript syntax variables, or those inside a function, using `var`. For more information on using variables, see* *"Variables" on page 14*.

- The line continuation symbol (\) in Lingo indicates that a long line of sample code has been broken into two or more lines. Lines of Lingo that are broken in this way are not separate lines of code. For example, the following code would still run:

```
-- Lingo syntax
tTexture = member("3D").model("box") \
    .shader.texture
```

  JavaScript syntax does not include a line continuation symbol. To break multiple lines of JavaScript syntax code, add a carriage return at the end of a line, and then continue the code on the following line.

- Semicolons can be used to specify the end of a statement of JavaScript syntax code. Semicolons do not apply to Lingo.

  Using a semicolon is optional. If used, it is placed at the end of a complete statement. For example, both of the following statements create a variable named `startValue`:

```
// JavaScript syntax
var startValue = 0
var startValue = 0;
```

  A semicolon does not necessarily specify the end of a line of JavaScript syntax code, and multiple statements can be placed on one line. However, placing separate statements on separate lines is recommended in order to improve readability. For example, the following three statements occupy only one line of code and function properly, but it is difficult to read the code:

```
// JavaScript syntax
_movie.go("Author"); var startValue = 0; _sound.beep();
```

- Character spaces within expressions and statements are ignored in both Lingo and JavaScript syntax. In strings of characters surrounded by quotation marks, spaces are treated as characters. If you want spaces in a string, you must insert them explicitly. For example, the first statement below ignores the spaces between the list items, and the second statement includes the spaces.

```
-- Lingo syntax
myList1 = ["1",      "2",      "3"] -- yields ["1", "2", "3"]
myList2 = ["  1  ", "  2  ", "  3  "] -- yields ["  1  ", "  2  ", "  3  "]
```

- Case-sensitivity can vary between Lingo and JavaScript syntax.

  Lingo is not case-sensitive in any circumstance—you can use uppercase and lowercase letters however you want. For example, the following four statements are equivalent:

```
-- Lingo syntax
member("Cat").hilite = true
member("cat").hiLite = True
MEMBER("CAT").HILITE = TRUE
Member("Cat").Hilite = true
```

  Although Lingo is not case-sensitive, it's a good habit to choose a case convention and use it consistently throughout your scripts. This makes it easier to identify names of handlers, variables, cast members, and so on.

  JavaScript syntax is case-sensitive when referring to objects, the top-level properties or methods that refer to objects, or when referring to user-defined variables. For example, the top-level `sprite()` method returns a reference to a specific Sprite object, and is implemented in Director with all lowercase letters. The first statement below refers to the name of the first sprite in a movie, while the second and third statements result in a script error.

```
// JavaScript syntax
sprite(1).name // This statement functions normally
Sprite(1).name // This statement results in a script error
```

```
SPRITE(1).name // This statement results in a script error
```

Literal strings are always case-sensitive in both Lingo and JavaScript syntax.

For more information on using strings, see "Strings" on page 12.

Data types

A *data type* is a set of data with values that have similar, predefined characteristics. Every variable and property value in Director is of a specific data type, and values returned by methods are of a specific data type.

For example, consider the following two statements. In the first statement, variable `intX` is assigned a whole number value of 14, which is an integer. So, the data type of variable `intX` is Integer. In the second statement, variable `stringX` is assigned a sequence of character values, which is a string. So, the data type of variable `stringX` is String.

```
-- Lingo syntax
intX = 14
stringX = "News Headlines"

// JavaScript syntax
var intX = 14;
var stringX = "News Headlines";
```

The values that are returned by methods or functions are also of an inherent data type. For example, the Player object's `windowPresent()` method returns a value that specifies whether a window is present. The returned value is `TRUE` (1) or `FALSE` (0).

Some data types are shared between Lingo and JavaScript syntax, and some data types are specific to one language or another. The set of data types that Director supports is fixed and cannot be modified, meaning that new data types cannot be added and existing data types cannot be removed. Director supports the following data types.

Data type	Description
# (symbol)	A self-contained unit that can be used to represent a condition or flag. For example, `#list` or `#word`.
Array	(JavaScript syntax only) Although not literally a data type, an Array object can be used to work with linear lists of values. The functionality of an Array object is similar to that of the List data type in Lingo.
Boolean	A value that is `TRUE` (1) or `FALSE` (0). In Lingo, all `TRUE` or `FALSE` values are simple integer constants, 1 for `TRUE`, 0 for `FALSE`. In JavaScript syntax, all `true` or `false` values are by default the actual Boolean `true` or `false` values, but are converted to simple integer constants automatically when required in Director. In Lingo, `TRUE` and `FALSE` can be either lowercase or uppercase. In JavaScript syntax, `true` and `false` must always be lowercase.
Color	Represents an object's color.
Constant	A piece of data whose value does not change.
Date	Although not literally a data type, in JavaScript syntax a Date object can be used to work with dates. In Lingo, use the date() method to create a Date object and work with dates.
Float	(Lingo only) A floating-point number. For example, 2.345 or 45.43.
Function	(JavaScript syntax only) Although not literally a data type, a Function object can be used to specify a string of code to run.
Integer	(Lingo only) A whole number. For example, 5 or 298.

Data type	Description
List	A linear or property list made up of values or property names and values, respectively.
Math	(JavaScript syntax only) Although not literally a data type, a Math object can be used to perform mathematical functions.
null	(JavaScript syntax only) Denotes a variable whose value behaves as 0 in numeric contexts and as `false` in Boolean contexts.
Number	(JavaScript syntax only) Although not literally a data type, a Number object can be used to represent numerical constants, such as a maximum value, not-a-number (NaN), and infinity.
Object	Although not literally a data type, an Object object can be used to create a custom named container that contains data and methods that act on that data.
Point	A point in the coordinate space that has both a horizontal and vertical coordinate.
Rect	A rectangle in the coordinate space.
RegExp	(JavaScript only) A regular expression pattern that is used to match character combinations in strings.
String	A contiguous sequence of keyboard symbols or character values. For example, "Director" or "$5.00".
undefined	(JavaScript syntax only) Denotes a variable that does not have a value.
Vector	A point in 3D space.
VOID	(Lingo only) Denotes an empty value.

Note: *Many of the data types and objects that are specific to JavaScript syntax contain their own set of methods and properties that can be used to further manipulate those types. While the* Director Scripting Reference *may refer to some of these methods and properties, it does not include complete reference information about them. For more detailed information on these data types and objects, and their methods and properties, see one of the many third-party resources on the subject.*

The built-in properties in Director, such as the Cast object's `name` property, can only be assigned values that are the same data type as that of the property's inherent data type. For example, the Cast object's `name` property's inherent data type is a string, so the value must be a string such as `News Headlines`. If you try to assign a value of a different data type to this property, such as the integer 20, a script error occurs.

If you create your own custom properties, their values can be of any data type, regardless of the data type of the initial value.

Both Lingo and JavaScript syntax are dynamically typed. This means that you do not have to specify the data type of a variable when you declare it, and data types are automatically converted as needed while a script runs.

For example, the following JavaScript syntax initially sets the variable `myMovie` to an integer, and later in the script it is set to a string. When the script runs, the date type of `myMovie` is converted automatically:

```
-- Lingo syntax
myMovie = 15 -- myMovie is initially set to an integer
...
myMovie = "Animations" -- myMovie is later set to a string

// JavaScript syntax
var myMovie = 15; // myMovie is initially set to an integer
...
myMovie = "Animations"; // myMovie is later set to a string
```

Literal values

A *literal value* is any part of a statement or expression that is to be used exactly as it is, rather than as a variable or a script element. Literal values that you encounter in script are character strings, integers, decimal numbers, cast member names and numbers, frame and movie names and numbers, symbols, and constants.

Each type of literal value has its own rules.

Strings

Strings are words or groups of characters that script treats as regular words instead of as variables. Strings must be enclosed in double quotation marks. For example, you might use strings to give messages to the user of your movie or to name cast members. In the following statement, `Hello` and `Greeting` are both strings. `Hello` is the literal text being put into the text cast member; `Greeting` is the name of the cast member:

```
-- Lingo syntax
member("Greeting").text = "Hello"
```

Similarly, if you test a string, double quotation marks must surround each string, as in the following example:

```
-- Lingo syntax
if "Hello Mr. Jones" contains "Hello" then soundHandler
```

Both Lingo and JavaScript syntax treat spaces at the beginning or end of a string as a literal part of the string. The following expression includes a space after the word *to*:

```
// JavaScript syntax
trace("My thoughts amount to ");
```

Although Lingo does not distinguish between uppercase and lowercase when referring to cast members, variables, and so on, literal strings are case-sensitive. For example, the following two statements place different text into the specified cast member, because `Hello` and `HELLO` are literal strings:

```
-- Lingo syntax
member("Greeting").text = "Hello"
member("Greeting").text = "HELLO"
```

In Lingo, the `string()` function can convert a numerical value into a string. In JavaScript syntax, the `toString()` method can convert a numerical value into a string.

Note: *Attempting to use the `toString()` method in JavaScript syntax on a `null` or `undefined` value results in a script error. This is in contrast with Lingo, in which the `string()` function works on all values, including those that are `VOID`.*

Numbers

In Lingo, there are two types of numbers: integers and decimals.

An *integer* is a whole number, without any fractions or decimal places, in the range of -2,147,483,648 and +2,147,483,647. Enter integers without using commas. Use a minus (-) sign for negative numbers.

A *decimal number*, also called a *floating-point number*, or *float*, is any number that includes a decimal point. In Lingo, the `floatPrecision` property controls the number of decimal places used to display these numbers. Director always uses the complete number, up to 15 significant digits, in calculations; Director rounds any number with more than 15 significant digits in calculations.

JavaScript syntax does not distinguish between integers and floating-point numbers, and merely uses numbers. For example, the following statements illustrate that the number 1 is an integer in Lingo and a number in JavaScript syntax, and that the decimal number 1.05 is a float in Lingo and a number in JavaScript syntax:

```
-- Lingo syntax
put(ilk(1)) -- #integer
put(ilk(1.05)) -- #float

// JavaScript syntax
trace(typeof(1)) // number
trace(typeof(1.05)) // number
```

In Lingo, you can convert a decimal to an integer by using the `integer()` function. You can also convert an integer to a decimal by performing a mathematical operation on the integer, for example, by multiplying an integer by a decimal. In JavaScript syntax, you can convert a string or a decimal number to a whole number by using the `parseInt()` function. As opposed to Lingo's `integer()` function, `parseInt()` rounds down. For example, the following statement rounds off the decimal number 3.9 and converts it to the integer 4 (Lingo) and the number 3 (JavaScript syntax):

```
-- Lingo syntax
theNumber = integer(3.9) -- results in a value of 4

// JavaScript syntax
var theNumber = parseInt(3.9); // results in a value of 3
```

In Lingo, the `value()` function can convert a string into a numerical value.

You can also use exponential notation with decimal numbers: for example, `-1.1234e-100` or `123.4e+9`.

In Lingo, you can convert an integer or string to a decimal number by using the `float()` function. In JavaScript syntax, you can convert a string to a decimal number by using the `parseFloat()` function. For example, the following statement stores the value 3.0000 (Lingo) and 3 (JavaScript syntax) in the variable `theNumber`:

```
-- Lingo syntax
theNumber = float(3) -- results in a value of 3.0000

// JavaScript syntax
var theNumber = parseFloat(3) // results in a value of 3
```

Constants

A *constant* is a named value whose content never changes.

In Lingo, the predefined terms `TRUE`, `FALSE`, `VOID`, and `EMPTY` are constants because their values are always the same. The predefined terms `BACKSPACE`, `ENTER`, `QUOTE`, `RETURN`, `SPACE`, and `TAB` are constants that refer to keyboard characters. For example, to test whether the last key pressed was the Space bar, use the following statement:

```
-- Lingo syntax
if _key.keyPressed() = SPACE then beep()
```

In JavaScript syntax, you can access predefined constants using some of the data types that are unique to JavaScript syntax. For example, the Number object contains constants such as `Number.MAX_VALUE` and `Number.NaN`, the Math object contains constants such as `Math.PI` and `Math.E`, and so on.

Note: *This reference does not provide in-depth information about the predefined constants in JavaScript syntax. For more information on these constants, see one of the many third-party resources on the subject.*

In JavaScript syntax, you can also define your own custom constants by using the `const` keyword. For example, the following statement creates a constant named `items`, and assigns it a value of 20. This value cannot be changed after it has been created:

```
// JavaScript syntax
const items = 20;
```

For more information on constants, see "Constants" on page 142.

Symbols

A *symbol* is a string or other value in Lingo that begins with the pound (#) sign.

Symbols are user-defined constants. Comparisons using symbols can usually be performed very quickly, providing more efficient code.

Note: *In Lingo, the uppercase for the symbol is mapped to the lowercase. If you try to convert a string (for example, Sonia) into a symbol by using the symbol function, the output will be sonia.*

For example, the first statement below runs more quickly than the second statement:

```
-- Lingo syntax
userLevel = #novice
userLevel = "novice"
```

Symbols cannot contain spaces or punctuation.

In both Lingo and JavaScript syntax, convert a string to a symbol by using the `symbol()` method.

```
-- Lingo syntax
x = symbol("novice") -- results in #novice

// JavaScript syntax
var x = symbol("novice"); // results in #novice
```

Convert a symbol back to a string by using the `string()` function (Lingo) or the `toString()` method (JavaScript syntax).

```
-- Lingo syntax
x = string(#novice) -- results in "novice"

// JavaScript syntax
var x = symbol("novice").toString(); // results in "novice"
```

In JavaScript syntax, you cannot compare symbols of the same name to determine whether they refer to the same symbol. To compare symbols of the same name, you must first convert them to strings by using the `toString()` method, and then perform the comparison.

Variables

Director uses *variables* to store and update values. As the name implies, a variable contains a value that can be changed or updated as a movie plays. By changing the value of a variable as the movie plays, Director can do things such as store a URL, track the number of times a user takes part in an online chat session, record whether a network operation is complete, and many more options.

It is a good idea always to assign a variable a known value the first time you declare the variable. This is known as *initializing* a variable. Initializing variables makes it easier to track and compare the variable's value as the movie plays.

Variables can be global or local. A *local variable* exists only as long as the handler in which it is defined is running. A *global variable* can exist and retain its value as long as Director is running, including when a movie branches to another movie. A variable can be global within an individual handler, a specific script, or an entire movie; the scope depends on how the global variable is initialized.

If you want a variable to be available throughout a movie, it is good practice to declare it in an `on prepareMovie` (Lingo) or a `function prepareMovie()` (JavaScript syntax) handler. This ensures that the variable is available from the very start of the movie.

For more information on using both global and local variables, see "Using global variables" on page 16 and "Using local variables" on page 18.

Storing and updating values in variables

Variables can hold data for any of the data types found in Director, such as integers, strings, `TRUE` or `FALSE` values, symbols, lists, or the result of a calculation. Use the equals (`=`) operator to store the values of properties and variables.

As mentioned in the Data types section of this reference, variables in both Lingo and JavaScript syntax are dynamically typed, which means that they can contain different types of data at different times. (The ability to change a variable's type distinguishes Lingo from other languages such as Java™ and C++, in which a variable's type cannot be changed.)

For example, the statement `x = 1` creates the variable `x`, which is an integer variable because you assigned the variable an integer. If you subsequently use the statement `x = "one"`, the variable `x` becomes a string variable, because the variable now contains a string.

You can convert a string to a number by using the `value()` function (Lingo) or the `parseInt()` method (JavaScript syntax), or a number to a string by using the `string()` function (Lingo) or the `toString()` method (JavaScript syntax).

The values of some properties can be both set (the value is assigned) and returned (the value is retrieved), and some property values can only be returned. Properties whose values can be both set and returned are called *read/write*, and those that can only be returned are called *read-only*.

Often these are properties that describe some condition that exists outside the control of Director. For example, you cannot set the `numChannels` cast member property, which indicates the number of channels within a movie that contain Adobe® Shockwave® content. However, you can return the number of channels by referring to the `numChannels` property of a cast member.

Assign a value to a variable

❖ Use the equals (`=`) operator.

For example, the following statement assigns a URL to the variable `placesToGo`:

```
// JavaScript syntax
var placesToGo = "http://www.adobe.com";
```

Variables can also hold the results of mathematical operations. For example, the following statement adds the result of an addition operation to the variable `mySum`:

```
-- Lingo syntax
mySum = 5 + 5 -- this sets mySum equal to 10
```

As another example, the following statement returns the cast member assigned to sprite 2 by retrieving the value of the sprite's `member` property and places it into the variable `textMember`:

```
-- Lingo syntax
textMember = sprite(2).member
```

It is good practice to use variable names that indicate what the variable is used for. This makes your scripts easier to read. For example, the variable `mySum` indicates that the variable contains a sum of numbers.

Test the values of properties or variables

Use the `put()` or the `trace()` functions in the Message window or check the values in the Watcher window; (`put()` and `trace()` provide identical functionality and are available to both Lingo and JavaScript syntax).

For example, the following statement displays the value assigned to the variable `myNumber` in the Message window:

```
-- Lingo syntax
myNumber = 20 * 7
put(myNumber) -- displays 140 in the Message window

// JavaScript syntax
var myNumber = 20 * 7;
trace(myNumber) // displays 140 in the Message window
```

Using global variables

Global variables can be shared among handlers, scripts, or movies. A global variable exists and retains its value as long as Director is running or until you call the `clearGlobals()` method.

In Adobe Shockwave Player, global variables persist among movies displayed by the `goToNetMovie()` method, but not among those displayed by the `goToNetPage()` method.

Every handler that declares a variable as global can use the variable's current value. If the handler changes the variable's value, the new value is available to every other handler that treats the variable as global.

It is good practice to start the names of all global variables with a lowercase *g*. This helps identify which variables are global when you examine your code.

Director provides a way to display all current global variables and their current values and to clear the values of all global variables.

Display all current global variables and their current values

Use the Global object's `showGlobals()` method in the Message window.

For more information on the Message window, see "Debugging in the Message window" on page 80.

Clear all current global variables

Use the Global object's `clearGlobals()` method in the Message window to set the value of all global variables to `VOID` (Lingo) or `undefined` (JavaScript syntax).

To monitor the values of global variables during movie playback, use the Object inspector. For more information on the Object inspector, see "Debugging in the Object inspector" on page 83.

Global variables in Lingo

In Lingo, variables are considered local by default, and you do not need to precede the variable name with any keyword. To declare a global variable, you must precede the variable with the keyword `global`.

If you declare a global variable at the top of a script and before any handlers, the variable is available to all handlers in that specific script. If you declare a global variable within a handler, the variable is available only to that handler; however, if you declare a global variable with the same name within two separate handlers, an update to the variable's value in one handler will also be reflected in the variable in the other handler.

The following example illustrates working with two global variables: `gScript`, which is available to all handlers in the script, and `gHandler`, which is available within its defining handler and any other handlers that declare it on the first line of the handler:

```
-- Lingo syntax
global gScript -- gScript is available to all handlers

on mouseDown
    global gHandler
    gScript = 25
    gHandler = 30
end

on mouseUp
    global gHandler
    trace(gHandler) -- displays 30
end
```

In Lingo, when you use the term `global` to define global variables, the variables automatically have `VOID` as their initial value.

Global variables in JavaScript syntax

In JavaScript syntax, variables are considered global by default. The scope of a global variable can be determined by how and where it is declared.

- If you declare a variable within a JavaScript syntax function without preceding the variable name with the keyword `var`, the variable is available to all functions within its containing script.
- If you declare a variable outside a JavaScript syntax function, with or without the keyword `var`, the variable is available to all functions within its containing script.
- If you declare a variable inside or outside a JavaScript syntax function by using the syntax `_global.`*`varName`*, the variable is available to all scripts within a movie.

The following example uses the syntax `_global.gMovie` in one script to declare the variable `gMovie` as global. This variable is available to all scripts within the movie:

```
// JavaScript syntax
_global.gMovie = 1; // Declare gMovie in one script

// Create a function in a separate script that operates on gMovie
function mouseDown() {
    _global.gMovie++;
    return(_global.gMovie);
}
```

The following example declares the global variable `gScript` in one script. This variable is available only to functions within that script:

```
// JavaScript syntax
var gScript = 1; // Declare gScript in a script

// gScript is available only to functions in the script that defines it
function mouseDown() {
    gScript++;
    return(gScript);
}
```

In JavaScript syntax, when you define variables before any handlers, the variables automatically have `undefined` as their initial value.

Using local variables

A local variable exists only as long as the handler in which it is defined is running. However, after a local variable is created, you can use the variable in other expressions or change its value while a script is still within the handler that defined the variable.

Treating variables as local is a good idea when you want to use a variable only temporarily in one handler. This helps you avoid unintentionally changing the value in another handler that uses the same variable name.

Create a local variable

In Lingo, assign the variable a value using the equals (`=`) operator.

In JavaScript syntax, inside a function precede the variable name with the keyword `var`, and then assign the variable a value using the equals operator.

Note: *Because JavaScript syntax variables are global by default, if you attempt to declare a local variable inside a function without using the keyword* `var`*, your script could produce unexpected behavior. Therefore, although using* `var` *is optional, it is strongly recommended that you declare all local JavaScript syntax variables using* `var` *to avoid any unexpected behavior.*

Display all current local variables in the handler

In Lingo only, use the `showLocals()` function.

In Lingo, you can use this method in the Message window or in handlers to help with debugging. The result appears in the Message window. The `showLocals()` method does not apply to JavaScript syntax.

To monitor the values of local variables during movie playback, use the Object inspector. For more information on the Object inspector, see "Debugging in the Object inspector" on page 83.

Operators

Operators are elements that tell Lingo and JavaScript syntax scripts how to combine, compare, or modify the values of an expression. Many of the operators in Director are shared between Lingo and JavaScript syntax, and some are unique to each language.

Some types of operators include the following:

- **Arithmetic operators** (such as `+`, `-`, `/`, and `*`)
- **Comparison operators** (such as `<`, `>`, and `>=`), which compare two arguments
- **Logical operators** (`not`, `and`, `or`), which combine simple conditions into compound ones
- **String operators** (such as `&`, `&&`, and `+`), which join, or concatenate, strings of characters

Note: *There are many more types of operators in JavaScript syntax than there are in Lingo, and not all of them are covered in this reference. For more information on additional operators in JavaScript 1.5, see one of the many third-party resources on the subject.*

The items that operators act upon are called *operands*. In Lingo, there are only binary operators. In JavaScript syntax, there are both binary and unary operators. A *binary operator* requires two operands, one before the operator and one after the operator. A *unary operator* requires a single operand, either before or after the operator.

In the following example, the first statement illustrates a binary operator, where the variables `x` and `y` are operands and the plus (`+`) sign is the operator. The second statement illustrates a unary operator, where the variable `i` is the operand and `++` is the operator.

```
// JavaScript syntax
x + y; // binary operator
i++; // unary operator
```

For reference information on operators, see "Operators" on page 606.

Understanding operator precedence

When two or more operators are used in the same statement, some operators take precedence over others in a precise hierarchy to determine which operators to execute first. This is called the operators' *precedence order*. For example, multiplication is always performed before addition. However, items in parentheses take precedence over multiplication. In the following example, without parentheses the multiplication in this statement occurs first:

```
-- Lingo syntax
total = 2 + 4 * 3 -- results in a value of 14
```

When parentheses surround the addition operation, addition occurs first:

```
-- Lingo syntax
total = (2 + 4) * 3 -- results in a value of 18
```

Descriptions of the types of operators and their precedence order follow. Operators with higher precedence are performed first. For example, an operator whose precedence order is 5 is performed before an operator whose precedence order is 4. Operations that have the same order of precedence are performed left to right.

Arithmetic operators

Arithmetic operators add, subtract, multiply, divide, and perform other arithmetic operations. Parentheses and the minus sign are also arithmetic operators.

Operator	Effect	Precedence
( )	Groups operations to control precedence order.	5
-	When placed before a number, reverses the sign of a number.	5
*	Performs multiplication.	4
mod	(Lingo only) Performs modulo operation.	4
/	Performs division.	4
%	(JavaScript syntax only) Returns the integer remainder of dividing two operands.	4
++	(JavaScript syntax only) Adds one to its operand. If used as a prefix operator (`++x`), returns the value of its operand after adding one. If used as a postfix operator (`x++`), returns the value of its operand before adding one.	4
--	(JavaScript syntax only) Subtracts one from its operand. The return value is analogous to that of the increment operator.	4
+	When placed between two numbers, performs addition.	3
-	When placed between two numbers, performs subtraction.	3

Note: *In Lingo, when only integers are used in an operation, the result is an integer. Using integers and floating-point numbers in the same calculation results in a floating-point number. In JavaScript syntax, all calculations essentially result in floating-point numbers.*

When dividing one integer by another does not result in a whole number, Lingo rounds the result down to the nearest integer. For example, the result of `4/3` is `1`. In JavaScript syntax, the actual floating-point value, 1.333, is returned.

To force Lingo to calculate a value without rounding the result, use `float()` on one or more values in an expression. For example, the result of `4/float(3)` is 1.333.

Comparison operators

Comparison operators compare two values and determine whether the comparison is true or false.

Operator	Meaning	Precedence
==	(JavaScript syntax only) Two operands are equal. If the operands are not of the same data type, JavaScript syntax attempts to convert the operands to an appropriate data type for the comparison.	1
===	(JavaScript syntax only) Two operands are equal and of the same data type.	1
!=	(JavaScript syntax only) Two operands are not equal. If the operands are not of the same data type, JavaScript syntax attempts to convert the operands to an appropriate data type for the comparison.	1
!==	(JavaScript syntax only) Two operands are not equal and/or not of the same type.	1
<>	(Lingo only) Two operands are not equal.	1
<	The left operand is less than the right operand.	1
<=	The left operand is less than or equal to the right operand.	1
>	The left operand is greater than the right operand.	1
>=	The left operand is great than or equal to the right operand.	1
=	(Lingo only) Two operands are equal.	1

Assignment operators

An *assignment operator* assigns a value to its left operand based on the value of its right operand. With the exception of the basic assignment operator equal (=), all of the following shortcut assignment operators apply only to JavaScript syntax.

Operator	Meaning	Precedence
=	Equal	1
x += y	(JavaScript syntax only) x = x + y	1
x -= y	(JavaScript syntax only) x = x - y	1
x *= y	(JavaScript syntax only) x = x * y	1
x /= y	(JavaScript syntax only) x = x / y	1
x %= y	(JavaScript syntax only) x = x % y	1

Logical operators

Logical operators test whether two logical expressions are true or false.

Use care when using logical operators and string operators in Lingo and JavaScript syntax. For example, in JavaScript syntax, `&&` is a logical operator that determines whether two expressions are true, but in Lingo, `&&` is a string operator that concatenates two strings and inserts a space between the two expressions.

Operator	Effect	Precedence
and	(Lingo only) Determines whether both expressions are true	4
&&	(JavaScript syntax only) Determines whether both expressions are true	4
or	(Lingo only) Determines whether either or both expressions are true	4
||	(JavaScript syntax only) Determines whether either or both expressions are true	4
not	(Lingo only) Negates an expression	5
!	(JavaScript syntax only) Negates an expression	5

The `not` (Lingo) or ! (JavaScript syntax) operator is useful for toggling a `TRUE` or `FALSE` value to its opposite. For example, the following statement turns on the sound if it's currently off and turns off the sound if it's currently on:

```
-- Lingo syntax
_sound.soundEnabled = not (_sound.soundEnabled)

// JavaScript syntax
_sound.soundEnabled = !(_sound.soundEnabled);
```

String operators

String operators combine and define strings.

Use care when using logical operators and string operators in Lingo and JavaScript syntax. For example, in JavaScript syntax, `&&` is a logical operator that determines whether two expressions are true, but in Lingo, `&&` is a string operator that concatenates two strings and inserts a space between the two expressions.

Operator	Effect	Precedence
&	(Lingo only) Concatenates two strings	2
+	(JavaScript syntax only) Concatenates two string values and returns a third string that is the union of the two operands	2
+=	(JavaScript syntax only) Concatenates one string variable and one string value, and assigns the returned value to the string variable	2
&&	(Lingo only) Concatenates two strings and inserts a space between the two	2
"	Marks the beginning or end of a string	1

Conditional constructs

By default, Director always executes script statements starting with the first statement and continuing in order until it reaches the final statement or a statement that instructs a script to go somewhere else.

The order in which statements are executed affects the order in which you should place statements. For example, if you write a statement that requires some calculated value, you need to put the statement that calculates the value first.

The first statement in the following example adds two numbers, and the second statement assigns a string representation of the sum to a field cast member named `Answer`, which appears on the Stage. The second statement could not be placed before the first statement because the variable `x` has not yet been defined.

```
-- Lingo syntax
x = 2 + 2
member("Answer").text = string(x)

// JavaScript syntax
var x = 2 + 2;
member("Answer").text = x.toString();
```

Both Lingo and JavaScript syntax provide conventions for altering the default execution order or script statements, and for performing actions depending on specific conditions. For example, you may want to do the following in your scripts:

- Execute a set of statements if a logical condition is true, or execute alternate statements if the logical condition is false.
- Evaluate an expression and attempt to match the expression's value to a specific condition.
- Execute a set of statements repeatedly until a specific condition is met.

Testing for logical conditions

To execute a statement or set of statements if a specified condition is true or false, you use the `if...then...else` (Lingo) or `if...else` (JavaScript syntax) structures. For example, you can create an `if...then...else` or `if...then` structure that tests whether text has finished downloading from the Internet and, if it has, then attempts to format the text. These structures use the following pattern to test for logical conditions:

- In both Lingo and JavaScript syntax, statements that check whether a condition is true or false begin with the term `if`.
- In Lingo, if the condition exists, the statements following the term `then` are executed. In JavaScript syntax, curly brackets (`{ }`) take the place of the Lingo term `then`, and must surround each individual `if`, `else`, or `else if` statement.
- In both Lingo and JavaScript syntax, if the condition does not exist, scripts skip to the next statement in the handler using the term `else` or `else if`.
- In Lingo, the term `end if` specifies the end of the `if` test. In JavaScript syntax, the `if` test ends automatically, so there is no term that explicitly ends the test.

To optimize your script's performance, test for the most likely conditions first.

The following statements test several conditions. The term `else if` specifies alternative tests to perform if previous conditions are false:

```
-- Lingo syntax
if _mouse.mouseMember = member(1) then
    _movie.go("Cairo")
else if _mouse.mouseMember = member(2) then
    _movie.go("Nairobi")
else
    _player.alert("You're lost.")
end if
```

```
// JavaScript syntax
if (_mouse.mouseMember = member(1)) {
    _movie.go("Cairo");
}
else if (_mouse.mouseMember = member(2)) {
    _movie.go("Nairobi");
}
else {
    _player.alert("You're lost.");
}
```

When writing `if...then` structures in Lingo, you can place the statement or statements following `then` in the same line as `then`, or you can place them on their own line by inserting a carriage return after `then`. If you insert a carriage return, you must also include an `end if` statement at the end of the `if...then` structure.

When writing `if` structures in JavaScript syntax, you can place the statement or statements following `if` in the same line as `if`, or you can place them on their own line by inserting a carriage return after `if`.

For example, the following statements are equivalent:

```
-- Lingo syntax
if _mouse.mouseMember = member(1) then _movie.go("Cairo")

if _mouse.mouseMember = member(1) then
    _movie.go("Cairo")
end if

// JavaScript syntax
if (_mouse.mouseMember = member(1)) { _movie.go("Cairo"); }

if (_mouse.mouseMember = member(1)) {
    _movie.go("Cairo");
}
```

For reference information on using the `if...then...else` and `if...else` structures, see "if" on page 195.

Evaluating and matching expressions

The `case` (Lingo) or `switch...case` (JavaScript syntax) structures are shorthand alternatives to using `if...then...else` or `if...then` structures when setting up multiple branching structures. The `case` and `switch...case` structures are often more efficient and easier to read than many `if...then...else` or `if...then` structures.

In Lingo, the condition to test for follows the term `case` in the first line of the `case` structure. The comparison goes through each line in order until Lingo encounters an expression that matches the test condition. When a matching expression is found, Director executes the Lingo that follows the matching expression.

In JavaScript syntax, the condition to test for follows the term `switch` in the first line of the structure. Each comparison in the test follows the term `case` for each line that contains a test. Each `case` comparison can be ended by using the optional term `break`. Including the term `break` breaks the program out of the `switch` structure and executes any statements following the structure. If `break` is omitted, the following `case` comparison is executed.

A `case` or `switch...case` structure can use comparisons as the test condition.

For example, the following `case` and `switch...case` structures test which key the user pressed most recently and respond accordingly:

- If the user pressed A, the movie goes to the frame labeled Apple.

- If the user pressed B or C, the movie performs the specified transition and then goes to the frame labeled Oranges.
- If the user pressed any other letter key, the computer beeps.

```
-- Lingo syntax
case (_key.key) of
    "a" : _movie.go("Apple")
    "b", "c":
         _movie.puppetTransition(99)
         _movie.go("Oranges")
    otherwise: _sound.beep()
end case

// JavaScript syntax
switch (_key.key) {
    case "a" :
        _movie.go("Apple");
        break;
    case "b":
    case "c":
         _movie.puppetTransition(99);
        _movie.go("Oranges");
        break;
        default: _sound.beep()
}
```

***Note:** In JavaScript syntax, only one comparison can be made per `case` statement.*

For reference information on using `case` structures, see "case" on page 190.

Repeating actions

In both Lingo and JavaScript syntax, you can repeat an action a specified number of times or while a specific condition exists.

In Lingo, to repeat an action a specified number of times, you use a `repeat with` structure. Specify the number of times to repeat as a range following `repeat with`.

In JavaScript syntax, to repeat an action a specified number of times, you use the `for` structure. The `for` structure takes three parameters: the first parameter typically initializes a counter variable, the second parameter specifies a condition to evaluate each time through the loop, and the third parameter is typically used to update or increment the counter variable.

The `repeat with` and `for` structures are useful for performing the same operation on a series of objects. For example, the following loop makes Background Transparent the ink for sprites 2 through 10:

```
-- Lingo syntax
repeat with n = 2 to 10
    sprite(n).ink = 36
end repeat

// JavaScript syntax
for (var n=2; n<=10; n++) {
    sprite(n).ink = 36;
}
```

The following example performs a similar action, but with decreasing numbers:

```
-- Lingo syntax
repeat with n = 10 down to 2
```

```
    sprite(n).ink = 36
end repeat

// JavaScript syntax
for (var n=10; n>=2; n--) {
    sprite(n).ink = 36;
}
```

In Lingo, to repeat a set of instructions as long as a specific condition exists, use the `repeat while` structure.

In JavaScript syntax, to repeat a set of instructions as long as a specific condition exists, use the `while` structure.

For example, the following statements instruct a movie to beep continuously whenever the mouse button is being pressed:

```
-- Lingo syntax
repeat while _mouse.mouseDown
    _sound.beep()
end repeat

// JavaScript syntax
while (_mouse.mouseDown) {
    _sound.beep();
}
```

Both Lingo and JavaScript syntax scripts continue to loop through the statements inside the loop until the condition is no longer true, or until one of the statements sends the script outside the loop. In the previous example, the script exits the loop when the mouse button is released because the `mouseDown` condition is no longer true.

In Lingo, to exit a loop, use the `exit repeat` statement.

In JavaScript syntax, to exit a loop you can use the term `break`. A loop also automatically exits when a condition is no longer true.

For example, the following statements make a movie beep while the mouse button is pressed, unless the mouse pointer is over sprite 1. If the pointer is over sprite 1, the script exits the loop and stops beeping. The `rollover()` method indicates whether the pointer is over the specified sprite.

```
-- Lingo syntax
repeat while _mouse.stillDown
    _sound.beep()
    if _movie.rollOver(1) then exit repeat
end repeat

// JavaScript syntax
while (_mouse.stillDown) {
    _sound.beep();
    if (_movie.rollOver(1)) {
        break;
    }
}
```

For reference information on the `repeat while` and `while` structures, see "repeat while" on page 207.

Events, messages, and handlers

A key component to creating powerful, useful scripts is an understanding of the concepts and functionality of events, messages, and handlers. Understanding the order in which events and messages are sent and received will help you determine exactly when specific scripts or parts of scripts should run. It will also help you debug scripts when specific actions are not occurring when you expect them to occur.

When a movie plays, the following occur:

- Events occur in response to either a system or user-defined action.
- Messages that correspond to these events are sent to the scripts in a movie.
- Handlers within scripts contain the instructions that run when a specific message is received.

The name of an event corresponds to the name of the message it generates, and the handler that handles the event corresponds to both the event and the message. For example, when the event named `mouseDown` occurs, Director generates and sends to scripts a message named `mouseDown`, which would in turn be handled by a handler named `mouseDown`.

Events

There are two categories of events that occur when a movie plays:

- **System events** occur without a user interacting with the movie and are predefined and named in Director—for example, when the playhead enters a frame, when a sprite is clicked, and so on.
- **User-defined events** occur in response to actions that you define. For example, you could create an event that occurs when the background color of a sprite changes from red to blue, when a sound has played five times, and so on.

Many system events, such as `prepareFrame`, `beginSprite`, and so on, occur automatically and in a predefined order while a movie is playing. Other system events, particularly mouse events such as `mouseDown`, `mouseUp`, and so on, do not necessarily occur automatically while a movie is playing, but rather when a user triggers them.

For example, when a movie first starts, the `prepareMovie` event always occurs first, the `prepareFrame` event always occurs second, and so on. However, the `mouseDown` and `mouseUp` events may never occur in a movie unless a user triggers them by clicking the movie.

The following lists illustrate the system events that always occur during a movie, and the order in which they occur.

When the movie first starts, events occur in the following order:

1 `prepareMovie`

2 `prepareFrame` Immediately after the `prepareFrame` event, Director plays sounds, draws sprites, and performs any transitions or palette effects. This event occurs before the `enterFrame` event. A `prepareFrame` handler is a good location for script that you want to run before the frame draws.

3 `beginSprite` This event occurs when the playhead enters a sprite span.

4 `startMovie` This event occurs in the first frame that plays.

When the movie encounters a frame, events occur in the following order:

1 `beginSprite` This event occurs only if new sprites begin in the frame.

2 `stepFrame`

3 `prepareFrame`

4 `enterFrame` After `enterFrame` and before `exitFrame`, Director handles any time delays required by the tempo setting, idle events, and keyboard and mouse events.

5 `exitFrame`

6 `endSprite` This event occurs only if the playhead exits a sprite in the frame.

When a movie stops, events occur in the following order:

1 `endSprite` This event occurs only if sprites currently exist in the movie.

2 `stopMovie`

For reference information on the predefined system events in Director, see "Events and Messages" on page 149.

Messages

To run the appropriate set of script statements at the right time, Director must determine what is occurring in the movie and which statements to run in response to certain events.

Director sends *messages* to indicate when specific events occur in a movie, such as when sprites are clicked, keyboard keys are pressed, a movie starts, the playhead enters or exits a frame, or a script returns a certain result.

The general order in which messages are sent to objects is as follows:

1 Messages are sent first to behaviors attached to a sprite involved in the event. If a sprite has more than one behavior attached to it, behaviors respond to the message in the order in which they are attached to the sprite.

2 Messages are sent next to a script attached to the cast member assigned to the sprite.

3 Messages are then sent to behaviors attached to the current frame.

4 Messages are sent last to movie scripts.

Although you can define your own message names, most common events that occur in a movie have built-in message names.

For reference information on the built-in messages in Director, see "Events and Messages" on page 149.

Defining custom messages

In addition to using built-in message names, you can define your own messages and corresponding handler names. A custom message can call another script, another handler, or the statement's own handler. When the called handler stops executing, the handler that called it resumes.

Custom message and handler names must meet the following criteria:

- They must start with a letter.
- They must include alphanumeric characters only (no special characters or punctuation).
- They must consist of one word or of several words connected by an underscore—no spaces are allowed.
- They must be different from the name of any predefined Lingo or JavaScript syntax element.

Using predefined Lingo or JavaScript keywords for message and handler names can create confusion. Although it is possible to explicitly replace or extend the functionality of a Lingo or JavaScript element by using it as a message or handler name, this should be done only in certain advanced situations.

When you have multiple handlers with similar functions, it is useful to give them names that have similar beginnings so they appear together in an alphabetical listing, such as the listing that can be displayed by the Edit > Find > Handler command.

Handlers

A *handler* is a set of statements within a script that runs in response to a specific event and subsequent message. Although Director contains built-in events and messages, you must create your own custom handlers for each event/message pair that you want to handle.

Deciding where to place handlers

You can place handlers in any type of script, and a script can contain multiple handlers. It's a good idea to group related handlers in a single place, though, for easier maintenance.

The following are some useful guidelines for many common situations:

- To associate a handler with a specific sprite, or to have a handler run in response to an action on a specific sprite, put the handler in a behavior attached to the sprite.
- To set up a handler that should be available any time the movie is in a specific frame, put the handler in a behavior attached to the frame.

  For example, to have a handler respond to a mouse click while the playhead is in a frame, regardless of where the click occurs, place a `mouseDown` or `mouseUp` handler in the frame behavior rather than in a sprite behavior.
- To set up a handler that runs in response to messages about events anywhere in the movie, put the handler in a movie script.
- To set up a handler that runs in response to an event that affects a cast member, regardless of which sprites use the cast member, put the handler in a cast member script.

Determining when handlers receive a message

After sending a message to scripts, Director checks for handlers in a definite order.

1 Director first checks whether a handler exists in the object from which the message was sent. If a handler is found, the message is intercepted, and the script in the handler runs.

2 If no handler is found, Director then checks cast members, in ascending order, for any associated movie scripts that might contain a handler for the message. If a handler is found, the message is intercepted, and the script in the handler runs.

3 If no handler is found, Director then checks whether a frame script contains a handler for the message. If a handler is found, the message is intercepted, and the script in the handler runs.

4 If no handler is found, Director then checks sprites, in ascending order, for any scripts associated with the sprites that might contain a handler for the message. If a handler is found, the message is intercepted, and the script in the handler runs.

After a handler intercepts a message, the message does not automatically pass on to the remaining locations. However, in Lingo you can use the `pass()` method to override this default rule and pass the message to other objects.

If no matching handler is found after the message passes to all possible locations, Director ignores the message.

The exact order of objects to which Director sends a message depends on the message. For information on the sequence of objects to which Director sends specific messages, see the entry for each message in "Events and Messages" on page 149.

Using parameters to pass values to a handler

By using parameters for values, you can give the handler exactly the values that it needs to use at a specific time, regardless of where or when you call the handler in the movie. Parameters can be optional or required, depending on the situation.

Create parameters for a handler

In Lingo, put the parameters after the handler name.

In JavaScript syntax, put the parameters within parentheses, and then put them after the handler name.

Use commas to separate multiple parameters.

When you call a handler, you must provide specific values for the parameters that the handler uses. You can use any type of value, such as a number, a variable that has a value assigned, or a string of characters. Values in the calling statement must be in the order that they follow in the handler's parameters, and they must be surrounded by parentheses.

In the following example, the variable assignment `mySum` calls the method `addThem`, which is passed the two values 2 and 4. The `addThem` handler replaces the parameter placeholders `a` and `b` with the two values passed to it, stores the result in the local variable `c`, and then uses the keyword `return` to send the result back to the original method, which is then assigned to `mySum`.

Because 2 is first in the list of parameters, it is substituted for `a` in the handler. Likewise, because 4 is second in the list of parameters, it is substituted for `b` in the handler.

```
-- Lingo syntax
mySum = addThem(2, 4) -- calling statement

on addThem a, b -- handler
    c = a + b
    return c -- returns the result to the calling statement
end

// JavaScript syntax
var mySum = addThem(2, 4); // calling statement

function addThem(a, b) { // handler
    c = a + b;
    return c; // returns the result to the calling statement
}
```

In Lingo, when you call a custom method from an object, a reference to the Script object in memory is always passed as an implied first parameter to the handler for that method. This means that you must account for the Script object reference in your handler.

For example, consider that you wrote a custom sprite method named `jump()` that takes a single integer as a parameter, and you placed the method in a behavior. When you call `jump()` from a Sprite object reference, the handler must also include a parameter that represents the Script object reference, and not just the single integer. In this case, the implied parameter is represented by the keyword `me`, but any term will work.

```
-- Lingo syntax
myHeight = sprite(2).jump(5)

on jump(me,a)
    return a + 15 -- this handler works correctly, and returns 20
end

on jump(a)
    return a + 15 -- this handler does not work correctly, and returns 0
end
```

You can also use expressions as values. For example, the following statement substitutes `3+6` for `a` and `8>2` (or `1`, representing `TRUE`) for `b`, and would return `10`:

```
-- Lingo syntax
```

```
mySum = addThem(3+6, 8>2)
```

In Lingo, each handler begins with the word `on` followed by the message that the handler should respond to. The last line of the handler is the word `end`. You can repeat the handler's name after `end`, but this is optional.

In JavaScript syntax, each handler begins with the word `function` followed by the message that the handler should respond to. The statements that make up the handler are surrounded by opening and closing brackets, as are all JavaScript syntax functions.

Returning results from handlers

Often, you want a handler to report some condition or the result of some action.

Return results from a handler

Use the keyword `return` to have a handler report a condition or the result of an action. For example, the following `findColor` handler returns the current color of sprite 1:

```
-- Lingo syntax
on findColor
    return sprite(1).foreColor
end

// JavaScript syntax
function findColor() {
    return(sprite(1).foreColor);
}
```

You can also use the keyword `return` by itself to exit from the current handler and return no value. For example, the following `jump` handler returns nothing if the `aVal` parameter equals 5; otherwise, it returns a value:

```
-- Lingo syntax
on jump(aVal)
    if aVal = 5 then return

    aVal = aVal + 10
    return aVal
end
// JavaScript syntax
function jump(aVal) {
    if(aVal == 5) {
        return;
    }
    else {
        aVal = aVal + 10;
        return(aVal);
    }
}
```

When you define a handler that returns a result, you must use parentheses after the handler when you call it from another handler. For example, the statement `put(findColor())` calls the `on findColor` handler and then displays the result in the Message window.

Linear lists and property lists

In your scripts, you may want to track and update lists of data, such as a series of names or the values assigned to a set of variables. Both Lingo and JavaScript syntax have access to linear lists and property lists. In a linear list, each element in the list is a single value. In a property list, each element in the list contains two values; the first value is a property name, and the second value is the value associated with that property.

Because both Lingo and JavaScript syntax have access to linear and property lists, it is recommended that you use linear lists and property lists if values in your code are shared between Lingo and JavaScript syntax scripts.

If values in your code are used only in JavaScript syntax scripts, it is recommended that you use JavaScript Array objects to work with lists of data. For more information on using arrays, see "JavaScript syntax arrays" on page 37.

Creating linear lists

You create a linear list in one of the following ways:

- In Lingo, use either the top-level `list()` function or the list operator (`[ ]`), using commas to separate items in the list.
- In JavaScript syntax, use the top-level `list()` function, using commas to separate items in the list.

The index into a linear list always starts with 1.

When you use the top-level `list()` function, you specify the list's elements as parameters of the function. This function is useful when you use a keyboard that does not provide square brackets.

All of the following statements create a linear list of three names and assign it to a variable:

```
-- Lingo syntax
workerList = ["Bruno", "Heather", "Carlos"] -- using the Lingo list operator
workerList = list("Bruno", "Heather", "Carlos") -- using list()

// JavaScript syntax
var workerList = list("Bruno", "Heather", "Carlos"); // using list()
```

You can also create empty linear lists. The following statements create empty linear lists:

```
-- Lingo syntax
workerList = [] -- using the Lingo list operator
workerList = list() -- using list() with no parameters

// JavaScript syntax
var workerList = list(); // using list() with no parameters
```

Creating property lists

You create a property list in one of the following ways:

- In Lingo, use either the top-level `propList()` function or the list operator (`[:]`). When using the list operator to create a property list, you can use either a colon to designate name/value elements and commas to separate elements in the list, or commas to both designate name/value elements and to separate elements in the list.
- In JavaScript syntax, use the top-level `propList()` function and insert commas to both designate name/value elements and to separate elements in the list.

When you use the top-level `propList()` function, you specify the property list's elements as parameters of the function. This function is useful when you use a keyboard that does not provide square brackets.

Properties can appear more than once in a given property list.

All of the following statements create a property list with four property names—`left`, `top`, `right`, and `bottom`—and their corresponding values:

```
-- Lingo syntax
sprite1Loc = [#left:100, #top:150, #right:300, #bottom:350]
sprite1Loc = ["left",400, "top",550, "right",500, "bottom",750]
sprite1Loc = propList("left",400, "top",550, "right",500, "bottom",750)

// JavaScript syntax
var sprite1Loc = propList("left",400, "top",550, "right",500, "bottom",750);
```

You can also create empty property lists. The following statements create empty property lists:

```
-- Lingo syntax
sprite1Loc = [:] -- using the Lingo property list operator
sprite1Loc = propList() -- using propList() with no parameters

// JavaScript syntax
var sprite1Loc = propList(); // using propList() with no parameters
```

Setting and retrieving items in lists

You can set and retrieve individual items in a list. The syntax differs for linear and property lists.

Set a value in a linear list

Do one of the following:

- Use the equals (=) operator.
- Use the `setAt()` method.

The following statements illustrate defining the linear list `workerList` that contains one value, `Heather`, and then adds `Carlos` as the second value in the list:

```
-- Lingo syntax
workerList = ["Heather"] -- define a linear list
workerList[2] = "Carlos" -- set the second value using the equal operator
workerList.setAt(2, "Carlos") -- set the second value using setAt()

// JavaScript syntax
var workerList = list("Heather"); // define a linear list
workerList[2] = "Carlos"; // set the second value using the equal operator
workerList.setAt(2, "Carlos"); // set the second value using setAt()
```

Retrieve a value in a linear list

1 Use the list variable followed by the number that indicates the value's position in the list. Place square brackets around the number.

2 Use the `getAt()` method.

The following statements create the linear list `workerList`, and then assign the second value in the list to the variable `name2`:

```
-- Lingo syntax
workerList = ["Bruno", "Heather", "Carlos"] -- define a linear list
name2 = workerList[2] -- use bracketed access to retrieve "Heather"
name2 = workerList.getAt(2) -- use getAt() to retrieve "Heather"

// JavaScript syntax
```

```
var workerList = list("Bruno", "Heather", "Carlos");
var name2 = workerList[2] // use bracketed access to retrieve "Heather"
var name2 = workerList.getAt(2) // use getAt() to retrieve "Heather"
```

Set a value in a property list

Do one of the following:

- Use the equals (=) operator.
- In Lingo only, use the `setaProp()` method.
- Use dot syntax.

The following Lingo statement uses the equals operator to make `sushi` the new value associated with the property `Bruno`:

```
-- Lingo syntax
foodList = [:] -- define an empty property list
foodList[#Bruno] = "sushi" -- associate sushi with Bruno
```

The following Lingo statement uses `setaprop()` to make `sushi` the new value associated with the property `Bruno`:

```
-- Lingo syntax
foodList = [:] -- define an empty property list
foodList.setaProp(#Bruno, "sushi") -- use setaProp()

// JavaScript syntax
foodList = propList() -- define an empty property list
foodList.setaProp("Bruno", "sushi") -- use setaProp()
```

The following statements use dot syntax to set the value associated with `Bruno` from `sushi` to `teriyaki`:

```
-- Lingo syntax
foodList = [#Bruno:"sushi"] -- define a property list
trace(foodList) -- displays [#Bruno: "sushi"]
foodList.Bruno = "teriyaki" -- use dot syntax to set the value of Bruno
trace(foodList) -- displays [#Bruno: "teriyaki"]

// JavaScript syntax
var foodList = propList("Bruno", "sushi"); // define a property list
trace(foodList); // displays ["Bruno": "sushi"]
foodList.Bruno = "teriyaki" // use dot syntax to set the value of Bruno
trace(foodList) -- displays [#Bruno: "teriyaki"]
```

Retrieve a value in a property list

Do one of the following:

- Use the list variable followed by the name of the property associated with the value. Place square brackets around the property.
- Use the `getaProp()` or `getPropAt()` methods.
- Use dot syntax.

The following statements use bracketed access to retrieve the values associated with the properties `breakfast` and `lunch`:

```
-- Lingo syntax
-- define a property list
foodList = [#breakfast:"Waffles", #lunch:"Tofu Burger"]
trace(foodList[#breakfast]) -- displays "Waffles"
trace(foodList[#lunch]) -- displays "Tofu Burger"
```

```
// JavaScript syntax
// define a property list
var foodList = propList("breakfast", "Waffles", "lunch", "Tofu Burger");
trace(foodList["breakfast"]); // displays Waffles
trace(foodList["lunch"]); // displays Tofu Burger
```

The following statements use `getaProp()` to retrieve the value associated with the property `breakfast`, and `getPropAt()` to retrieve the property at the second index position in the list:

```
-- Lingo syntax
-- define a property list
foodList = [#breakfast:"Waffles", #lunch:"Tofu Burger"]
trace(foodList.getaProp(#breakfast)) -- displays "Waffles"
trace(foodList.getPropAt(2)) -- displays #lunch

// JavaScript syntax
// define a property list
var foodList = propList("breakfast", "Waffles", "lunch", "Tofu Burger");
trace(foodList.getaProp("breakfast")) // displays Waffles
trace(foodList.getPropAt(2)) // displays lunch
```

The following statements use dot syntax to access the values associated with properties in a property list:

```
-- Lingo syntax
-- define a property list
foodList = [#breakfast:"Waffles", #lunch:"Tofu Burger"]
trace(foodList.breakfast) -- displays "Waffles"

// JavaScript syntax
// define a property list
var foodList = propList("breakfast", "Waffles", "lunch", "Tofu Burger");
trace(foodList.lunch); // displays Tofu Burger
```

Checking items in lists

You can determine the characteristics of a list and the number of items the list contains by using the following methods:

- To display the contents of a list, use the `put()` or `trace()` functions, passing the variable that contains the list as a parameter.
- To determine the number of items in a list, use the `count()` method (Lingo only) or the `count` property.
- To determine a list's type, use the `ilk()` method.
- To determine the maximum value in a list, use the `max()` method.
- To determine the minimum value in a list, use the `min()` function.
- To determine the position of a specific property, use the `findPos`, `findPosNear`, or `getOne` command.

The following statements use `count()` and `count` to display the number of items in a list:

```
-- Lingo syntax
workerList = ["Bruno", "Heather", "Carlos"] -- define a linear list
trace(workerList.count()) -- displays 3
trace(workerList.count) -- displays 3

// JavaScript syntax
var workerList = list("Bruno", "Heather", "Carlos"); // define a linear list
trace(workerList.count); // displays 3
```

The following statements use `ilk()` to determine a list's type:

```
-- Lingo syntax
x = ["1", "2", "3"]
trace(x.ilk()) // returns #list

// JavaScript syntax
var x = list("1", "2", "3");
trace(x.ilk()) // returns #list
```

The following statements use `max()` and `min()` to determine the maximum and minimum values in a list:

```
-- Lingo syntax
workerList = ["Bruno", "Heather", "Carlos"] -- define a linear list
trace(workerList.max()) -- displays "Heather"
trace(workerList.min()) -- displays "Bruno"

// JavaScript syntax
var workerList = list("Bruno", "Heather", "Carlos"); // define a linear list
trace(workerList.max()); // displays Heather
trace(workerList.min()); // displays Bruno
```

The following statements use `findPos` to get the index position of a specified property in a property list:

```
-- Lingo syntax
-- define a property list
foodList = [#breakfast:"Waffles", #lunch:"Tofu Burger"]
trace(foodList.findPos(#lunch)) -- displays 2

// JavaScript syntax
// define a property list
var foodList = propList("breakfast", "Waffles", "lunch", "Tofu Burger");
trace(foodList.findPos("breakfast")); // displays 1
```

Adding and deleting items in lists

You can add or delete items in a list by using the following methods:

- To add an item at the end of a list, use the `append()` method.
- To add an item at its proper position in a sorted list, use the `add()` or `addProp()` methods.
- To add an item at a specific place in a linear list, use the `addAt()` method.
- To add an item at a specific position in a property list, use the `addProp()` method.
- To delete an item from a list, use the `deleteAt()`, `deleteOne()`, or `deleteProp()` methods.
- To replace an item in a list, use the `setAt()` or `setaProp()` methods.

The following statements use `append()` to add an item to the end of a list:

```
-- Lingo syntax
workerList = ["Bruno", "Heather", "Carlos"] -- define a linear list
workerList.append("David")
trace(workerList) -- displays ["Bruno", "Heather", "Carlos", "David"]

// JavaScript syntax
var workerList = list("Bruno", "Heather", "Carlos"); // define a linear list
workerList.append("David");
trace(workerList); // displays ["Bruno", "Heather", "Carlos", "David"]
```

The following statements use `addProp()` to add a property and an associated value to a property list:

```
-- Lingo syntax
-- define a property list
```

```
foodList = [#breakfast:"Waffles", #lunch:"Tofu Burger"]
foodList.addProp(#dinner, "Spaghetti") -- adds [#dinner: "Spaghetti"]

// JavaScript syntax
// define a property list
var foodList = propList("breakfast", "Waffles", "lunch", "Tofu Burger");
foodList.addProp("dinner", "Spaghetti"); // adds ["dinner": "Spaghetti"]
```

You do not have to explicitly remove lists. Lists are automatically removed when they are no longer referred to by any variable. Other types of objects must be removed explicitly, by setting variables that refer to them to `VOID` (Lingo) or `null` (JavaScript syntax).

Copying lists

Assigning a list to a variable and then assigning that variable to a second variable does not make a separate copy of the list. For example, the first statement below creates a list that contains the names of two continents, and assigns the list to the variable `landList`. The second statement assigns the same list to a new variable `continentList`. In the third statement, adding `Australia` to `landList` also automatically adds `Australia` to the list `continentList`. This happens because both variable names point to the same List object in memory. The same behavior occurs by using an array in JavaScript syntax.

```
-- Lingo syntax
landList = ["Asia", "Africa"]
continentList = landList
landList.add("Australia") -- this also adds "Australia" to continentList
```

Create a copy of a list that is independent of another list

Use the `duplicate()` method.

For example, the following statements create a list and then make an independent copy of the list:

```
-- Lingo syntax
oldList = ["a", "b", "c"]
newList = oldList.duplicate() -- makes an independent copy of oldList

// JavaScript syntax
var oldList = list("a", "b", "c");
var newList = oldList.duplicate(); // makes an independent copy of oldList
```

After `newList` is created, editing either `oldList` or `newList` has no effect on the other.

Sorting lists

Lists are sorted in alphanumeric order, with numbers being sorted before strings. Strings are sorted according to their initial letters, regardless of how many characters they contain. Sorted lists perform slightly faster than unsorted lists.

A linear list is sorted according to the values in the list. A property list is sorted according to the property names in the list or array.

After the values in a linear or property list are sorted, they will remain sorted, even as values are added to or removed from the lists.

Sort a list

Use the `sort()` method.

For example, the following statements sort a nonsorted alphabetical list:

```
-- Lingo syntax
oldList = ["d", "a", "c", "b"]
oldList.sort() -- results in ["a", "b", "c", "d"]

// JavaScript syntax
var oldList = list("d", "a", "c", "b");
oldList.sort(); // results in ["a", "b", "c", "d"]
```

Creating multidimensional lists

You can also create multidimensional lists that enable you to work with the values of more than one list at a time.

In the following example, the first two statements create the separate linear lists `list1` and `list2`. The third statement creates a multidimensional list and assigns it to `mdList`. To access the values in a multidimensional list, the fourth and fifth statements use brackets to access the values in the list; the first bracket provides access to a specified list, and the second bracket provides access to the value at a specified index position in the list.

```
-- Lingo syntax
list1 = list(5,10)
list2 = list(15,20)
mdList = list(list1, list2)
trace(mdList[1][2]) -- displays 10
trace(mdList[2][1]) -- displays 15

// JavaScript syntax
var list1 = list(5,10);
var list2 = list(15,20);
var mdList = list(list1, list2);
trace(mdList[1][2]); // displays 10
trace(mdList[2][1]); // displays 15
```

JavaScript syntax arrays

JavaScript syntax arrays are similar to Lingo-style linear lists in that each element in an array is a single value. One of the main differences between JavaScript syntax arrays and Lingo-style linear lists is that the index into an array always starts with 0.

You create a JavaScript syntax array by using the Array object. You can use either square brackets (`[ ]`) or the `Array` constructor to create an array. The following two statements create an array with two values:

```
// JavaScript syntax
var myArray = [10, 15]; // using square brackets
var myArray = new Array(10, 15); // using the Array constructor
```

You can also create empty arrays. The following two statements create an empty array:

```
// JavaScript syntax
var myArray = [];
var myArray = new Array();
```

Note: *The* Director Scripting Reference *does not include a complete reference for JavaScript syntax Array objects. For more complete information on using Array objects, see one of the many third-party resources on the subject.*

Checking items in arrays

You can determine the characteristics of an array and the number of items the array contains by using the following methods:

- To display the contents of a list, use the `put()` or `trace()` function, passing the variable that contains the list as a parameter.
- To determine the number of items in an array, use the Array object's `length` property.
- To determine an array's type, use the `constructor` property.

  The following example illustrates determining the number of items in an array by using the `length` property, and then returning the type of object by using the `constructor` property:

```
// JavaScript syntax
var x = ["1", "2", "3"];
trace(x.length) // displays 3
trace(x.constructor == Array) // displays true
```

Adding and deleting items in arrays

You can add or delete items in an array by using the following methods:

- To add an item at the end of an array, use the Array object's `push()` method.
- To add an item at its proper position in a sorted array, use the Array object's `splice()` method.
- To add an item at a specific position in an array, use the Array object's `splice()` method.
- To delete an item from an array, use the Array object's `splice()` method.
- To replace an item in an array, use the Array object's `splice()` method.

The following example illustrates using the Array object's `splice()` method to add items to, delete items from, and replace items in an array:

```
// JavaScript syntax
var myArray = new Array("1", "2");
trace(myArray); displays 1,2

myArray.push("5"); // adds the value "5" to the end of myArray
trace(myArray); // displays 1,2,5

myArray.splice(3, 0, "4"); // adds the value "4" after the value "5"
trace(myArray); // displays 1,2,5,4

myArray.sort(); // sort myArray
trace(myArray); // displays 1,2,4,5

myArray.splice(2, 0, "3");
trace(myArray); // displays 1,2,3,4,5

myArray.splice(3, 2); // delete two values at index positions 3 and 4
trace(myArray); // displays 1,2,3

myArray.splice(2, 1, "7"); // replaces one value at index position 2 with "7"
trace(myArray); displays 1,2,7
```

Copying arrays

Assigning an array to a variable and then assigning that variable to a second variable does not make a separate copy of the array.

For example, the first statement below creates an array that contains the names of two continents, and assigns the array to the variable `landList`. The second statement assigns the same list to a new variable `continentList`. In the third statement, adding `Australia` to `landList` also automatically adds `Australia` to the array `continentList`. This happens because both variable names point to the same Array object in memory.

```
// JavaScript syntax
var landArray = new Array("Asia", "Africa");
var continentArray = landArray;
landArray.push("Australia"); // this also adds "Australia" to continentList
```

Create a copy of an array that is independent of another array

Use the Array object's `slice()` method.

For example, the following statements create an array and then use `slice()` to make an independent copy of the array:

```
// JavaScript syntax
var oldArray = ["a", "b", "c"];
var newArray = oldArray.slice(); // makes an independent copy of oldArray
```

After `newArray` is created, editing either `oldArray` or `newArray` has no effect on the other.

Sorting arrays

Arrays are sorted in alphanumeric order, with numbers being sorted before strings. Strings are sorted according to their initial letters, regardless of how many characters they contain.

Sort an array

Use the Array object's `sort()` method.

The following statements sort a non-sorted alphabetical array:

```
// JavaScript syntax
var oldArray = ["d", "a", "c", "b"];
oldArray.sort(); // results in a, b, c, d
```

The following statements sort a non-sorted alphanumeric array:

```
// JavaScript syntax
var oldArray = [6, "f", 3, "b"];
oldArray.sort(); // results in 3, 6, b, f
```

Sorting an array results in a new sorted array.

Creating multidimensional arrays

You can also create multidimensional arrays that enable you to work with the values of more than one array at a time.

In the following example, the first two statements create the separate arrays `array1` and `array2`. The third statement creates a multidimensional array and assigns it to `mdArray`. To access the values in a multidimensional array, the fourth and fifth statements use brackets to access the values in the array; the first bracket provides access to a specified array, and the second bracket provides access to value at a specified index position in the array.

```
// JavaScript syntax
var array1 = new Array(5,10);
var array2 = [15,20];
var mdArray = new Array(array1, array2);
trace(mdArray[0][1]); // displays 10
trace(mdArray[1][0]); // displays 15
```

Chapter 3: Writing Scripts in Director

Scripts in Director® 11 support all kinds of functionality in movies that would not be possible otherwise. As you write scripts, you may find the need for increasingly advanced scripts to support complex interactivity in your Director movies. Intermediate and advanced scripting concepts and techniques are presented here, including information about object-oriented scripting in Director.

If you are new to scripting in Director, make sure to read "Director Scripting Essentials" on page 4 in addition to the topics here.

Choosing between Lingo and JavaScript syntax

Both Lingo and JavaScript syntax provide access to the same objects, events, and scripting APIs. Therefore, it does not necessarily matter which language you choose to write your scripts. Your choice might be as simple as deciding which language you have the most knowledge of and are most comfortable with.

To understand how scripting languages typically work with a given object and event model in Director, consider the following:

- In general, a given scripting language, such as Lingo or JavaScripts syntax, is wrapped around a given object and event model in order to provide access to those objects and events.
- JavaScript is an implementation of the ECMAScript standard that is wrapped around a web browser's object and event model to provide access to the browser's objects and events.
- ActionScript is an implementation of the ECMAScript standard that is wrapped around the Adobe® Flash® object and event model to provide access to Flash objects and events.
- The Director implementation of JavaScript syntax is an implementation of ECMAScript that is wrapped around the Director object and event model to provide access to Director objects and events.
- Lingo is a custom syntax that is wrapped around the Director object and event model to provide access to Director objects and events.

Lingo and JavaScript syntax are the two languages you can use to access the same Director object and event model. Scripts written in one language have the same capabilities as scripts written in the other language.

Therefore, after you know how to access the scripting APIs in one language, you essentially know how to access them in the other language. For example, JavaScript syntax code can access Lingo data types such as symbols, linear lists, property lists, and so on, create and invoke Lingo parent scripts and behaviors, create and invoke Xtra extensions, and use Lingo string chunk expressions. Also, both JavaScript syntax and Lingo scripts can be used within a single movie; however, a single script cast member can contain only one syntax or the other.

There are two main differences between Lingo and JavaScript syntax:

- Each language contains some terminology and syntax conventions that are unique to each language. For example, the syntax for an event handler is different in Lingo than it is in JavaScript syntax:

```
-- Lingo syntax
on mouseDown
    ...
end
```

```
// JavaScript syntax
function mouseDown() {
    ...
}
```

For more information on the terminology and syntax conventions used for each language, see "Scripting terminology" on page 5 and "Scripting syntax" on page 7.

- Some of the scripting APIs are accessed slightly differently in each language. For example, you would use different constructs to access the second word in the first paragraph of a text cast member:

```
-- Lingo syntax
member("News Items").paragraph[1].word[2]

// JavaScript syntax
member("News Items").getPropRef("paragraph", 1).getProp("word", 2);
```

Scripting in dot syntax format

Whether you write scripts in Lingo or JavaScript syntax, you write them by using the dot syntax format. You use dot syntax to access the properties or methods related to an object. A dot syntax statement begins with a reference to an object, followed by a period (dot), and then the name of the property, method, or text chunk that you want to specify. Each dot in a statement essentially represents a move from a higher, more general level in the object hierarchy to a lower, more specific level in the object hierarchy.

For example, the following statement first creates a reference to the cast library named "News Stories", and then uses dot syntax to access the number of cast members in that cast library.

To identify chunks of text, include terms after the dot to refer to more specific items within text. For example, the first statement below refers to the first paragraph of the text cast member named "News Items". The second statement below refers to the second word in the first paragraph.

```
-- Lingo syntax
member("News Items").paragraph[1]
member("News Items").paragraph[1].word[2]

// JavaScript syntax
member("News Items").getPropRef("paragraph", 1);
member("News Items").getPropRef("paragraph", 1).getProp("word", 2);
```

For certain objects that handle cascading property access to either data or a specific cast member type, as illustrated in the previous two statements, access to the properties is not supported by using normal JavaScript syntax. Therefore, you must use the `getPropRef()` and `getProp()` methods to access cascading properties in JavaScript syntax.

There are a few things to note about this JavaScript syntax exception:

- This technique must be applied to 3D objects, text cast members, field cast members, and XML Parser Xtra extensions accessed by using JavaScript syntax.
- You must use the `getPropRef()` method to store a reference to one of the previously mentioned objects or its properties by using JavaScript syntax.
- You must use the `getProp()` method to retrieve a property value of one of the previously mentioned objects or its properties by using JavaScript syntax.

• 3D objects and properties must be accessed by using their fully qualified names in JavaScript syntax. For example, in Lingo, the property `shader` can be used as a shortcut for the property `shaderList[1]`. However, in JavaScript syntax, the property `shaderList[1]` must be used at all times.

Introducing the Director objects

In basic terms, *objects* are logical groupings of named data that also can contain methods that act on that data. In this release of Director, the scripting APIs have been grouped into objects and are accessed through these objects. Each object provides access to a specific set of named data and type of functionality. For example, the Sprite object provides access to the data and functionality of a sprite, the Movie object provides access to the data and functionality of a movie, and so on.

The objects used in Director fall into the following four categories. Depending on the type of functionality you want to add and the part of a movie you are adding functionality to, you will use the objects from one or more of these categories:

- Core objects
- Media types
- Scripting objects
- 3D objects

Core objects

This category of objects provides access to the core functionality and features available in Director, such as the Director player engine, movie windows, sprites, sounds, and so on. They represent the base layer through which all APIs and other object categories are accessed.

There are also a group of top-level methods and properties that enable you to access all of the core objects directly, instead of having to traverse the object hierarchy to access a specific core object.

For a reference of the available core objects and their APIs, see "Director Core Objects" on page 93.

Media types

This category of objects provides access to the functionality of the various media types, such as RealMedia, DVD, Animated GIF, and so on, which are added to movies as cast members.

Literally, media types are not actually objects, but rather cast members that are of a specific type of media. When a media type is added to a movie as a cast member, it not only inherits the functionality of the core Member object, it also extends the Member object by providing additional functionality that is available only to the specified media type. For example, a RealMedia cast member has access to the Member object's methods and properties, but it also has additional methods and properties that are specific to RealMedia. All other media types also exhibit this behavior.

For a reference of the available media types and their APIs, see "Media Types" on page 110.

Scripting objects

This category of objects, also known as *Xtra extensions*, provides access to the functionality of the software components, such as XML Parser, Fileio, SpeechXtra, and so on, that are installed with Director and extend core Director functionality. The preexisting Xtra extensions provide capabilities such as importing filters and connecting to the Internet. If you know the C programming language, you can create your own custom Xtra extensions.

For a reference of the available scripting objects and their APIs, see "Scripting Objects" on page 128.

3D objects

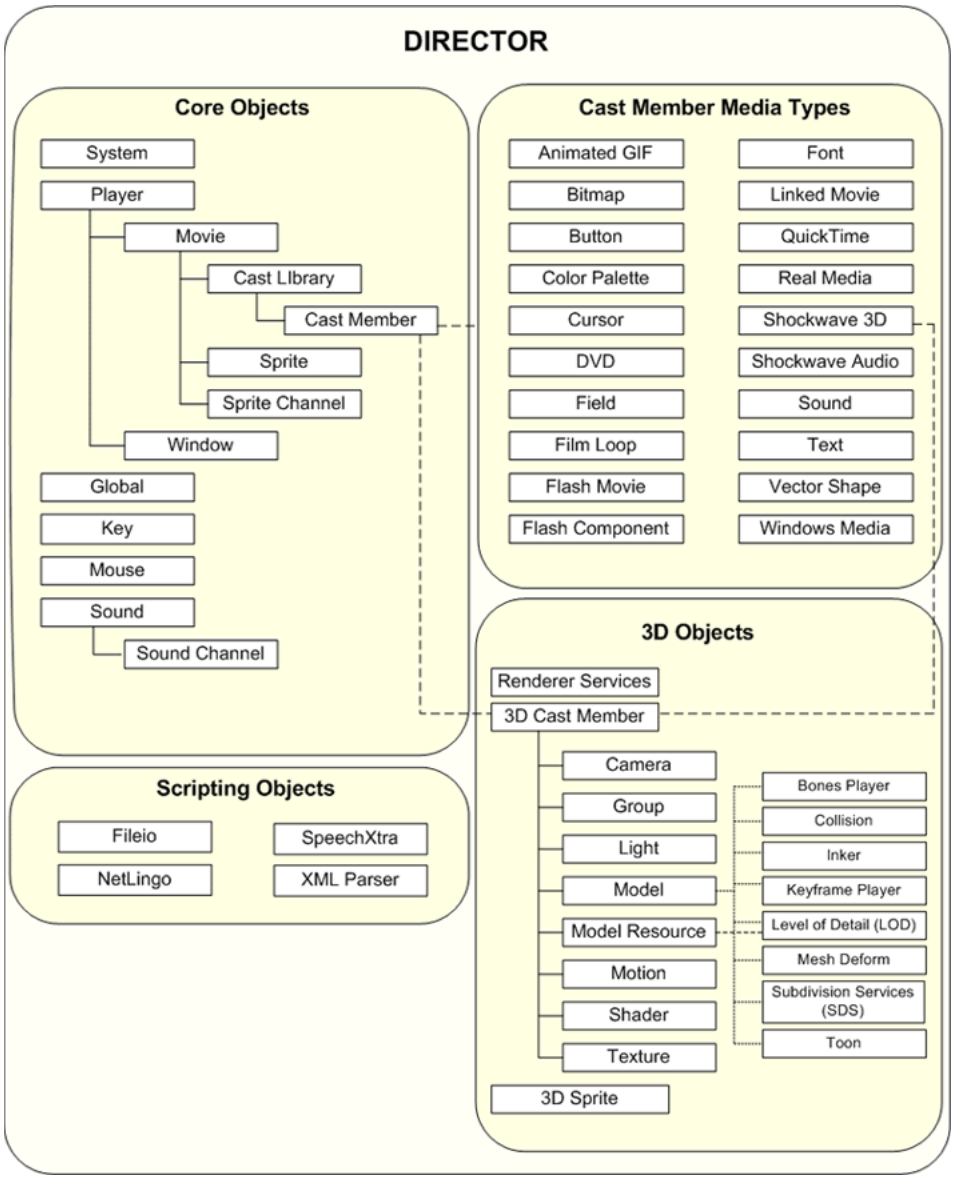
This category of objects provides access to the functionality of the cast members and text that are used to create 3D movies.

For more information about 3D movies, see the Using Director topics in the Director Help Panel.

For a reference of the available 3D objects and their APIs, see "3D Objects" on page 132.

Object model diagrams

The following diagrams illustrate the basic high-level relationships between the object groups and their hierarchies within Director. For information on object creation, properties and methods, and other APIs, see the relevant API reference topics.



Top-level functions and properties

There are a number of top-level functions and properties that provide direct access to the core objects and functionality in Director. You will likely make extensive use of many of these functions and properties as you create references to core objects, new images, lists, and so on. For example, the top-level `_movie` property refers directly to the core Movie object, and the top-level `list()` function creates a linear list.

The following tables list the top-level functions and properties.

Top-level functions	
castLib()	randomVector()
channel() (Top level)	rect()
color()	script()
date() (formats), date() (System)	showLocals()
image()	sound()
isBusy()	sprite()
list()	symbol()
member()	timeout()
point()	trace()
propList()	vector()
put()	window()
random()	xtra()

Top-level properties	
_global	_player
_key	_sound
_mouse	_system
_movie	

Introduction to object-oriented programming in Director

By using either Lingo or JavaScript syntax, you can apply object-oriented programming principles to your scripts. Applying object-oriented principles typically makes programming easier by letting you write less code and letting you use simpler logic to accomplish tasks, in addition to increasing the reusability and modularity of your code.

Depending on the scripting language you are using, you apply these principles using two different paradigms:

- In Lingo, you use parent scripts, ancestor scripts, and child objects to simulate object-oriented programming.
- In JavaScript syntax, you use standard JavaScript-style object-oriented programming techniques to create classes and subclasses.

Each paradigm enables you to apply the advantages of object-oriented programming to your scripts, so it does not really matter which scripting language you are using. You merely apply the principles in different ways.

Because each scripting language uses a different paradigm to apply object-oriented principles, the techniques described for one language won't work in the other language. Therefore, you only need to read the content that applies to the scripting language you are using:

- For more information on simulating object-oriented programming in Lingo, see "Object-oriented programming with Lingo syntax" on page 47.

• For more information on object-oriented programming in JavaScript syntax, see "Object-oriented programming with JavaScript syntax" on page 56.

Object-oriented programming with Lingo syntax

In Lingo, parent scripts provide the advantages of object-oriented programming. You can use parent scripts to generate script objects that behave and respond similarly yet can still operate independently of each other.

You can create multiple copies (or instances) of a parent script by using Lingo. Each instance of a parent script is called a child object. You can create child objects on demand as the movie plays. Director does not limit the number of child objects that can be created from the same parent script. You can create as many child objects as the computer's memory can support.

Director can create multiple child objects from the same parent script, just as Director can create multiple instances of a behavior for different sprites. You can think of a parent script as a template, and child objects as implementations of the parent template.

The discussion about Lingo parent scripts and child objects describes the basics of how to write parent scripts and create and use child objects, and also provides script examples. It does not teach fundamental object-oriented programming concepts; however, to use parent scripts and child objects successfully, you must understand object-oriented programming principles. For an introduction to the basics of object-oriented programming, see one of the many third-party resources on that subject.

Similarity to other object-oriented languages

If you are familiar with an object-oriented programming language such as Java or C++, you may already understand the concepts that underlie parent scripting but know them by different names.

Terms that Director uses to describe parent scripts and child objects correspond to the following common object-oriented programming terms:

Parent scripts in Director correspond to classes in object-oriented programming.

Child objects in Director correspond to instances in object-oriented programming.

Property variables in Director correspond to instance variables or member variables in object-oriented programming.

Handlers in Director correspond to methods in object-oriented programming.

Ancestor scripts in Director correspond to the Super class or base class in object-oriented programming.

Parent script and child object basics

In Lingo, a *parent script* is a set of handlers and properties that define a child object; it is not a child object itself. A *child object* is a self-contained, independent instance of a parent script. Children of the same parent have identical handlers and properties, so child objects in the same group can have similar responses to events and messages.

Typically, parent scripts are used to build child objects that make it easier to organize movie logic. These child objects are especially useful when a movie requires the same logic to be run several times concurrently with different parameters. You can also add a child object to a Sprite object's `scriptInstanceList` or the Movie object's `actorList` as a way to control animation.

Because all the child objects of the same parent script have identical handlers, those child objects respond to events in similar ways. However, because each child object maintains independent values for the properties defined in the parent script, each child object can behave differently than its sibling objects—even though they are instances of the same parent script.

For example, you can create a parent script that defines child objects that are editable text fields, each with its own property settings, text, and color, regardless of the other text fields' settings. By changing the values of properties in specific child objects, you can change any of these characteristics as the movie plays without affecting the other child objects based on the same parent script.

Similarly, a child object can have a property set to either `TRUE` or `FALSE` regardless of that property's setting in sibling child objects.

A parent script refers to the name of a script cast member that contains the property variables and handlers. A child object created from a parent script is essentially a new instance of the script cast member.

Differences between child objects and behaviors

While child objects and behaviors are similar in that they both can have multiple instances, they have some important differences as well. The main difference between child objects and behaviors is that behaviors are associated with locations in the Score because they are attached to sprites. Behavior objects are automatically created from initializers stored in the Score as the playhead moves from frame to frame and encounters sprites with attached behaviors. In contrast, child objects from parent scripts must be created explicitly by a handler.

Behaviors and child objects differ in how they become associated with sprites. Director automatically associates a behavior with the sprite that the behavior is attached to, but you must explicitly associate a child object with a sprite. Child objects do not require sprite references and exist only in memory.

Ancestor basics

Parent scripts can declare *ancestors*, which are additional scripts whose handlers and properties a child object can call on and use.

Ancestor scripting lets you create a set of handlers and properties that you can use and reuse for multiple parent scripts.

A parent script makes another parent script its ancestor by assigning the script to its `ancestor` property. For example, the following statement makes the script What_Everyone_Does an ancestor to the parent script in which the statement occurs:

```
-- Lingo syntax
ancestor = new(script "What_Everyone_Does")
```

When handlers and properties are not defined in a child object, Director searches for the handler or property in the child's ancestors, starting with the child's parent script. If a handler is called or a property is tested and the parent script contains no definition for it, Director searches for a definition in the ancestor script. If a definition exists in the ancestor script, that definition is used.

A child object can have only one ancestor at a time, but that ancestor script can have its own ancestor, which can also have an ancestor, and so on. This lets you create a series of parent scripts whose handlers are available to a child object.

Writing a parent script

A parent script contains the code needed to create child objects and define their possible actions and properties. First, you must decide how you want the child objects to behave. Then, you can write a parent script that does the following:

- Optionally declares any appropriate property variables; these variables represent properties for which each child object can contain a value independent of other child objects.
- Sets up the initial values of the child objects' properties and variables in the `on new` handler.
- Contains additional handlers that control the child objects' actions.

Declaring property variables

Each child object created from the same parent script initially contains the same values for its property variables. A property variable's value belongs only to the child object it's associated with. Each property variable and its value persists as long as the child object exists. The initial value for the property variable is typically set in the `on new` handler; if it's not set, the initial value is `VOID`.

Declare a property variable

Use the `property` keyword at the beginning of the parent script.

Set and test property variables from outside the child object

Set and test property variables in the same way you would any other property in your scripts, by using the syntax *`objectRef.propertyName`*.

For example, the following statement sets the `speed` property of an object `car1`:

```
car1.speed = 55
```

Creating the new handler

Each parent script typically uses an `on new` handler. This handler creates the new child object when another script issues a `new(script` *`parentScriptName`*`)` command, which tells the specified parent script to create a child object from itself. The `on new` handler in the parent script can also set the child object's initial property values, if you want.

The `on new` handler always starts with the phrase `on new,` followed by the `me` variable and any parameters being passed to the new child object.

The following `on new` handler creates a new child object from the parent script and initializes the child's `spriteNum` property with the value passed to it in the `aSpriteNum` parameter. The `return me` statement returns the child object to the handler that originally called the `on new` handler.

```
-- Lingo syntax
property spriteNum

on new me, aSpriteNum
    spriteNum = aSpriteNum
    return me
end
```

For more information on calling the `on new` handlers, see "Creating a child object" on page 50.

Adding other handlers

You determine a child object's behavior by including in the parent script the handlers that produce the desired behavior. For example, you could add a handler to make a sprite change color.

The following parent script defines a value for the property `spriteNum`, and contains a second handler that changes the `foreColor` property of the sprite:

```
-- Lingo syntax
property spriteNum

on new me, aSpriteNum
    spriteNum = aSpriteNum
    return me
end

on changeColor me
    spriteNum.foreColor = random(255)
end
```

Referring to the current object

Typically, one parent script creates many child objects, and each child object contains more than one handler. The special parameter variable `me` tell the handlers in a child object that they are to operate on the properties of that object and not on the properties of any other child object. This way, when a handler within a child object refers to properties, the handler uses its own child object's values for those properties.

The term `me` must always be the first parameter variable stated in every handler definition in a parent script. It is always important to define `me` as the first parameter for parent scripts and to pass the same parameter if you need to call other handlers in the same parent script, since these will be the handlers in each of the script's child objects.

When referring to properties defined in ancestor scripts, you must use the `me` parameter as the source of the reference. This is because the property, while defined in the ancestor script, is nevertheless a property of the child object. For example, the following statement uses `me` to refer to an object and access properties defined in an ancestor of the object:

```
-- Lingo syntax
x = me.y -- access ancestor property y
```

Because the variable `me` is present in each handler of a child object, it indicates that all the handlers control that same child object.

Creating a child object

Child objects exist entirely in memory; they are not saved with a movie. Only parent and ancestor scripts exist on disk.

To create a new child object, you use the `new()` method and assign the child object a variable name or position in a list so you can identify and work with it later.

To create a child object and assign it to a variable, use the following syntax:

```
-- Lingo syntax
variableName = new(script "scriptName", parameter1, parameter2, ...)
```

The *`scriptName`* parameter is the name of the parent script, and *`parameter1`*, *`parameter2`*, `...` are any parameters you are passing to the child object's `on new` handler. The `new()` method creates a child object whose ancestor is *`scriptName`*. It then calls the `on new` handler in the child object with the specified parameters.

You can issue a `new()` statement from anywhere in a movie. You can customize the child object's initial settings by changing the values of the parameters passed with the `new()` statement.

Each child object requires only enough memory to record the current values of its properties and variables and a reference to the parent script. Because of this, in most cases, you can create and maintain as many child objects as you require.

You can produce additional child objects from the same parent script by issuing additional `new()` statements.

You can create child objects without immediately initializing their property variables by using the `rawNew()` method. The `rawNew()` method does this by creating the child object without calling the parent script's `on new` handler. In situations where large numbers of child objects are needed, `rawNew()` allows you to create the objects ahead of time and defer the assignment of property values until each object is needed.

The following statement creates a child object from the parent script Car without initializing its property variables and assigns it to the variable `car1`:

```
-- Lingo syntax
car1 = script("Car").rawNew()
```

To initialize the properties of one of these child objects, call its `on new` handler:

```
car1.new
```

Checking child object properties

You can check the values of specific property variables in individual child objects by using a simple *`objectName.propertyName`* syntax. For example, the following statement assigns the variable `x` the value of the `carSpeed` property of the child object in the variable `car1`:

```
-- Lingo syntax
x = car1.carSpeed
```

Querying object properties from outside the objects themselves can be useful for getting information about groups of objects, such as the average speed of all the Car objects in a racing game. You might also use the properties of one object to help determine the behavior of other objects that are dependent on it.

In addition to checking the properties that you assign, you can check whether a child object contains a specific handler or find out which parent script an object came from. This is useful when you have objects that come from parent scripts that are similar but that have subtle differences.

For example, you may want to create a scenario in which one of several parent scripts is used to create a child object. You can then determine which parent script a particular child object came from by using the `script()` function, which returns the name of an object's parent script.

The following statements check whether the object `car1` was created from the parent script named `Car`:

```
-- Lingo syntax
if car1.script = script("Car") then
    _sound.beep()
end if
```

You can also get a list of the handlers in a child object by using the `handlers()` method, or check whether a particular handler exists in a child object by using the `handler()` method.

The following statement places a list of the handlers in the child object `car1` into the variable `myHandlerList`:

```
-- Lingo syntax
myHandlerList = car1.handlers()
```

The resulting list would look something like this:

```
[#start, #accelerate, #stop]
```

The following statements use the `handler()` method to check whether the handler `on accelerate` exists in the child object `car1`:

```
-- Lingo syntax
if car1.handler(#accelerate) then
    put("The child object car1 contains the handler named on accelerate.")
end if
```

Removing a child object

You can remove a child object from a movie by setting all variables that contain a reference to the child object to another value. If the child object has been assigned to a list, such as `actorList`, you must also remove the child object from the list.

Remove a child object and the variables that refer to it

Set each variable to `VOID`.

Director deletes the child object when there are no more references to it. In the following example, `ball1` contains the only reference to a specific child object, and it is set to `VOID` to delete the object from memory:

```
-- Lingo syntax
ball1 = VOID
```

Remove an object from actorList

Use the `delete()` method to delete the item from the list.

Using scriptInstanceList

You can use the `scriptInstanceList` property to dynamically add new behaviors to a sprite. Normally, `scriptInstanceList` is the list of behavior instances created from the behavior initializers defined in the Score. If you add child objects created from parent scripts to this list, the child objects receive the messages sent to other behaviors.

For example, the following statement adds a child object to the `scriptInstanceList` property of sprite 10:

```
-- Lingo syntax
add(sprite(10).scriptInstanceList, new(script "rotation", 10))
```

The following is a possible parent script that the previous statement refers to:

```
-- Lingo syntax parent script "rotation"
property spriteNum

on new me, aSpriteNum
    spriteNum = aSpriteNum
    return me
end

on prepareFrame me
    sprite(spriteNum).rotation = sprite(spriteNum).rotation + 1
end
```

When a child object is added to `scriptInstanceList`, you must initialize the child object's `spriteNum` property. Typically, you do this from a parameter passed in to the `on new` handler.

Note: *The `beginSprite` message is not sent to dynamically added child objects.*

For reference information on `scriptInstanceList`, see "scriptInstanceList" on page 984.

Using actorList

You can set up a special list of child objects (or any other objects) that receives its own message each time the playhead enters a frame or the `updateStage()` method updates the Stage.

The special list is `actorList`, which contains only objects that have been explicitly added to the list.

The message is the `stepFrame` message that is sent only when the playhead enters a frame or the `updateStage()` command is used.

Objects in `actorList` receive a `stepFrame` message instead of an `enterFrame` message at each frame. If the objects have an `on stepFrame` handler available, the script in the handler runs each time the playhead enters a new frame or the `updateStage()` method updates the Stage.

Some possible uses of `actorList` and `stepFrame` are to animate child objects that are used as sprites or to update a counter that tracks the number of times the playhead enters a frame.

An `on enterFrame` handler could achieve the same results, but the `actorList` property and `stepFrame` handler are optimized for performance in Director. Objects in `actorList` respond more efficiently to `stepFrame` messages than to `enterFrame` messages or custom messages sent after an `updateStage()` method.

Add an object to the actorList

Use the `actorList` property as follows, where *`childObject`* is a reference to the child object to add:

```
-- Lingo syntax
_movie.actorList.add(childObject)
```

The object's `stepFrame` handler in its parent or ancestor script then runs automatically each time the playhead advances. The object is passed as the first parameter, `me`, to the `on stepFrame` handler.

Director does not clear the contents of `actorList` when branching to another movie, which can cause unpredictable behavior in the new movie. If you do not want child objects in the current movie to be carried over into the new movie, insert a statement that clears `actorList` in the `on prepareMovie` handler of the new movie.

Clear child objects from actorList

Set `actorList` to `[ ]`, which is an empty list.

For reference information on `actorList`, see "actorList" on page 634.

Creating Timeout objects

A Timeout object is a script object that acts like a timer and sends a message when the timer expires. This is useful for scenarios that require specific things to happen at regular time intervals or after a particular amount of time has elapsed.

Timeout objects can send messages that call handlers inside child objects or in movie scripts. You create a Timeout object by using the `new()` keyword. You must specify a name for the object, a handler to be called, and the frequency with which you want the handler to be called. After a Timeout object is created, Director keeps a list of currently active Timeout objects, called `timeOutList`.

The syntax described below is necessary for all new movies authored in Adobe Director 11, or for older movies playing in Adobe Director 11 whose `scriptExecutionStyle` property has been set to a value of 10. Movies authored in Director MX and earlier have a `scriptExecutionStyle` property set to a value of 9, which allows you to use the syntax found in Director MX and earlier.

Create Timeout objects

```
-- Lingo syntax when scriptExecutionStyle is set to 9
variableName = timeout().new(timeoutName, timeoutPeriod, #timeoutHandler, {,
targetObject})

-- Lingo syntax when scriptExecutionStyle is set to 10
variableName = timeout().new(timeoutName, timeoutPeriod, timeoutHandler, targetObject)
variableName = new timeout(timeoutName, timeoutPeriod, timeoutHandler, targetObject)

// JavaScript syntax
variableName = new timeout(timeoutName, timeoutPeriod, timeoutFunction, targetObject)
```

This statement uses the following elements:

- *variableName* is the variable you are placing the Timeout object into.
- timeout indicates which type of Lingo object you are creating.
- *timeoutName* is the name you give to the Timeout object. This name appears in the timeOutList. It is the #name property of the object.
- new creates a new object.
- *intMilliseconds* indicates the frequency with which the Timeout object should call the handler you specify. This is the #period property of the object. For example, a value of 2000 calls the specified handler every 2 seconds.
- #*handlerName* is the name of the handler you want the object to call. This is the #timeOutHandler property of the object. You represent it as a symbol by preceding the name with the # sign. For example, a handler called on accelerate would be specified as #accelerate.
- *targetObject* indicates which child object's handler should be called. This is the #target property of the object. It allows specificity when many child objects contain the same handlers. If you omit this parameter, Director looks for the specified handler in the movie script.

The following statement creates a Timeout object named timer1 that calls an on accelerate handler in the child object car1 every 2 seconds:

```
-- Lingo syntax
myTimer = timeOut("timer1").new(2000, #accelerate, car1)
```

To determine when the next timeout message will be sent from a particular Timeout object, check its #time property. The value returned is the point in time, in milliseconds, when the next timeout message will be sent. For example, the following statement determines the time when the next timeout message will be sent from the Timeout object timer1 and displays it in the Message window:

```
-- Lingo syntax
put(timeout("timer1").time)
```

Using timeOutList

When you begin creating Timeout objects, you can use timeOutList to check the number of Timeout objects that are active at a particular moment.

The following statement sets the variable x to the number of objects in timeOutList by using the count property:

```
-- Lingo syntax
x = _movie.timeoutList.count
```

You can also refer to an individual Timeout object by its number in the list.

The following statement deletes the second Timeout object in timeOutList by using the forget() method:

```
-- Lingo syntax
```

```
timeout(2).forget()
```

Relaying system events with Timeout objects

When you create Timeout objects that target specific child objects, you enable those child objects to receive system events. Timeout objects relay these events to their target child objects. The system events that can be received by child objects include `prepareMovie`, `startMovie`, `stopMovie`, `prepareFrame`, and `exitFrame`. By including handlers for these events in child objects, you can make the child objects respond to them for whatever purposes you see fit. System events received by child objects are also received by movie scripts, frame scripts, and other scripts designed to respond to them.

The following parent script contains a handler for the system event `exitFrame` and a custom handler `slowDown`:

```
-- Lingo syntax
property velocity

on new me
    velocity = random(55)
end

on exitFrame
    velocity = velocity + 5
end

on slowDown mph
    velocity = velocity - mph
end
```

Associating custom properties with Timeout objects

If you want to associate custom properties with a Timeout object, you may want to create a Timeout object that uses as a target anything other than a reference to a script instance object. When you use this technique, the target data becomes data that is associated with the Timeout object, and can be used in your timeout handler.

The following example illustrates using this technique:

```
-- Lingo syntax
-- initialize a timeout object and pass it a data property list (tData)
-- instead of a reference to a script instance object
tData = [#beta: 0]
tTO = timeout("betaData").new(50,#targetHandler,tData)

-- within a movie script, create the targetHandler handler
on targetHandler (aData)
    -- increment and display the beta property
    tData.beta = tData.beta + 1
    put(tData.beta)
end targetHandler
```

In the previous example, the `beta` property keeps incrementing. This means that you can initialize several Timeout objects that all call the same movie script handler, and each Timeout object can have its own data list associated with it.

In general, keep the following in mind:

- When using a reference to a script instance as a target, the target handler in that particular script instance is called. This technique does not allow the use of custom properties.
- When using a reference to anything other than a script instance (such as a property list) as a target, the target handler in a movie script is called. This technique allows the use of custom properties.

Object-oriented programming with JavaScript syntax

Object-oriented programming in JavaScript syntax is somewhat different than it is in other object-oriented languages such as Java and C++—while some object-oriented languages are class-based, JavaScript syntax is prototype-based.

The following two bullet points compare and contrast, at a high level, class-based languages with prototype-based languages such as JavaScript syntax:

- In class-based languages, you create class definitions that define the initial properties and methods that characterize all instances created from those classes. A class definition contains special methods, called *constructor methods*, that are used to create the instances of that class. When an instance is created by using the `new` operator in association with a particular constructor method, that instance inherits all the properties of its parent class. That instance can also perform other processing specific to that instance depending on the constructor that was called.

  In a class definition, you perform inheritance by creating a subclass that inherits all the properties of its parent class, in addition to defining new properties and optionally modifying inherited ones. The parent class from which a subclass is created is also known as a *superclass*.

- In prototype-based languages, such as JavaScript syntax, there is no distinction between classes, instances, subclasses, and so on—they are all known as objects. Instead of using class definitions, in JavaScript syntax you use *prototype objects* as the template from which new objects are created. Similar to class-based languages, in JavaScript syntax, you create a new object by using the `new` operator in association with a constructor function.

  Instead of using superclasses and subclasses, in JavaScript syntax, you associate prototype objects with constructor functions to perform inheritance. This process is very similar to using superclasses and subclasses, only with different terminology.

  Also, as opposed to class-based languages, in JavaScript syntax you can add and remove properties from an object or set of objects at runtime. For example, if you add a property to a prototype object at runtime, any instance objects for which it is a prototype also get that property.

Object-oriented terminology

Because all types in JavaScript syntax are known as objects, class-based terms such as *superclass*, *subclass*, *class*, *instance*, and so on do not have literal technical meanings in JavaScript syntax. However, all of these terms essentially map to objects in JavaScript syntax and are convenient to use generically when referring to the different types of JavaScript syntax objects. Therefore, these class-based terms are used interchangeably with *object* throughout the discussion about object-oriented programming in JavaScript syntax to mean the following:

- **superclass** Any class from which subclasses (objects) are created; a parent class.
- **subclass** Any class that has been created from a superclass (object); a child class.
- **class** A generic term for a superclass or subclass; a parent or child class.
- **instance** or **object instance** A single object that has been created from a superclass.

Custom classes

One of the major advantages of object-oriented programming is the ability to create your own custom classes that enable you to add custom functionality to your scripts. The predefined classes provided by JavaScript syntax, such as Object, String, Math, and so on are useful in some cases, but they may not provide the functionality you require to accomplish your task. For example, suppose you want some objects in your movie to represent types of transportation, such as cars, boats, planes, and so on, and that you want each category to exhibit unique characteristics and functionality. Neither the predefined JavaScript syntax classes nor the predefined Director objects may directly provide the functionality that you need. Therefore, you may want to create a new class for each type of transportation so you can define unique characteristics for each type.

Keep in mind that when you create custom JavaScript syntax classes, you still have access to all the features and functionality of the predefined Director objects. This means that although the predefined Director objects may not directly provide the functionality that you need, you can still use them in your custom classes to access their values and predefined functionality.

Constructor functions

In JavaScript syntax, a constructor function represents the class that contains the template from which new object instances are created. Constructor functions create and initialize (set the default state of) properties in the new objects.

Constructor functions are essentially identical in format to regular JavaScript syntax method functions. The difference between a constructor function and a method function is that a constructor function uses the special `this` keyword to represent a reference to the new object that is being initialized. A method function typically only performs some action on a given set of an object's data.

The following example illustrates one way to create a Rectangle constructor function that could be used to initialize the height and width of new Rectangle objects:

```
function Rectangle(w, h) {
    this.width = w;
    this.height = h;
}
```

You can also create a constructor function by using *function literal* syntax. Function literal syntax provides the same functionality as the syntax used previously and is merely an alternative way to write the constructor. The following example illustrates using function literal syntax to create a Rectangle constructor function similar to the one illustrated previously:

```
Rectangle = function(w, h) {
    this.width = w;
    this.height = h;
}
```

Note: *When defining constructor functions that apply to a movie, be sure to place them in a movie script so they are available globally.*

It is considered good scripting practice to give constructor functions names that map to their functionality, and to use initial capitalization in their names, such as `Rectangle` or `Circle`.

Constructor functions are typically used only to initialize new objects but can also return the object if desired. If you do return the initialized object, the returned object becomes the value of the `new` expression.

Object instances

The most common way to create a new object instance is to use the `new` operator followed by the name of a constructor function. The following examples create new object instances:

```
var objRandom = new Object(); // assigns a reference to an Object object
var objString = new String(); // assigns a reference to a String object
```

A constructor function can optionally define parameters that a new object instance passes to it to initialize the state of the object instance. If a constructor function does define parameters used during initialization of new object instances, the property values are initialized as follows:

- If you pass values to the constructor function during initialization, the properties that received values are set to those values.
- If you do not pass values to the constructor function during initialization, the properties that did not receive values are set to `undefined`.

When you create new object instances, the keyword `this` is used in the body of the associated constructor function to refer to the new object instance. Therefore, a new object instance is initialized with all of the properties defined by using the `this.`*`propertyName`* syntax.

In the following example, a `Circle` constructor function uses the keyword `this` to specify the names of three properties that will be associated with new object instances. The statement following the constructor initializes a new object instance by passing values to the constructor. These values are used as the initial values of the properties specified by the keyword `this`.

```
// Circle constructor function
function Circle(x, y, r) {
    this.xCoord = x;
    this.yCoord = y;
    this.radius = r;
}

// xCoord = 10, yCoord = 15, radius = 5
var objCircle = new Circle(10, 15, 5);
```

Now that `objCircle` has been initialized, you can access its properties. Using the `objCircle` instance created previously, you could set some variables equal to the values of its properties.

```
var theXCoord = objCircle.xCoord; // assigns the value 10 to theXCoord
var theYCoord = objCircle.yCoord; // assigns the value 15 to theYCoord
var theRadius = objCircle.radius; // assigns the value 5 to theRadius
```

Note: *For more information on using dot syntax to access properties and methods of an object, see* *"Scripting in dot syntax format" on page 42**.*

It is considered good scripting practice to give new objects names that map to their functionality, and to name them by using lowercase letters, such as `objRectangle` or `objCircle`.

You can also create an object instance by using *object literal* syntax, which eliminates the need for the `new` operator and a constructor function. You typically only use this technique when you need only one instance of an object that has not been defined in a constructor function. The following example creates an object instance with `x = 1`, `y = 2`, and `radius = 2`:

```
var objSmallCircle = { x:1, y:2, radius:2 };
```

Object inheritance

In addition to being able to create your own custom classes, another major advantage of object-oriented programming is the ability of subclasses to inherit the properties and methods of the superclasses from which they were created. Inheritance enables you to easily create objects that already have built-in properties and functionality.

In JavaScript syntax, there is one superclass that acts as the base class from which all other subclasses are created—the Object superclass. The Object superclass contains a few basic properties and methods. The subclasses that are created by using Object as a template always inherit these basic properties and methods, and likely define their own properties and methods. Subclasses of these classes inherit from Object, from their superclasses, and so on. All additional objects that you create would continue this chain of inheritance.

For example, Object contains the `constructor` property and the `toString()` method. If you create a new class named `SubObj1`, it is a subclass of Object, and therefore automatically inherits the `constructor` property and the `toString()` method of Object. If you then create another class named `SubObj2` using `SubObj1` as a superclass, `SubObj2` would also inherit the `constructor` property and the `toString()` method of Object, in addition to any custom properties and methods you defined in `SubObj1`.

Two of the important properties that your custom constructor functions inherit from the Object superclass are `prototype` and `constructor`. The `prototype` property represents the prototype object of a class, which enables you to add variables (properties) and methods to object instances, and is the means by which inheritance is typically implemented in JavaScript syntax. The `constructor` property represents the constructor function itself. The use of these properties is explained in the following sections.

Prototype objects

As previously stated, when you create a subclass, it automatically inherits the properties and methods of the superclass on which it is based. In JavaScript syntax, inheritance is typically implemented by using prototype objects. A subclass actually inherits its properties and methods from the prototype object of its superclass, and not from the superclass itself. This important point offers a distinct advantage: all properties and methods do not literally have to be copied from a class to an object instance of that class, which can dramatically decrease the amount of memory required by new object instances.

Every class in JavaScript syntax, including the predefined Object class, contains only one prototype object. Every object instance created from a class has access to the properties and methods in the prototype object of that class. Therefore, the prototype object of a class is typically the only object that actually stores the properties and methods for that class; an object instance only contains the properties required to initialize that instance.

In your code, it appears that each object instance actually contains those properties and methods because you can access them directly from each object instance, but the instance is actually using the prototype object to access them. The prototype object of a class is created automatically when you create the class. You access the prototype object by using the `prototype` property of the class.

Because a prototype object of a class stores properties that are shared by all object instances, they are ideally suited to define properties and methods whose values will be shared across all object instances. By sharing properties and methods across object instances, you can easily create instances that exhibit a defined default behavior and can also customize any instances that deviate from the default behavior.

Prototype objects typically are not suited to define properties and methods whose values may vary across object instances. In cases where values may vary across object instances, you typically define those properties and methods within the class itself.

To specify the scope of a custom property or method, you define it as one of the following four types:

Instance variables

Instance variables are any variables (properties) that are defined in a constructor function and are copied into each object instance of that constructor. All object instances have their own copies of instance variables. This means that if there are five object instances of a `Circle` class, there are five copies of each instance variable defined in the class. Because each object instance has its own copy of an instance variable, each object instance can assign a unique value to an instance variable without modifying the values of other copies of the instance variable. You access instance variables directly from their containing object instances.

The following example defines four instance variables—`make`, `model`, `color`, and `speed`—in a constructor function. These four instance variables are available directly from all object instances of the `Car` constructor:

```
function Car(make, model, color) { // define a Car class
    this.make = make;
    this.model = model;
    this.color = color;
    this.speed = 0;
}
```

The following object instance `objCar` contains all four instance variables. Although a value for the instance variable `speed` is not passed to the `Car` constructor, `objCar` still has a `speed` property whose initial value is 0 because the `speed` variable is defined in the `Car` constructor.

```
// objCar.make="Subaru", objCar.model="Forester",
// objCar.color="silver", objCar.speed = 0
var objCar = new Car("Subaru", "Forester", "silver");
```

Instance methods

Instance methods are any methods that are accessible through an object instance. Object instances do not have their own copies of instance methods. Instead, instance methods are first defined as functions, and then properties of the constructor function's prototype object are set to the function values. Instance methods use the keyword `this` in the body of the defining constructor function to refer to the object instance they are operating on. Although a given object instance does not have a copy of an instance method, you still access instance methods directly from their associated object instances.

The following example defines a function named `Car_increaseSpeed()`. The function name is then assigned to the `increaseSpeed` property of the `Car` class's prototype object:

```
// increase the speed of a Car
function Car_increaseSpeed(x) {
    this.speed += x;
    return this.speed;
}
Car.prototype.increaseSpeed = Car_increaseSpeed;
```

An object instance of `Car` could then access the `increaseSpeed()` method and assign its value to a variable by using the following syntax:

```
var objCar = new Car("Subaru", "Forester", "silver");
var newSpeed = objCar.increaseSpeed(30);
```

You can also create an instance method by using function literal syntax. Using function literal syntax eliminates the need to define a function and the need to assign a property name to the function name.

The following example uses function literal syntax to define an `increaseSpeed()` method that contains the same functionality as the `increaseSpeed()` function defined previously:

```
// increase the speed of a Car
Car.prototype.increaseSpeed = function(x) {
    this.speed += x;
    return this.speed;
}
```

Class variables

Also known as *static* variables, these are any variables (properties) that are associated with a class, and not an object instance. There is always only one copy of a class variable, regardless of the number of object instances that are created from that class. Class variables do not use the prototype object to implement inheritance. You access a class variable directly through the class, and not through an object instance; you must define a class in a constructor function before you can define class variables.

The following example defines two class variables—`MAX_SPEED` and `MIN_SPEED`:

```
function Car() { // define a Car class
    ...
}

Car.MAX_SPEED = 165;
Car.MIN_SPEED = 45;
```

You would access the `MAX_SPEED` and `MIN_SPEED` class variables directly from the `Car` class.

```
var carMaxSpeed = Car.MAX_SPEED; // carMaxSpeed = 165
var carMinSpeed = Car.MIN_SPEED; // carMinSpeed = 45
```

Class methods

Also known as *static* methods, these are any methods that are associated with a class, and not an object instance. There is always only one copy of a class method, regardless of the number of object instances that are created from that class. Class methods do not use the prototype object to implement inheritance. You access a class method directly through the class, and not through an object instance; you must define a class in a constructor function before you can define class methods.

The following example defines a function named `setInitialSpeed()` that can change the default speed of new car instances. The function name is assigned to the `setInitialSpeed` property of the `Car` class:

```
function Car(make, model, color) { // define a Car class
    this.make = make;
    this.model = model;
    this.color = color;
    this.speed = Car.defaultSpeed;
}
Car.defaultSpeed = 10; // initial speed for new Car instances
// increase the speed of a Car
function Car_setInitialSpeed(x) {
    Car.defaultSpeed = x;
}
Car.setInitialSpeed = Car_setInitialSpeed;
```

You access the `setInitialSpeed()` class method directly from the `Car` class.

```
var newSpeed = Car.setInitialSpeed(30);
```

You can also create a class method by using function literal syntax. The following example uses function literal syntax to define a `setInitialSpeed()` method that contains the same functionality as the `setInitialSpeed()` function defined previously:

```
// increase the speed of a Car
Car.setInitialSpeed = function(x) {
    Car.defaultSpeed = x;
}
```

Recommended steps for defining a class

The following list describes the recommended steps to follow when defining a class:

1 Define a constructor function that is used as the template from which all object instances are initialized. You may additionally define any instance variables in the constructor function by using the keyword `this` to refer to an object instance.

2 Define any instance methods, and possibly additional instance variables, that are stored in the prototype object of a class. These instance methods and variables are available to all object instances and are accessible through the prototype object of the class.

3 Define any class methods, class variables, and constants that are stored in the class itself. These class methods and variables are accessible only through the class itself.

In your code, when you access a property of an object instance, JavaScript syntax searches the object instance itself for that property. If the instance does not contain the property, JavaScript syntax then searches the prototype object of the superclass from which the instance was created. Because an object instance is searched before the prototype object of the class from which it was created, object instance properties essentially hide properties from the prototype object of their superclasses. This means that both an object instance and its superclass could realistically define a property with the same name but different values.

Deleting variables

You can delete a class variable or an instance variable by using the `delete` operator. The following example illustrates this process.

```
function Car() { // define a Car constructor function
    ...
}
Car.color = "blue"; // define a color property for the Car class
Car.prototype.engine = "V8"; // define an engine property for the prototype

var objCar = new Car();

trace(Car.color); // displays "blue"
trace(objCar.engine); // displays "V8"

delete Car.color;
delete Car.prototype.engine;

trace(Car.color); // displays undefined
trace(objCar.engine); // displays undefined
```

Accessing the constructor property of a prototype object

When you define a class by creating a constructor function, JavaScript syntax creates a prototype object for that class. When the prototype object is created, it initially includes a `constructor` property that refers to the constructor function itself. You can use the `constructor` property of a prototype object to determine the type of any given object.

In the following example, the `constructor` property contains a reference to the constructor function used to create the object instance. The value of the `constructor` property is actually a reference to the constructor itself and not a string that contains the constructor's name:

```
function Car() { // define a Car class
    // initialization code here
}
var myCar = new Car(); // myCar.constructor == function Car() {}
```

Creating properties dynamically

Another advantage of using prototype objects to implement inheritance is that properties and methods that are added to a prototype object are automatically available to object instances. This is true even if an object instance was created before the properties or methods were added.

In the following example, the `color` property is added to the prototype object of a `Car` class after an object instance of `Car` has already been created:

```
function Car(make, model) { // define a Car class
    this.make = make;
    this.model = model;
}
var myCar = new Car("Subaru", "Forester"); // create an object instance

trace(myCar.color); // returns undefined

// add the color property to the Car class after myCar was initialized
Car.prototype.color = "blue";

trace(myCar.color); // returns "blue"
```

You can also add properties to object instances after the instances have been created. When you add a property to a specific object instance, that property is available only to that specific object instance. Using the `myCar` object instance created previously, the following statements add the `color` property to `myCar` after it has already been created:

```
trace(myCar.color); // returns undefined

myCar.color = "blue"; // add the color property to the myCar instance

trace(myCar.color); // returns "blue"

var secondCar = new Car("Honda", "Accord"); // create a second object instance

trace(secondCar.color); // returns undefined
```

Writing scripts in the Script window

When you write scripts for a movie, the quantity and variety of scripts can be very large. Deciding which methods or properties to use, how to structure scripts effectively, and where scripts should be placed requires careful planning and testing, especially as the complexity of your movie grows.

Before you begin writing scripts, formulate your goal and understand what you want to achieve. This is as important—and typically as time-consuming—as developing storyboards for your work.

When you have an overall plan for the movie, you are ready to start writing and testing scripts. Expect this to take time. Getting scripts to work the way you want often takes more than one cycle of writing, testing, and debugging.

The best approach is to start simple and test your work frequently. When you get one part of a script working, start writing the next part. This approach helps you identify bugs efficiently and ensures that your scripts are solid as they become more complex.

The Script window provides a number of features that help you create and modify your scripts.

The Script window in Director allows you to add advanced, scripting-based interactivity to movies. In the Script window, you can code using either Lingo or JavaScript syntax. Lingo is the traditional scripting language of Director. JavaScript syntax was recently introduced to support multimedia developers who prefer working with JavaScript.

By scripting in the Script window, you can accomplish many of the same tasks that you can in the graphical interface of Director — such as moving sprites on the Stage or playing sounds. But much of the usefulness of scripting is in the flexibility that it brings to a movie. Instead of playing a series of frames exactly as the Score dictates, a movie can have scripts that control frame play in response to specific conditions and events.

Note: *In addition to the Script window, where you can create your own scripts, Director includes a set of prepackaged instructions (called* behaviors*) that you can simply drag to sprites and frames. Behaviors let you add script-based interactivity without writing scripts. For more information on behaviors, see the Behaviors topics in the Director Help Panel.*

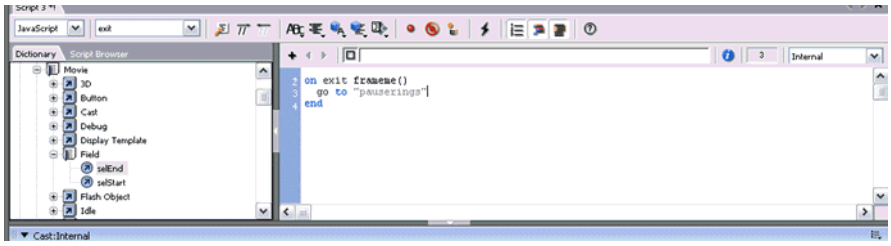
The Script window has an Explorer panel and a Script Editor. By default, the Explorer panel appears to the left of the Script Editor. You can view the Explorer panel in the Dictionary view or the Script Browser view. To set the default position of the Explorer panel, use the Script Window Preferences dialog box (Edit > Preferences > Script).

Dictionary view

The Dictionary view displays a list of built-in Lingo script/JavaScript functions arranged in a tree structure. The functions are classified based on their category, and alphabetically as an index.

Use the Dictionary view to do the following:

- Browse built-in functions for Lingo scripts and JavaScripts.
- Use the built-in functions to create scripts.



Browse functions and create scripts using the Dictionary view

1 Select Window > Script. The Script window appears.

2 Click the Dictionary tab.

3 Select Lingo or JavaScript from the pop-up menu. The corresponding built-in functions are displayed in the panel below.

Lingo: Displays functions for Lingo scripts, 3D Lingo Scripts, and scripting Xtras used in the current movie. The functions are organized as categories (Global, Movie, Player, and so on.)

JavaScript: Displays functions for JavaScripts, 3D JavaScripts, and scripting Xtras used in the current movie. The functions are organized as categories (Global, Movie, Player, and so on.)

4 Expand each category to display its associated functions by clicking the plus sign (+) next to it. To view functions in an alphabetical order, expand the index category.

5 To add a function to the Script Editor for creating scripts, double-click the function.

6 Save the script.

7 Click the Recompile All Modified Scripts icon.

Script Browser view

The Script Browser view displays the scripts and associated handlers that have been used in the movie. You can create new scripts and handlers in this view.

Use the Script Browser view to do the following:

- Browse scripts and handlers in the current movie as a tree or a list.
- Sort scripts based on their Script name, Cast name, Cast number, or Script type in the list view.
- Locate a handler in the Script Editor.
- Create scripts under each script type or script cast.

Browse and create scripts using the Script Browser view

1 Select Window > Script to open the Script window.

2 Click the Script Browser tab. The scripts are displayed in a tree structure in the panel below. To display the scripts as a list, click the Script Browser View button.

Tree view: Scripts are categorized on the basis of the Script type (such as Behavior Scripts, Movie Scripts, Parent Scripts, and the Cast Library) that they are created in. Cast Member Scripts are also listed here. To open a script in the Script Editor, double-click the script. Handlers belonging to a script appear as a tree under the relevant script node.

3 To view a list of compiled handlers in the script, expand the script <name> node. Uncompiled handlers are not displayed.

4 To locate a handler in the Script Editor, double-click the handler. The handler is highlighted in the Script Editor. Alternatively, click the Go to Handler icon in the scripting bar. For more information, see "Finding handlers and text in scripts" on page 70.

5 To add or remove a comment in the script, click the Comment Or Uncomment icon in the scripting bar.

6 To toggle a breakpoint, click the Toggle Breakpoint icon in the scripting bar. Alternatively, press F9 or click the blue bar next to the piece of code.

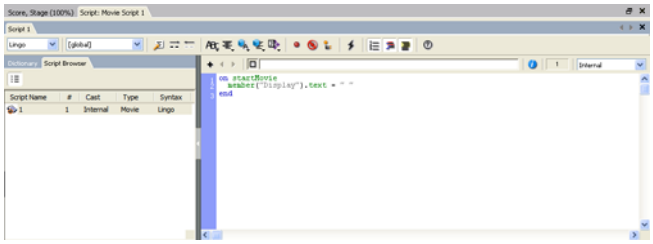
7 **List view:** Scripts are organized in a column in a list on the basis of Script name, Cast name, Cast number, and Type. To sort a list in a column, click the desired column header.

8 To create a new script in the tree view, right-click a script type, and select Add New <script type> from the pop-up menu. When you enter a name for the script in the Script Editor, the name of the script is displayed in the list.

9 Save the script.

10 Click the Recompile All Modified Scripts icon.

Cast Member Scripts are listed in the Cast Member Scripts folder in the tree view and as Script Names in the list view. Because these scripts are attached to a specific cast member and are not actual cast members, they are removed from the Script Browser only when you delete the associated cast member.



For information on alternative ways to create and open scripts, see "Performing common tasks" on page 71.

Note: *To close a Script tab, click the X button on the tab, or right-click the tab area and then select the Close <script type:script name>option.*

Opening and closing multiple scripts

Multiple scripts can be opened as different tabs in the Script window. Because the Script window can accommodate only a fixed number of tabs, some of the tabs might be hidden. To navigate to the hidden tabs, use the '>' and '<' Browse Script tab icons.

Open multiple Script windows

Do one of the following:

- From the Script window, select Windows > New Script Window.
- Press Alt+w+n.

You can use the Alt+w+n shortcut key to open additional instances of any active window. For example, if you have opened the Vector window, use the Alt+w+n shortcut key to open additional Vector windows.

Close a Script tab

1 Click the tab of the Script window that you want to close.

2 Do one of the following:

- Click the tab to close (if not already the active tab) and then click the X button.
- Right-click the tab or the area next to the tab and select Close <script type:script name>.

Setting Script window preferences

You can change the font of text in the Script window and define different colors for various code components. To change the default font of text in the Script window and the color of various code elements, you use Script window preferences. Director automatically colors different types of code elements unless you turn off Auto Coloring.

To display the Explorer panel in the other windows, click the arrow icon in the splitter bar located between the Script Editor and the Explorer panel.

Set Script window preferences

1 Select Edit > Preferences > Script.

2 To choose the default font, click the Font button and select settings from the Font dialog box.

3 To choose the default color of text in the Script window, select a color from the Color menu.

4 To choose the background color for the Script window, select a color from the Background color menu.

5 To make new Script windows automatically color certain code elements, select Auto Coloring. This option is on by default. With Auto Coloring off, all text appears in the default color.

6 To make new Script windows automatically format your scripts with proper indenting, select Auto Format. This option is on by default.

Note: *The auto coloring and auto formatting features do not apply to JavaScript syntax code. Therefore, if you are authoring scripts using JavaScript syntax, the Auto Coloring and Auto Format buttons in the Script window are disabled, and terms such as* `function`*,* `var`*, and* `this` *will appear in the default text color.*

7 To make new Script windows display line numbers for your scripts, select Line Numbering. This option is on by default.

8 If Auto Coloring is on, select colors for the following code elements from the corresponding color menus:

- Keywords
- Comments
- Literals
- Custom (terms you define in your own code)

9 To change the line number column background color, click the Line Numbers color menu and choose a new color.

10 To change the location of the Call Stack, Variable, and Watcher panes in the Debugger window, select Left, Top, Right, or Bottom from the Debugger Panes menu.

11 Select Lingo or Javascript in the Default Script Type pop-up menu. Director uses the selected option as the default when opening the Explorer panel.

Note: *This is an application-level change that is retained after you close and re-open Director.*

12 To move the Explorer panel to a different location adjacent to the Script Editor in the Script window, select Left, Top, Right, or Bottom in the Explorer Panel pop-up menu. By default, the Explorer panel appears to the left of the Script Editor.

Note: *When debugging the script, a separate instance of the Script Editor opens along with the Debugger panel. This document window can be reordered in the application by shuffling it with the other open windows.*

Deleting scripts

You can delete scripts using the Explorer panel.

Delete a script

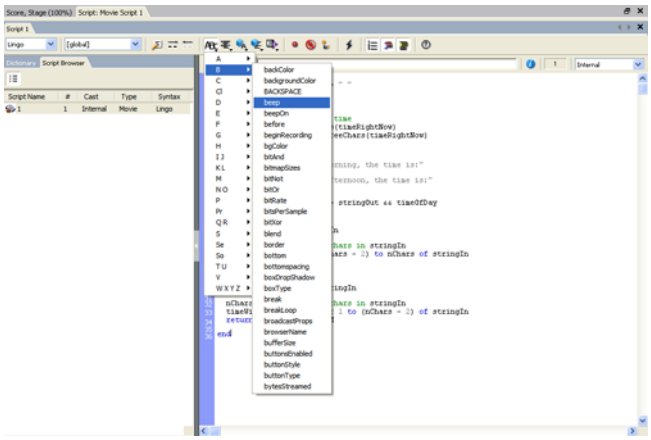
Do one of the following:

- In the Script Browser, right-click a script entry and select Delete.
- In the list view of the Script Explorer, select a script and press Delete.

Note: *When you delete a script that is open, the tab corresponding to the script is also removed from the Script window. If the open script you deleted was the only script in your movie, the Script window is disabled, and the tab in which the script appeared displays the text* Script. *To enable the text editor, click the plus sign (+) in the Script Editor panel.*

Inserting common scripting terms

The Script window provides pop-up menus of common scripting terms that you can use to insert statements in a script. The same menus also appear in the Message window.



In both the Script window and the Message window, you can select which scripting syntax you want to use for a particular script.

Select the scripting syntax

From the Script Syntax pop-up menu, select either Lingo or JavaScript.

After you select a scripting syntax, you enter code in the syntax you have chosen. If you try to compile a script in a syntax other than the one you chose, a script error occurs.

When entering scripts in the Script window, you can comment out or uncomment either single or multiple lines of code by using the Comment or Uncomment buttons. Depending on which scripting syntax you have chosen, the Comment and Uncomment buttons display the correct comment markers for that syntax; Lingo uses double hyphens (`--`), and JavaScript syntax uses double slashes (`//`).

Comment code

Highlight the line or lines of code you want to comment out, and click Comment.

Note: *When using the Comment button to comment out multiple lines of JavaScript syntax code, Director inserts double slashes before each line. You can also comment out multiple lines of code by inserting /* before the first line of commented code and inserting */ after the last line of commented code, but you must do this manually.*

Uncomment code

Highlight the line or lines of code that you want to remove comments from, and click Uncomment.

Both the Script window and the Message window contain the following menus:

- The Alphabetical Lingo menu lists every element in alphabetical order, except 3D Lingo.
- The Categorized Lingo menu lists categories of elements according to the features they are often used for. It does not include 3D Lingo.
- The Alphabetical 3D Lingo menu lists all 3D Lingo elements in alphabetical order.
- The Categorized 3D Lingo menu lists categories of all 3D Lingo elements according to the features they are used for.
- The Scripting Xtras pop-up menu includes the methods and properties of all scripting Xtra extensions found, regardless of whether they are Adobe or third-party Xtra extensions.

Note: *The scripting Xtra extensions listed in the Scripting Xtras pop-up menu are only those that support the* `Interface()` *method and whose names actually appear in the pop-up menu. Although some cast member media types such as 3D and DVD also support the* `Interface()` *method, they do not appear in the Scripting Xtras pop-up menu because they are not implemented in Director as scripting Xtra extensions.*

When you select an element from the pop-up menus, Director inserts the element at the insertion point in the Script window.

When an element requires additional parameters, placeholder names are included that indicate the additional required information. When more than one argument or parameter is required, the first one is highlighted for you, so all you must do is type to replace it. You must select and change the other parameters yourself.

Some cast member types and scripting Xtra extensions provide scripting terms that do not appear in the pop-up menus. These member types and Xtra extensions often have their own documentation, and you can find some information from within Director.

Display a list of available Xtra extensions

Issue either `put(_player.xtraList)` or `trace(_player.xtraList)` in the Message window.

Display a list of available scripting Xtra extensions

Issue either `put(_player.scriptingXtraList)` or `trace(_player.scriptingXtraList)` in the Message window.

Display a list of methods and properties for an Xtra extension

From the Scripting Xtras pop-up menu, point to an Xtra extension, and on the submenu, click Put Interface. The methods and properties for that Xtra extension appear in the Message window.

Entering and editing text

Entering and editing text in a Script window is similar to entering and editing text in any other field.

The following are common editing tasks that you perform in the Script window:

- To select a word, double-click the word.
- To select an entire script, choose Select All from the Edit menu.
- To start a new line, enter a carriage return.
- In Lingo, to wrap a long line of code with a continuation symbol, press Alt+Enter (Windows®) or Option+Return (Mac®) where you want to insert a soft line break. The continuation symbol (\) that appears indicates that the statement continues on the next line.

  In JavaScript syntax, to wrap a long line of code, insert a regular line break by pressing Enter (Windows) or Return (Mac). The Lingo continuation symbol causes a script error in JavaScript syntax scripts.

- To locate a handler in the current script, select the handler's name from the Go to Handler pop-up menu in the Script window.
- To compile any modified scripts, click the Script window's Recompile All Modified Scripts button or close the Script window. When you modify a Script, an asterisk appears in the Script window title bar, indicating that the script needs to be recompiled.
- To compile all scripts in a movie, select Recompile All Scripts from the Control menu.
- To reformat a script with proper indentation, press Tab in the Script window.

  Director automatically indents statements when the syntax is correct. If a line does not indent properly, there is a problem in the syntax on that line.

- To open a second Script window, Alt-click (Windows) or Option-click (Mac) the New Cast Member button in the Script window. This can be helpful for editing two different sections of a long script simultaneously.
- To toggle Line Numbering, click the Line Numbering button.
- To toggle Auto Coloring, click the Auto Coloring button. Auto Coloring displays each type of Lingo element (properties, commands, and so on) in a different color.
- To toggle Auto Formatting, click the Auto Format button. Auto Formatting adds the correct indenting to your scripts each time you add a carriage return or press Tab.

  Note: *The Auto Coloring and Auto Formatting features do not apply to JavaScript syntax code. Therefore, if you are authoring scripts using JavaScript syntax, the Auto Coloring and Auto Format buttons in the Script window are disabled, and terms such as* `function`*,* `var`*, and* `this` *will appear in the default text color.*

Finding handlers and text in scripts

The Find command in the Edit menu is useful for finding handlers and for finding and editing text and handlers.

Find handlers in scripts

1. Select Edit > Find > Handler.

   The Find Handler dialog box appears.

   The leftmost column in the Find Handler dialog box displays the name of each handler in the movie. The middle column displays the number of the cast member associated with the handler's script, along with the name of that cast member. The rightmost column lists the cast library that contains the cast member.

2. Select the handler that you want to find.
3. Click Find.

   The handler appears in the Script window.

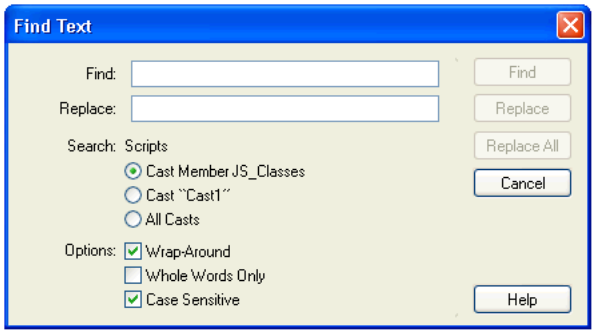
The title bar at the top of the Script window indicates the script's type.

Find text in scripts

1 Make the Script window active.

2 Select Edit > Find > Text.

The Find Text dialog box appears.



3 Enter text that you want to find in the Find field, and then click Find.

By default, find is not case-sensitive: `ThisHandler`, `thisHandler`, and `THISHANDLER` are all the same for search purposes. Click the Case Sensitive check box to make the find case-sensitive.

Specify which cast members to search

Select the appropriate option under Search: Scripts.

Start the search over from the beginning after the search reaches the end

Select the Wrap-Around option.

Search only for whole words and not fragments of other words that match the word

Select the Whole Words Only option.

Find the next occurrence of the text specified in the Find field

Select Edit > Find Again.

Find occurrences of selected text

1 Select the text.

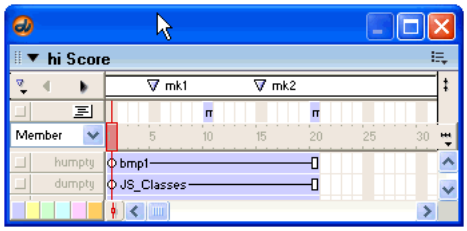
2 Select Edit > Find > Selection.

Performing common tasks

The following are ways to perform common tasks for creating, attaching, and opening scripts.

Create a frame behavior (script attached to a frame)

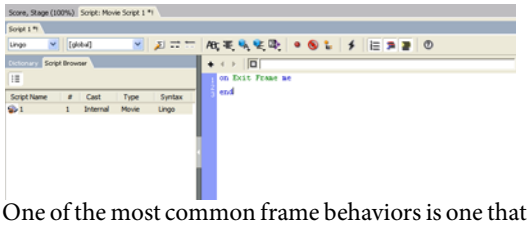
Double-click the behavior channel in the frame to which you want to attach the behavior.



A. *Behavior channels*

When you create a new behavior, the behavior receives the cast library number of the first available location in the current Cast window.

When you create a new frame behavior, the Script window opens and already contains the Lingo `on exitFrame` handler. The first line contains `on exitFrame`, followed by a line with a blinking insertion point, and then a line with the word `end`. This makes it easy for you to quickly attach a common Lingo behavior to the frame. To make this handler work with JavaScript syntax, replace `on exitFrame` with `function exitFrame() {`, and replace `end` with `}`.



One of the most common frame behaviors is one that keeps the playhead looping in a single frame. This can be useful when you want your movie to keep playing in a single frame while waiting for the user to click a button or for a digital video or sound to finish playing.

Keep the playhead in a single frame

In a frame behavior, type the following statement on the line after the `on exitFrame` (Lingo) or `function exitFrame()` (JavaScript syntax) statement:

```
-- Lingo syntax
_movie.go(_movie.frame)

// JavaScript syntax
_movie.go(_movie.frame);
```

The Movie object's `frame` property refers to the frame currently occupied by the playhead. This statement essentially tells the playhead to "go back to the frame you are currently in."

Create a sprite behavior (script attached to a sprite)

In the Score or on the Stage, select the sprite that you're attaching the behavior to. Then select Window > Behavior Inspector and select New Behavior from the Behavior pop-up menu.

When you create a new sprite behavior, the Script window opens and already contains the Lingo `on mouseUp` handler. The first line contains the line `on mouseUp`, followed by a line with a blinking insertion point, and then a line with the word `end`. This makes it easy for you to quickly attach a common behavior to the sprite. To make this handler work with JavaScript syntax, replace `on mouseUp` with `function mouseUp() {`, and replace `end` with `}`.

Open a behavior for editing

1 Double-click the behavior in the Cast window.

The Behavior inspector opens.

2 Click the Script Window icon in the Behavior inspector.

The Script window displays the behavior.

Alternatively, you can open the Script window and cycle through the scripts until you reach the behavior.

Remove a behavior from a Score location

❖ Select the location and then delete the script from the list displayed in the Property inspector (Behavior tab).

Attach existing behaviors to sprites or frames

❖ Do one of the following:

• Drag a behavior from a cast to a sprite or frame in the Score or (for sprites) to a sprite on the Stage.

• In the Score, select the sprites or frames that you're attaching the behavior to. Then select Window > Behavior Inspector and select the existing behavior from the Behavior pop-up menu.

Create a movie script (script attached to a movie)

Do one of the following:

• To create a movie script using Script Explorer in the tree view:

1 Select a movie node under the cast in which you want to add the movie script.

2 Right-click and select Add New Movie Script.

• If the current script in the Script window is a movie script, click the New Script button in the Script window. (Clicking the New Script button always creates a script of the same type as the current script.)

• If the current script in the Script window is not a movie script, click the New Script button and then change the new script's type with the Script Type pop-up menu in the Script tab of the Property inspector.

• If no sprites or scripts are selected in the cast, Score, or Stage, then open a new Script window; this creates a new movie script by default.

Open a movie script or parent script for editing

❖ After opening the script using the Script Explorer, double-click the script in the Cast window.

Change a script's type

1 Select the script in the Cast window or open it in the Script window.

2 Click the Script tab of the Property inspector and select a script type from the Script Type pop-up menu.

Cycle through the scripts in the Script window

❖ Use the Previous Cast Member and Next Cast Member arrows at the top of the Script window to advance or back up to a script.

Duplicate a script

❖ Select the script in the Cast window and select Duplicate from the Edit menu.

To create a script that is attached automatically to every sprite made from a specific cast member, attach the script to the cast member itself.

Create a script attached to a cast member or open an existing one

❖ Do one of the following:

• Right-click (Windows) or Control-click (Mac) a cast member in the Cast window and select Cast Member Script from the context menu.

• Select a cast member in the Cast window and then click the Cast Member Script button in the Cast window.

Using linked scripts

In addition to scripts stored as internal cast members, you can choose to keep scripts in external text files and link them to your Director movie. These linked scripts are similar to linked image or digital video files that you can import into Director movies.

Advantages of using linked scripts include the following:

• One person can work on the Director file while another works on the script.

• You can easily exchange scripts with others.

• You can control the scripts separately from the Director file in a source code control application such as Microsoft® Visual SourceSafe® or Perforce® by Perforce Software. Applications such as these prevent multiple programmers working on the same Director project from overwriting each other's work.

Linked scripts are used by Director only during authoring. At runtime, Director projectors and Adobe® Shockwave® Player use a special internal copy of the script data stored in the movie. This way, your linked scripts need not be distributed with your movies and cannot be copied by end users.

Import a script as a linked text file

1 Select File > Import.

2 Select Script as the type of file to import.

3 Select the script file(s) you want to import.

You can import files with the file extensions .txt, .ls, or .js. The .ls extension is the Director linked script extension.

To create a list of files you want to import, you can use the Add and Add All buttons. This is especially useful if you want to import scripts from multiple locations.

4 Select Link to External File from the Media pop-up menu.

5 Click Import.

You can edit linked scripts normally in the Director Script window. Changes you make are written to the external files each time you save your Director movie. (If you imported the linked script from a UNIX® server, UNIX line endings are preserved.) If you import a script whose text file is locked, you won't be able to edit the script in Director.

You cannot apply custom text colors to linked scripts in the Script window. Script Auto Coloring, however, is enabled for linked scripts.

Turn an internal script cast member into an external, linked script cast member

1 Select the internal cast member and click the Script tab of the Property inspector.

2 Click Link Script As.

3 Enter a name for the script file in the Save As dialog box.

4 Click Save.

Reload a linked script after it is edited

Use the Member object's `unload()` method.

If a linked script is edited outside of Director, you can reload it by using the `unload()` method in the Message window. The following statement causes the script member `myScript` to be unloaded and then reloaded:

```
-- Lingo syntax
member("myScript").unload()

// JavaScript syntax
member("myScript").unload();
```

Chapter 4: Debugging Scripts in Director

Scripts do not always do what you want the first time. The script often has an error in its syntax: possibly a word is misspelled or a small part of the script is missing. Other times, the script might work but does not produce the expected result. Mistakes or bugs almost always occur when you write scripts, so you should allow enough time for debugging when you develop multimedia titles.

As your skill with scripting increases, you'll probably encounter different types of problems as you master one area but start learning others. However, the basic troubleshooting techniques described here are useful for novice and advanced users alike.

The best way to correct a bug in your scripts varies from situation to situation. There are not one or two standard procedures that resolve the problem. You must use a variety of tools and techniques, such as the following:

- An overview and understanding of how scripts in the movie interact with each other
- Familiarity and practice with common debugging methods

The following tools help you identify problems in scripts:

- **The Message window**, when tracing is on, displays a record of the frames that play and the handlers that run in the movie.
- **The Debugger window** displays the values of global variables, properties of the script that is currently running, the sequence of handlers that ran to get to the current point, and the value of variables and expressions that you select.
- **The Script window** lets you enter comments, insert stopping points in the script, and select variables whose value is displayed in the Object inspector.
- **The Object inspector** lets you view and set the values of objects and properties you select.

Good scripting habits

Good scripting habits can help you avoid many scripting problems in the first place.

- Try to write your scripts in small sets of statements and test each one as you write it. This isolates potential problems where they are easier to identify.
- Insert comments that explain what the script statements are intended to do and what the values in the script are for. This makes it easier to understand the script if you return to it later or if someone else works on it. For example, the comment in the following statements make the purpose of the `if...then` structure and repeat loop clear:

```
-- Lingo syntax
-- Loop until the "s" key is pressed
repeat while not(_key.keyPressed("s"))
    _sound.beep()
end repeat

// JavaScript syntax
// Loop until the "s" key is pressed
while(!_key.keyPressed("s")) {
    _sound.beep();
}
```

- Make sure that the script's syntax is correct. Use the Script window's pop-up menus to insert preformatted versions of scripting elements. Rely on the API topics of this reference to check that statements are set up correctly.
- Use variable names that indicate the variables' purpose. For example, a variable that contains a number should be called something like `newNumber` instead of `ABC`.

Basic debugging

Debugging involves strategy and analysis, not a standard step-by-step procedure. This section describes the basic debugging approaches that programmers successfully use to debug any code, not just Lingo or JavaScript syntax.

Before you modify a movie significantly, always make a backup copy. It may help to name the copies incrementally, for example, fileName_01.dir, fileName_02.dir, fileName_03.dir, and so on to keep track of the various stages of a movie.

Identifying the problem

It might seem obvious, but the first thing to do when debugging is to identify the problem. Is a button doing the wrong thing? Is the movie going to the wrong frame? Is a field not editable when it should be?

You may also want to determine what you expect a particular script to do, and then compare your expectation with what the script actually does. This process helps you clearly define your goal and see what parts of the goal are not being met.

If you copied a script or a portion of a script from another movie or from a written example, check whether the script was designed for some specific conditions. Perhaps it requires that a sprite channel is already scripted. Maybe cast member names must follow a specific style convention.

Locating the problem

Do the following to start locating a problem:

- Think backwards through the chain to identify where the unexpected started to happen.
- Use the Message window to trace which frames the movie goes through and the handlers that your scripts run.
- Determine what the scripts should be doing and consider what in the statements relates to the problem. For example, if a text cast member is not editable when you expect it to be, where in the movie does (or does not) your script set the cast member's `editable` property?
- If a sprite does not change as intended on the Stage, is the `updateStage()` method needed somewhere?
- Does the problem occur only on certain computers and not others? Does it happen only when the display is set to millions of colors? Maybe something in the computer is interfering with the application.

You can focus on specific lines of script by inserting a breakpoint—a point where the script pauses its execution and invokes the Debugger window—in a line. This gives you a chance to analyze conditions at that point before the script proceeds. For information on how to insert breakpoints in a script, see "Debugging in the Debugger window" on page 86.

Solving simple problems

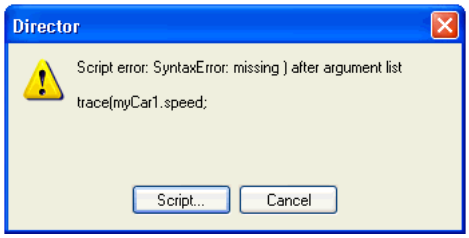
When finding a bug, it's a good idea to check for simple problems first.

The first debugging test occurs when you compile your script. You can compile your script by doing one of the following:

- In the Script window, click Recompile All Modified Scripts.
- From the Control menu, click Recompile All Scripts.
- Press Shift+F8.
- Close the Script window.

It is typically more convenient to compile scripts by using one of the first three options. The fourth option requires that you close the Script window each time you want to compile a script.

When you compile your script, Director® gives you an error message if the script contains incorrect syntax. The message usually includes the line in which the problem was first detected. A question mark appears at the spot in the line where Director first found the problem.



For example, the first line in the previous error message tells you that the script error in question is a syntax error and tells you what the syntax error is. The second line in the error message displays the actual line of code that contains the syntax error.

Looking for syntax errors

Syntax errors are probably the most common bug encountered while scripting. When a script fails, it is a good idea to first make sure that:

- Terms are spelled correctly, spaces are in the correct places, and necessary punctuation is used. Director cannot interpret incorrect syntax.
- Quotation marks surround the names of cast members, labels, and strings within a statement.
- All necessary parameters are present. The specific parameters depend on the individual element. See the API topic entries in this reference to determine any additional parameters that an element requires.

Looking for other simple bugs

If your script compiles without an error message, it might contain a bug. If your script is not doing what you want, check the following:

- Values for parameters are correct. For example, using an incorrect value for the number of beeps that you want the `beep()` method to generate gives you the wrong number of beeps.
- Values that change—such as variables and the content of text cast members—have the values you want. You can display their values in the Object inspector by selecting the name of the object and clicking Inspect Object in the Script window, or in the Message window by using the `put()` or `trace()` functions.
- The scripting elements do what you think they do. You can check their behavior by referring to the API topic entries in this reference.

• Capitalization is correct (ff the script is written in JavaScript syntax). JavaScript syntax is case-sensitive, which means that all methods, functions, properties, and variables must be referred to by using the correct capitalization.

If you attempt to call a method or function by using incorrect capitalization, you will receive a script error.

If you attempt to access a variable or property by using incorrect capitalization, you may not receive a script error, but your script may not behave as expected. For example, the following `mouseUp` handler contains a statement that attempts to access the `itemLabel` property by using incorrect capitalization. This script does not produce a script error, but instead dynamically creates a new variable with the incorrect capitalization. The value of the newly created variable is `undefined`.

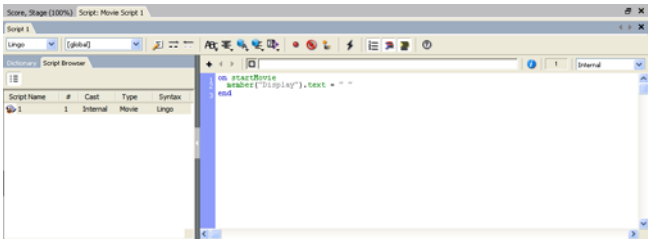
```
// JavaScript syntax
function beginSprite() {
    this.itemLabel = "Blue prints";
}

function mouseUp() {
    trace(this.itemlabel) // creates the itemlabel property
}
```

Debugging in the Script window

The Script window contains several features that can help you debug scripts.

Open the Script window

Select Window > Script.



Make the current line of code a comment

Click Comment.

Remove commenting from the current line of code

Click Uncomment.

Turn breakpoints in the current line of code on and off

Click Toggle Breakpoint.

Turn off all breakpoints

Click Ignore Breakpoints.

Add the selected expression or variable to the Object inspector

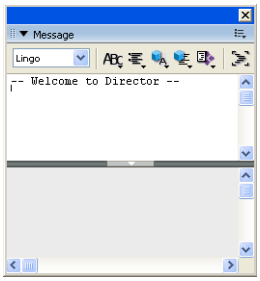
Click Inspect Object.

Debugging in the Message window

The Message window provides a way for you to test scripting commands and to monitor what is happening in your scripts while a movie plays.

Open the Message window

Select Window > Message.



Managing the Message window

The Message window has an Input pane and an Output pane. The Input pane is editable. The Output pane is read-only. The only way to display text in the Output pane is by calling the `put()` or `trace()` functions.

You can adjust the sizes of the Input and Output panes by dragging the horizontal divider that separates them.

Resize the Output pane

Drag the horizontal divider to a new position.

Hide the Output pane completely

Click the Collapse/Expand button in the center of the horizontal divider.

When the Output pane is hidden, output from scripts that execute is displayed in the Input pane.

Display the Output pane if it is hidden

Click the Collapse/Expand button again.

Clear the contents of the Message window

Click the Clear button.

If the Output pane is visible, its contents are deleted.

If the Output pane is not visible, the contents of the Input pane are deleted.

Delete a portion of the contents of the Output pane

1. Select the text to be deleted.
2. Press the Backspace or Delete key.

Copy text in the Input or Output pane

1. Select the text.
2. Select Edit > Copy.

Testing scripts in the Message window

You can test Lingo and JavaScript syntax statements to see how they work by entering them in the Message window and observing the results. When you enter a command in the Message window, Director executes the command immediately, regardless of whether a movie is playing.

Before entering the statements you want to test, you must first select which scripting syntax, Lingo or JavaScript syntax, you are going to test.

Select the scripting syntax

From the Script Syntax pop-up menu, select either Lingo or JavaScript.

Test a one-line statement

1 Type the statement directly in the Message window.

2 Press Enter (Windows®) or Return (Mac®). Director executes the statement.

If the statement is valid, the Message window displays the result of the statement in the Output pane at the bottom of the window. If the script is invalid, an alert appears.

For example, if you type the following statement into the Message window:

```
-- Lingo syntax
put(50+50)

// JavaScript syntax
trace(50+50);
```

```
and press Enter (Windows) or Return (Mac), the result appears in the Output pane:
```

```
-- Lingo syntax
-- 100

// JavaScript syntax
// 100
```

```
If you type the following statement into the Message window:
```

```
-- Lingo syntax
_movie.stage.bgColor = 255

// JavaScript syntax
_movie.stage.bgColor = 255;
```

and press Enter (Windows®) or Return (Mac®), the Stage becomes black.

You can test multiple lines of code all at once by copying and pasting statements into the Message window or by pressing Shift+Return after each line of code.

Execute multiple lines of code by copying and pasting

1 Copy the lines of code to the clipboard.

2 Enter a blank line in the Message window.

3 Paste the code into the Input pane of the Message window.

4 Place the insertion point at the end of the last line of code.

5 Press Control+Enter (Windows) or Control+Return (Mac). Director finds the first blank line above the insertion point and executes each line of code after the blank line in succession.

Enter multiple lines of code manually

1 Enter a blank line in the Message window.

2 Enter the first line of code.

3 Press Shift+Return at the end of the line.

4 Repeat steps 2 and 3 until you have entered the last line of code.

5 Press Control+Enter (Windows) or Control+Return (Mac). Director finds the first blank line above the insertion point and executes each line of code after the blank line in succession.

You can test a handler without running the movie by writing the handler in a Movie Script or Behavior Script window, and then calling it from the Message window.

Test a handler

1 Copy and paste or manually enter a multiline handler into the Message window as described in the previous two procedures.

2 Place the insertion point at the end of the last line of code.

3 Press Enter (Windows) or Return (Mac). The handler executes.

Any output from `put()` or `trace()` statements in the handler appears in the Message window.

Like the Script window, the Message window contains pop-up menus of scripting commands. When you select a command from one of the pop-up menus, the command appears in the Message window with the first argument that you must provide selected. Several menus are provided to give you easy access to the whole catalog of scripting terms.

The pop-up menus include the following:

- Alphabetical Lingo includes all commands except 3D Lingo, presented in an alphabetical list.
- Categorized Lingo includes all commands except 3D Lingo, presented in a categorized list.
- Alphabetical 3D Lingo includes all 3D Lingo, presented in an alphabetical list.
- Categorized 3D Lingo includes all 3D Lingo, presented in a categorized list.
- Scripting Xtras includes the methods and properties of all scripting Xtra extensions found, regardless of whether they are Adobe® or third-party Xtra extensions.

Note: *The scripting Xtra extensions listed in the Scripting Xtras pop-up menu are only those that support the* `Interface()` *method and whose names actually appear in the pop-up menu. Although some cast member media types such as 3D and DVD also support the* `Interface()` *method, they do not appear in the Scripting Xtras pop-up menu because they are not implemented in Director as scripting Xtra extensions.*

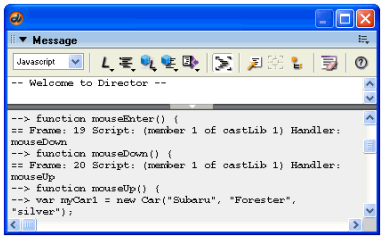
Monitoring scripts in the Message window

You can set the Output pane of the Message window to display a record of the statements that a movie executes as it plays. This is useful for tracking the flow of your code and seeing the result of specific statements. You can do this in one of two ways.

Display statements in the Output pane

Do one of the following:

- On the Message window, click Trace.
- Set the Movie object's `traceScript` property to `TRUE`.



Entries after a double equal sign (==) indicate what has occurred in the movie—such as which frame the movie has just entered, which script is running, or the result of a method or setting a value.

For example, the following line indicates several things:

```
== Frame: 39 Script: 1 Handler: mouseUp
```

- The movie entered frame 39.
- The movie ran script 1, the first script attached to the frame.
- The movie ran the `mouseUp` handler in script 1 after the movie entered the frame.

Entries after an arrow made up of a double hyphen and right angle bracket (`-->`) indicate lines of your code that have run. For example, the following Lingo lines:

```
--> _sound.fadeOut(1, 5*60)
--> if leftSide < 10 then
--> if leftSide < 200 then
--> _movie.go("Game Start")
```

indicate that these Lingo statements have run. Suppose you were trying to determine why the playhead did not go to the frame labeled "Game Start." If the line `--> _movie.go("Game Start")` never appeared in the Message window, maybe the condition in the previous statement was not what you expected.

The Message window Output pane can fill with large amounts of text when the Trace button is on. To delete the contents of the Output pane, click the Clear button. If the Output pane is not visible, the contents of the Input pane are deleted.

You can keep track of the value of variables and other objects by selecting the name of the object in the Message window and clicking the Inspect Object button. The object is added to the Object inspector, where its value is displayed and updated as the movie plays. For more information on the Object inspector, see "Debugging in the Object inspector" on page 83.

When you are in debugging mode, you can follow how a variable changes by selecting it in the Message window and then clicking the Watch Expression button. Director then adds the variable to the Watcher pane in the Debugger window, where its value is displayed and updated as you work in the Debugger window. For more information on the Watcher pane, see "Debugging in the Debugger window" on page 86.

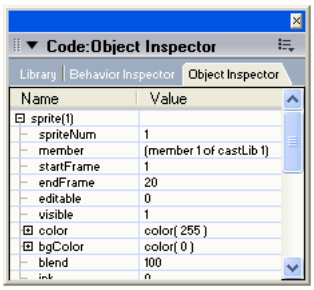
Debugging in the Object inspector

With the Object inspector, you can view and set properties of many kinds of objects that are not displayed in the Property inspector. These include Scripting objects such as global variables, lists, child objects from parent scripts, all 3D cast member properties, sprite properties, script expressions, and so on. In addition, the Object inspector displays changes to object properties that occur while a movie plays, such as changes due to scripts or changes to sprite Score properties. These kinds of runtime changes are not displayed in the Property inspector during movie playback.

To display values of JavaScript variables in the object inspector, you must declare them without the var preceding them.

Open the Object inspector

Select Window > Object Inspector.



Understanding object structure

The Object inspector can be very useful for understanding the structure of complex objects. For example, 3D cast members have many layers of properties. Because the Object inspector shows you a visual representation of the nested structure of those properties, it makes it much easier to become familiar with them and their relationships to each other. Understanding the property structure of objects in Director is important when writing scripts.

The ability to watch the values of properties change while a movie plays is helpful for understanding what is happening in the movie. It is especially helpful when testing and debugging scripts because you can watch as the values change based on scripts you've written.

The Director Debugger window displays this information also, but it is only available when you are in debugging mode. For more information on debugging, see "Advanced debugging" on page 91.

Viewable objects

The following are some of the objects you can enter into the Object inspector:

- Sprites, such as `sprite(3)`
- Cast members, such as `member("3d")`
- Global variables, such as `gMyList`
- Child objects, such as `gMyChild`
- Adobe® Flash® objects, such as `gMyFlashObject`; for more information about using Flash objects in Director, see the Using Director topics in the Director Help Panel.
- Script expressions, such as `sprite(7).blend`

Viewing objects

There are three ways to view an object in the Object inspector. You can drag items directly into the Object inspector, enter the name of an item into the Object inspector manually, or use the Inspect Object button in the Message and Script windows.

Drag an item to the Object inspector

Do one of the following:

- Select a sprite in the Score window and drag it to the Object inspector.
- Select a cast member in the Cast window and drag it to the Object inspector.
- Select the name of an object in the Script, Message, or Text window and drag it to the Object inspector.

Enter an object manually in the Object inspector

1 Double-click in the first empty cell in the Object column of the Object inspector.

2 Type the name of the object into the cell. Use the same name you would use to refer to the object in your scripts.

3 Press Enter (Windows) or Return (Mac). If the object has subproperties, a plus sign (+) appears to the left of it.

4 Click the plus sign. The properties of the object appear below it. Properties with subproperties appear with a plus sign to their left. Click each plus sign to display the subproperties.

View an object using the Inspect Object button

1 In the Script window, highlight the part of a statement that refers to an object.

2 In the Script window, click Inspect Object. If the object has subproperties, a plus sign (+) appears to the left of it.

3 Click the plus sign. The properties of the object appear below it. Properties with subproperties appear with a plus sign to their left. Click each plus sign to display the subproperties.

Note: *Inspecting large numbers of objects or large individual objects in the Object inspector can cause noticeable performance issues during authoring, particularly when Autopoll is enabled. For example, inspecting a list that contains 10,000 entries can make Director appear to hang while the display is updating.*

Navigating objects

You can also navigate the contents of the Object inspector with the arrow keys on your keyboard.

Move up or down in the list of items

Use the Up and Down arrow keys.

View an item's subproperties

Select the item and press the Right arrow key.

Hide an item's subproperties

Select the item and press the Left arrow key.

Using Autopoll

System properties, such as `milliseconds` and `colorDepth`, are updated in the Object inspector only when the Autopoll option is turned on. Using Autopoll increases the processor workload, which can slow your movie's performance when you add more than a few system properties to the Object inspector.

Turn on Autopoll

1 Right-click (Windows) or Control-click (Mac) in the Object inspector. The Object inspector context menu appears.

2 Select Autopoll from the context menu. When Autopoll is on, a check mark appears next to the Autopoll item in the context menu.

Turn off Autopoll

Select Autopoll from the context menu again.

Modifying object or property values

You can set the value of an object or property in the Object inspector by entering a new value in the box to the right of the object or property name.

Set an object or property value

1. Double-click the value to the right of the item name.
2. Enter the new value for the item.
3. Press Enter (Windows) or Return (Mac). The new value is set and reflected in your movie immediately.

You can enter a script expression as the value for an item. For example, you might set the value of `sprite(3).locH` to the expression `sprite(8).locH + 20`.

Removing objects

You can also remove items from the Object inspector.

Remove a single item from the Object inspector

Select the item and press the Backspace (Windows) or Delete (Mac) key.

Clear the entire contents of the Object inspector

Right-click (Windows) or Control-click (Mac) inside the Object inspector and select Clear All from the context menu.

When you open a separate movie from the one you are working on, the objects you entered in the Object inspector remain. This makes it easy to compare different versions of the same movie. When you exit Director, the items in the Object inspector are lost.

Debugging in the Debugger window

The Debugger window is a special mode of the Script window. It provides several tools for finding the causes of problems in your scripts. By using the Debugger window, you can quickly locate the parts of your code that are causing problems. The Debugger window allows you to run scripts one line at a time, skip over nested handlers, edit the text of scripts, and view the values of variables and other objects as they change. Learning to use the tools in the Debugger window can help you become a more efficient programmer.

The Debugger window can help you locate and correct errors in your scripts. It includes several tools that let you do the following:

- See the part of the script that includes the current line of code.
- Track the sequence of handlers that were called before getting to the current handler.
- Run selected parts of the current handler.
- Run selected parts of nested handlers called from the current handler.
- Display the value of any local variable, global variable, or property related to the code that you're investigating.

Entering debugging mode

In order to access the Debugger window, a break must occur in a script. A break occurs when Director encounters a script error or a breakpoint in a script.

When a script error occurs, the Script Error dialog box appears. The dialog box displays information about the type of error that occurred and asks you whether you want to debug the script, edit the script in the Script window, or cancel.

Enter debugging mode

Do one of the following:

- Click Debug in the Script Error dialog box.
- Place a breakpoint in a script.

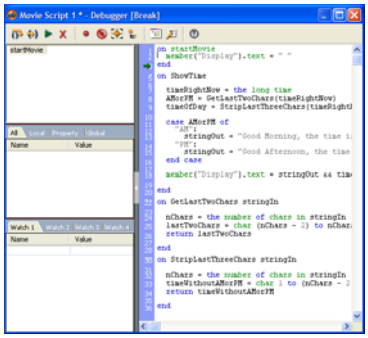
When Director runs and encounters a breakpoint, the script stops executing and the Script window changes to debugging mode. The movie is still playing, but the execution of your scripts is stopped until you use the Debugger window to tell Director how to proceed. If you have multiple Script windows open, Director searches for one containing the script where the breakpoint occurred and changes that window to debugging mode.

Add a breakpoint that will open the Debugger window

1 In the Script window, open the script that should contain the breakpoint.

2 Click in the left margin of the Script window next to the line of code where you want the breakpoint to appear, or place the insertion point on the line of code and click Toggle Breakpoint. Your code will stop executing at the beginning of this line, and the Script window will enter debugging mode.

If the Script window is open when Director encounters a script error or a breakpoint, the Debugger window replaces the Script window.



Stop debugging

Do one of the following:

- Click the Run Script button in the Debugger window. This resumes normal script execution.
- Click the Stop Debugging button in the Debugger window. This stops both the debugging session and the movie.

  The Script window reappears in place of the Debugger window.

When the Debugger window opens, it shows the current line of code and offers several choices for what to run next.

See which is the current line of code

Look for the green arrow next to a line of code in the Script pane.

The green arrow points to the current line. You cannot select a different line of code by clicking it in the Script pane.

Viewing the call stack in the Debugger window

The Call Stack pane displays the sequence of nested handlers that ran before the current line of code. This sequence is called the call stack. Use the call stack to keep track of the structure of your code while you are debugging. You can view the variables associated with a specific handler by clicking the handler name in the Call Stack pane. The variables are displayed in the Variable pane.

Viewing variables in the Debugger window

The Variable pane of the Debugger window displays the variables associated with the current handler. The current handler is the handler displayed in the Script pane and the last handler displayed in the Call Stack pane. You can also display the variables associated with previous handlers in the call stack. As you step through a script, changes to the values of any of the variables are displayed in red. For more information on stepping through scripts, see "Stepping through scripts in the Debugger window" on page 89.

Display the variables associated with a handler in the call stack

Click the name of the handler in the Call Stack pane. The variables appear in the Variable pane.

The Variable pane includes four tabs for viewing variables:

The All tab displays both global and local variables associated with the current handler.

The Local tab displays only the local variables associated with the selected handler.

The Property tab displays the properties declared by the current script.

The Global tab displays only the global variables associated with the selected handler.

Sort the variables in the Variable pane

- To sort the variables by name, click the word *Name* that appears above the variable names.
- To sort the variables in reverse-alphabetical order, click the word *Name* a second time.

You can change the values of local variables of the current handler and global variables in the Variable pane. You cannot change the value of local variables that are not in the current handler.

Change the value of a variable in the Variable pane

1. Double-click the value of the variable in the Value column.
2. Enter the new value for the variable.
3. Press Enter (Windows) or Return (Mac).

Viewing objects in the Debugger window

With the Watcher pane in the Debugger window, you can view variables and other data objects associated with the current handler, as well as objects associated with other handlers. By adding objects to the Watcher pane, you can keep track of their values as they change because of scripts. When the value of an object changes due to the execution of a line of code, Director changes the color of the object's name in the Watcher pane to red.

The Watcher pane displays only the objects you add to it. You can use up to four separate tabs in the Watcher pane to organize objects into groups.

Add an object to the Watcher pane whose name appears in the Script pane

1. Select the name of the object in the Script pane.
2. Click the Watch Expression button.

Add an object to the Watcher pane whose name does not appear in the Script pane

1. Double-click the first empty cell in the object column of the Watcher pane.
2. Type the name of the object in the cell and press Enter (Windows) or Return (Mac).

   If the object has properties, a plus sign (+) appears next to the object's name.

Display an object's properties

Click the plus sign next to the object name.

The Watcher pane lets you organize objects in a few ways.

Organize objects in the Watcher pane

Do one of the following:

- To sort the objects in the Watcher pane, click the Name column head at the top of the left column. The object names in the column are listed in alphabetical order.
- To sort the objects in reverse-alphabetical order, click the Name column head a second time.
- To organize objects into groups, use the tabs in the Watcher pane. To add an object to a specific tab, click the tab you want to use before adding the object.
- To clear the contents of a tab in the Watcher pane, select the tab and then right-click (Windows) or Control-click (Mac) in the Watcher pane and select Clear All.

Stepping through scripts in the Debugger window

The Debugger window provides you with a set of tools for running scripts slowly, so you can watch the effect that each line of code has on your movie. You can execute one line of code at a time and choose whether to execute nested handlers one line at a time or all at once.

Execute only the current line of code indicated by the green arrow

Click the Step Script button.

Many handlers include calling statements to other handlers. You can focus your attention on such nested handlers, or ignore them and focus on the code in the current handler.

When you are confident that nested handlers are performing as expected and want to concentrate on the code in the current handler, the Debugger window can step over nested handlers and go directly to the next line of code in the current handler. When the Debugger steps over a nested handler, it executes the handler but does not display the handler's code or pause within the nested handler.

Step over nested handlers

Click the Step Script button in the Debugger window.

The Step Script button runs the current line of code, runs any nested handlers that the line calls, and then stops at the next line in the handler.

If you suspect that nested handlers are not performing as expected and want to study their behavior, the Debugger window can run nested handlers line by line as well.

Run nested handlers one line at a time

Click the Step Into Script button in the Debugger window.

Clicking the Step Into button runs the current line of code and follows the normal flow through any nested handlers called by that line. When finished with a nested handler, the Debugger window stops at the next line of code within the upper-level handler.

When you are finished debugging, you can exit the Debugger at any time.

Resume normal execution of code and exit the Debugger window

Click the Run Script button.

Exit the Debugger and stop playback of the movie

Click the Stop Debugging button.

Editing scripts in debugging mode

When you are in debugging mode, you may edit your scripts directly in the Debugger window. This enables you to fix bugs as soon as you find them and then continue debugging.

Edit a script in the Debugger window

1 Click in the Script pane and place the insertion point where you want to begin typing.

2 Enter the changes to the script.

You can jump to a specific handler by selecting the name of the handler and clicking the Go to Handler button.

3 When you are finished debugging and editing scripts, click the Stop Debugging button. The Script window returns to Script mode.

4 Click the Recompile All Modified Scripts button.

Debugging projectors and Shockwave movies

This section discusses debugging during runtime in Director projectors and movies that contain Adobe® Shockwave® content. You can use either the Message window or enable full script error dialog boxes to debug projectors and movies that contain Shockwave content.

Debug using the Message window

Set the Player object's `debugPlaybackEnabled` property to `TRUE`.

When this property is `TRUE`, playing back a projector or a movie that contains Shockwave content opens a Message window (Windows) or a Message text file (Mac), and the results of any `put()` or `trace()` function calls are output to these formats.

If at any time during the movie the `debugPlaybackEnabled` property is set to `FALSE`, the Message window or text file is closed and cannot be opened again during that playback session, even if `debugPlaybackEnabled` is set back to `TRUE` later in that session.

Debug by enabling full script error dialogs

In an .ini file for a projector or a movie that contains Shockwave content, set the `DisplayFullLingoErrorText` property to 1.

This generates more descriptive error text in the script error dialog box than the generic error text. For example, a generic error message might look like:

```
Script error: Continue?
```

Setting the `DisplayFullLingoErrorText` property to 1 could generate the following error message:

```
Script error: list expected
```

For information on creating and modifying an appropriate .ini file for a projector or a movie that contains Shockwave content, see the Director.ini template file in the root Director installation folder.

Advanced debugging

If the problem is not easy to identify, try the following approaches:

- Determine which section has the problem. For example, if clicking a button produces the wrong result, investigate the script assigned to the button.

  If a sprite does the wrong thing, try checking the sprite's property values. Are they set to the values you want when you want?

- Figure out where the script flows. When a section of the movie does not do what you want, first try to trace the movie's sequence of events in your head. Look at other scripts in the message hierarchy to make sure Director is running the correct handler.

- Follow the tracing in the Message window; this shows which frames the movie goes through and any handlers that the movie calls as the movie plays.

- Try using the Step Script and Step Into features in the Debugger window and see whether the results differ from what you expect.

- Check variables and expressions. Analyze how their values change as the movie plays. See if they change at the wrong time or do not change at all. If the same variable is used in more than one handler, make sure that each handler that uses the variable states that the variable is global.

  You can track the values of variables and expressions by displaying their values in the Watcher pane of the Debugger window or the Object inspector.

- Make changes one at a time. Don't be afraid to change things in a handler to see if the change eliminates the problem or gives some result that helps point to the problem.

  However, do not trade one problem for another. Change things one at a time and change them back if the problem is not fixed. If you introduce too many changes before solving a problem, you might not determine what the original problem was and you might even introduce new problems.

- Re-create the section. If you have not found the problem, try re-creating the section from scratch. For example, if rolling the pointer over a sprite does not make the sprite behave the way you want, create a simple movie that contains just the sprite and handler with the `rollOver()` method.

  Do not just copy and paste scripts; that might just copy the problem. Re-creating the section lets you reconstruct the logic at its most basic level and verify that Director is working as you expect. If the section that you re-create still does not work properly, chances are that there is something wrong in the logic for the section.

If the section that you re-create works properly, compare that section to the original movie to see where the two differ. You can also copy the section into the original piece and see whether this corrects the problem.

Chapter 5: Director Core Objects

The core objects in Director® provide access to the core functionality and features available in Director, projectors, and the Adobe® Shockwave® Player. Core objects include the Director player engine, movie windows, sprites, sounds, and so on. They represent the base layer through which almost all APIs and other object categories are accessed; the exceptions are the scripting objects, which extend the core functionality of Director.

For an illustration of how the core objects relate to each other and to other objects in Director, see "Object model diagrams" on page 44.

Cast Library

Represents a single cast library within a movie.

A movie can consist of one or more cast libraries. A cast library can consist of one or more cast members, which represent media in a movie, such as sounds, text, graphics, and other movies.

You can create a reference to a cast library by using either the top level `castLib()` function or the Movie object's `castLib` property. For example, if a movie contains a cast library named `scripts`, you could create a reference to this cast library by doing the following:

- Use the top level castLib() method.

```
-- Lingo syntax
libScript = castLib("scripts")

// JavaScript syntax
var libScript = castLib("scripts");
```

- Use the Movie object's castLib property.

```
-- Lingo syntax
libScript = _movie.castLib["scripts"]

// JavaScript syntax
var libScript = _movie.castLib["scripts"];
```

Method summary for the Cast Library object

Method
findEmpty()

Property summary for the Cast Library object

Property
fileName (Cast)
member (Cast)
name

Property (Continued)
number (Cast)
preLoadMode
selection

See also

castLib, castLib(), Member, Movie, Player, Sprite, Window

Global

Provides a location to store and access global variables. These variables are available to both Lingo and JavaScript syntax.

You can access the Global object by using the top level _global property. You can either assign _global to a variable, or use the _global property directly to access the Global object's methods and any defined global variables.

- Assign _global to a variable.

```
-- Lingo syntax
objGlobal = _global

// JavaScript syntax
var objGlobal = _global;
```

- Use the _global property directly.

```
-- Lingo syntax
_global.showGlobals()

// JavaScript syntax
_global.showGlobals();
```

- Access a global variable.

```
-- Lingo syntax
global gSuccess

on mouseDown
    gSuccess = "Congratulations!"
    put(gSuccess) -- displays "Congratulations!"
end

// JavaScript syntax
_global.gSuccess = "Congratulations!";

function mouseDown() {
    trace(_global.gSuccess); // displays "Congratulations!"
}
```

Method summary for the Global object

Method
clearGlobals()
showGlobals()

See also

_global

Key

Used to monitor a user's keyboard activity.

You can access the Key object by using the top level _key property. You can either assign _key to a variable, or use the _key property directly to access the Key object's methods and properties.

- Assign _key to a variable.

```
-- Lingo syntax
objKey = _key

// JavaScript syntax
var objKey = _key;
```

- Use the _key property directly.

```
-- Lingo syntax
isCtrlDown = _key.controlDown

// JavaScript syntax
var isCtrlDown = _key.controlDown;
```

Method summary for the Key object

Method
keyPressed()

Property summary for the Key object

Property
commandDown
controlDown
key
keyCode
optionDown
shiftDown

See also

_key

Member

Represents a cast member within a cast library. Cast members are the media and script assets in a movie. Media cast members may be text, bitmaps, shapes, and so on. Script cast members include behaviors, movie scripts, and so on.

A cast member can be referenced either by number or by name.

- When referring to a cast member by number, Director searches a particular cast library for that cast member, and retrieves the member's data. This method is faster than referring to a cast member by name. However, because Director does not automatically update references to cast member numbers in script, a numbered reference to a cast member that has moved position in its cast library will be broken.
- When referring to a cast member by name, Director searches all cast libraries in a movie from first to last, and retrieves the member's data when it finds the named member. This method is slower than referring to a cast member by number, especially when referring to large movies that contain many cast libraries and cast members. However, a named reference to a cast member allows the reference to remain intact even if the cast member moves position in its cast library.

You can create a reference to a cast member by using either the top level `member()` function, or by using the `member` property of the Cast, Movie, or Sprite object.

The following examples illustrate creating a reference to a cast member.

- Use the top level `member()` function.

```
-- Lingo syntax
objTree = member("bmpTree")

// JavaScript syntax
var objTree = member("bmpTree");
```

- Use the Sprite object's `member` property.

```
-- Lingo syntax
objTree = sprite(1).member;

// JavaScript syntax
var objTree = sprite(1).member;
```

Method summary for the Member object

Method
copyToClipBoard()
duplicate() (Member)
erase()
importFileInto()
move()
pasteClipBoardInto()
preLoad() (Member)
unLoad() (Member)

Property summary for the Member object

Property	
castLibNum	modifiedDate
comments	name
creationDate	number (Member)
fileName (Member)	purgePriority
height	rect (Member)
hilite	regPoint
linked	scriptText
loaded	size
media	thumbNail
mediaReady	type (Member)
modified	width
modifiedBy	

See also

Media Types, member(), member (Cast), member (Movie), member (Sprite), Movie, Player, Scripting Objects, Sprite, Window

Mouse

Provides access to a user's mouse activity, including mouse movement and mouse clicks.

You can access the Mouse object by using the top level _mouse property. You can either assign _mouse to a variable, or use the _mouse property directly to access the Mouse object's properties.

- Assign _mouse to a variable.

```
-- Lingo syntax
objMouse = _mouse

// JavaScript syntax
var objMouse = _mouse;
```

- Use the _mouse property directly.

```
-- Lingo syntax
isDblClick = _mouse.doubleClick

// JavaScript syntax
var isDblClick = _mouse.doubleClick;
```

Property summary for the Mouse object

Property	
clickLoc	mouseLoc
clickOn	mouseMember
doubleClick	mouseUp
mouseChar	mouseV
mouseDown	mouseWord
mouseH	rightMouseDown
mouseItem	rightMouseUp
mouseLine	stillDown

See also

_mouse

Movie

Represents a movie being played within the Director player.

The Director player can contain one or more movies. A movie can consist of one or more cast libraries. A cast library can consist of one or more cast members, which represent the media and script assets in a movie. Media cast members may be text, bitmaps, shapes, and so on. Script cast members include behaviors, movie scripts, and so on. Sprites are created from cast members and are used on the Stage of a movie.

You can refer to the currently active movie by using the top level `_movie` property. You can refer to any movie in the player by using the Window object's `movie` property.

- Refer to the currently active movie.

```
-- Lingo syntax
objMovie = _movie

// JavaScript syntax
var objMovie = _movie;
```

- Use the Window object's movie property to access the movie in a particular window.

```
-- Lingo syntax
objMovie = _player.window[2].movie

// JavaScript syntax
var objMovie = _player.window[2].movie;
```

You can use a movie reference to access not only the methods and properties of a movie itself, you can also call Lingo and JavaScript syntax handlers, and access the movie's cast members and sprites, including their methods and properties. This differs from previous versions of Director in which you had to use the `tell` command to work with movies. The Movie object provides a simpler way to work with movies.

Method summary for the Movie object

Method	
beginRecording()	newMember()
cancelIdleLoad()	preLoad() (Movie)
clearFrame()	preLoadMember()
constrainH()	preLoadMovie()
constrainV()	printFrom()
delay()	puppetPalette()
deleteFrame()	puppetSprite()
duplicateFrame()	puppetTempo()
endRecording()	puppetTransition()
finishIdleLoad()	ramNeeded()
frameReady() (Movie)	rollOver()
go()	saveMovie()
goLoop()	sendAllSprites()
goNext()	sendSprite()
goPrevious()	stopEvent()
idleLoadDone()	unLoad() (Movie)
insertFrame()	unLoadMember()
label()	unLoadMovie()
marker()	updateFrame()
mergeDisplayTemplate()	updateStage()

Property summary for the Movie object

Property	
aboutInfo	frameTransition
active3dRenderer	idleHandlerPeriod
actorList	idleLoadMode
allowCustomCaching	idleLoadPeriod
allowGraphicMenu	idleLoadTag
allowSaveLocal	idleReadChunkSize
allowTransportControl	imageCompression
allowVolumeControl	imageQuality
allowZooming	keyboardFocusSprite

Property (Continued)	
beepOn	lastChannel
buttonStyle	lastFrame
castLib	markerList
centerStage	member (Movie)
copyrightInfo (Movie)	name
displayTemplate	paletteMapping
dockingEnabled	path (Movie)
editShortCutsEnabled	preferred3dRenderer
enableFlashLingo	preLoadEventAbort
exitLock	score
fileFreeSize	scoreSelection
fileSize	script
fileVersion	sprite (Movie)
fixStageSize	stage
frame	timeoutList
frameLabel	traceLoad
framePalette	traceLogFile
frameScript	traceScript
frameSound1	updateLock
frameSound2	useFastQuads
frameTempo	xtraList (Movie)

See also

`_movie`, `Cast Library`, `Member`, `movie`, `Player`, `Sprite`, `Window`

Player

Represents the core playback engine used to manage and execute the authoring environment, movies in a window (MIAWs), projectors, and Shockwave Player.

The Player object provides access to all of the movies and windows that it is managing, in addition to any Xtra extensions that are available.

You can create a reference to the Player object by using the top level `_player` property.

- Assign `_player` to a variable.

```
-- Lingo syntax
objPlayer = _player
```

```
// JavaScript syntax
var objPlayer = _player;
```

- Use the _player property directly.

```
-- Lingo syntax
_player.alert("The movie has ended.")

// JavaScript syntax
_player.alert("The movie has ended.");
```

Method summary for the Player object

Method	
alert()	getPref()
appMinimize()	halt()
cursor()	open() (Player)
externalParamName()	quit()
externalParamValue()	setPref()
flushInputEvents()	windowPresent()

Property summary for the Player object

Property	
activeCastLib	netPresent
activeWindow	netThrottleTicks
alertHook	organizationName
applicationName	productName
applicationPath	productVersion
currentSpriteNum	safePlayer
debugPlaybackEnabled	scriptingXtraList
digitalVideoTimeScale	searchCurrentFolder
disableImagingTransformation	searchPathList
emulateMultibuttonMouse	serialNumber
externalParamCount	sound (Player)
frontWindow	switchColorDepth
inlineImeEnabled	toolXtraList
itemDelimiter	transitionXtraList
lastClick	userName
lastEvent	window

Property (Continued)	
lastKey	xtra
lastRoll	xtraList (Player)
mediaXtraList	

See also

_player, Cast Library, Member, Movie, Sprite, Window

Sound

Controls audio playback in all eight available sound channels.

The Sound object consists of Sound Channel objects, which represent individual sound channels.

You can create a reference to the Sound object by using top level _sound property.

- Assign _sound to a variable.

```
-- Lingo syntax
objSound = _sound

// JavaScript syntax
var objSound = _sound;
```

- Use the _sound property to access the Sound object's soundDevice property.

```
-- Lingo syntax
objDevice = _sound.soundDevice

// JavaScript syntax
var objDevice = _sound.soundDevice;
```

Method summary for the Sound object

Method
beep()
channel() (Sound)

Property summary for the Sound object

Property
soundDevice
soundDeviceList
soundEnabled
soundKeepDevice
soundLevel
soundMixMedia

See also

_sound, Sound Channel

Sound Channel

Represents an individual sound channel found within the Sound object.

There are eight available sound channels. You can use a Sound Channel object in script to access and modify any of the eight sound channels.

Note: *You can modify only the first two sound channels in the Score of the Director user interface.*

You can create a reference to a Sound Channel object by using the top level `sound()` method, the Player object's `sound` property, or the Sound object's `channel()` method. For example, you can reference sound channel 2 by doing the following:

- Use the top level `sound()` method.

```
-- Lingo syntax
objSoundChannel = sound(2)

// JavaScript syntax
var objSoundChannel = sound(2);
```

- Use the Player object's `sound` property.

```
-- Lingo syntax
objSoundChannel = _player.sound[2]

// JavaScript syntax
var objSoundChannel = _player.sound[2];
```

- Use the Sound object's `channel()` method.

```
-- Lingo syntax
objSoundChannel = _sound.channel(2)

// JavaScript syntax
var objSoundChannel = _sound.channel(2);
```

Method summary for the Sound Channel object

Method	
breakLoop()	play() (Sound Channel)
fadeIn()	playFile()
fadeOut()	playNext() (Sound Channel)
fadeTo()	queue()
getPlayList()	rewind() (Sound Channel)
isBusy()	setPlayList()
pause() (Sound Channel)	stop() (Sound Channel)

Property summary for the Sound Channel object

Property	
channelCount	member (Sound Channel)
elapsedTime	pan
endTime	sampleCount
loopCount	sampleRate
loopEndTime	startTime
loopsRemaining	status
loopStartTime	volume (Sound Channel)

See also

`channel() (Sound)`, `sound (Player)`, `sound()`, `Sound`

Sprite

Represents an occurrence of a cast member in a sprite channel of the Score.

A Sprite object covers a sprite span, which is a range of frames in a given sprite channel. A Sprite Channel object represents an entire sprite channel, regardless of the number of sprites it contains.

Note: *Flash® member components placed on the stage (Flash sprites) that are invisible can be accessed only by using the member object. Using a sprite object to access a Flash sprite with an invisible property will result in an error.*

A sprite can be referenced either by number or by name.

- When referring to a sprite by number, Director searches all sprites that exist in the current frame of the Score, starting from the lowest numbered channel, and retrieves the sprite's data when it finds the numbered sprite. This method is faster than referring to a sprite by name. However, because Director does not automatically update references to sprite numbers in script, a numbered reference to a sprite that has moved position on the Stage will be broken.

- When referring to a sprite by name, Director searches all sprites that exist in the current frame of the Score, starting from the lowest numbered channel, and retrieves the sprite's data when it finds the named sprite. This method is slower than referring to a sprite by number, especially when referring to large movies that contain many cast libraries, cast members, and sprites. However, a named reference to a sprite allows the reference to remain intact even if the sprite moves position on the Stage.

You can create a reference to a Sprite object by using the top level `sprite()` function, the Movie object's `sprite` property, or the Sprite Channel object's `sprite` property.

- Use the top level `sprite()` function.

```
-- Lingo syntax
objSprite = sprite(1)

// JavaScript syntax
var objSprite = sprite(1);
```

- Use the Movie object's `sprite` property.

```
-- Lingo syntax
objSprite = _movie.sprite["willowTree"]

// JavaScript syntax
var objSprite = _movie.sprite["willowTree"];
```

- Use the Sprite Channel object's `sprite` property.

```
-- Lingo syntax
objSprite = channel(3).sprite

// JavaScript syntax
var objSprite = channel(3).sprite;
```

You can use a reference to a Sprite object to access the cast member from which the sprite was created. Any changes made to the cast member from which a sprite was created are also reflected in the sprite. The following example illustrates changing the text of a text cast member from which sprite 5 was created. This change to the cast member will also be reflected in sprite 5.

```
-- Lingo syntax
labelText = sprite(5)
labelText.member.text = "Weeping Willow"

// JavaScript syntax
var labelText = sprite(5);
labelText.member.text = "Weeping Willow";
```

Property summary for the Sprite object

Property	
backColor	locV
blend (Sprite)	locZ
bottom	member (Sprite)
constraint	name (Sprite)
cursor	quad
editable	rect (Sprite)
endFrame	right
filterlist	rotation
flipH	skew
flipV	spriteNum
foreColor	startFrame
height	top
ink	width
left	
locH	

See also

Cast Library, Member, Movie, Player, sprite (Movie), sprite (Sprite Channel), sprite(), Sprite Channel, Window

Sprite Channel

Represents an individual sprite channel in the Score.

A Sprite object covers a sprite span, which is a range of frames in a given sprite channel. A Sprite Channel object represents an entire sprite channel, regardless of the number of sprites it contains.

Sprite channels are controlled by the Score by default. Use the Sprite Channel object to switch control of a sprite channel over to script during a Score recording session.

A sprite channel can be referenced either by number or by name.

- When referring to a sprite channel by number, you access the channel directly. This method is faster than referring to a sprite channel by name. However, because Director does not automatically update references to sprite channel numbers in script, a numbered reference to a sprite channel that has moved position in the Score will be broken.
- When referring to a sprite channel by name, Director searches all channels, starting from the lowest numbered channel, and retrieves the sprite channel's data when it finds the named sprite channel. This method is slower than referring to a sprite channel by number, especially when referring to large movies that contain many cast libraries, cast members, and sprites. However, a named reference to a sprite channel allows the reference to remain intact even if the sprite channel moves position in the Score.

You can create a reference to a Sprite Channel object by using the top level `channel()` method, and referring to either the channel number or name.

```
-- Lingo syntax
objSpriteChannel = channel(2) -- numbered reference
objSpriteChannel = channel("background") -- named reference

// JavaScript syntax
var objSpriteChannel = channel(2); // numbered reference
var objSpriteChannel = channel("background"); // named reference
```

You can use a reference to a Sprite Channel object to access the sprite that is currently being used in a particular sprite channel. The following example illustrates accessing the background color of the sprite that is currently being used in sprite channel 2.

```
-- Lingo syntax
labelSprite = channel(2).sprite.backColor

// JavaScript syntax
var labelSprite = channel(2).sprite.backColor;
```

Method summary for the Sprite Channel object

Method
makeScriptedSprite()
removeScriptedSprite()

Property summary for the Sprite Channel object

Property
name (Sprite Channel)
number (Sprite Channel)
scripted
sprite (Sprite Channel)

See also

Cast Library, channel() (Top level), Member, Movie, Player, Sprite, Window

System

Provides access to system and environment information, including system level methods.

You can create a reference to the System object by using the top level _system property.

- Assign _system to a variable.

```
-- Lingo syntax
objSystem = _system

// JavaScript syntax
var objSystem = _system;
```

- Use the _system property directly.

```
-- Lingo syntax
sysDate = _system.date()

// JavaScript syntax
var sysDate = _system.date();
```

Method summary for the System object

Method
date() (System)
restart()
shutDown()
time() (System)

Property summary for the System object

Property
colorDepth

Property (Continued)
deskTopRectList
environmentPropList
milliseconds

See also

_system

Window

Represents a window in which a movie is playing, including the Stage window and any other movies in a window (MIAWs) that are in use.

You can create a reference to a Window object by using the top level `window()` function, the Player object's `window` property, or the Player object's `windowList` property.

- Use the top level `window()` method.

```
-- Lingo syntax
objWindow = window("Sun")

// JavaScript syntax
var objWindow = window("Sun");
```

- Use the Player object's `window` property.

```
-- Lingo syntax
objWindow = _player.window["Sun"]

// JavaScript syntax
var objWindow = _player.window["Sun"];
```

- Use the Player object's `windowList` property.

```
-- Lingo syntax
objWindow = _player.windowList[1]

// JavaScript syntax
var objWindow = _player.windowList[1];
```

Note: *When creating a named reference to a window by using either the top level `window()` function or the Player object's `window` property, a reference to that window is created only if a window by that name exists. If a window by that name does not exist, the reference contains `VOID` (Lingo) or `null` (JavaScript syntax).*

The movie object property `scriptExecutionStyle` is set to a value of 10 by default, and `windowType` is deprecated by default in favor of the `appearanceOptions` and `titlebarOptions` property lists. If `scriptExecutionStyle` is set to a value of 9, `windowType` is fully functional.

Method summary for the Window object

Method	
close()	moveToBack()
forget() (Window)	moveToFront()
maximize()	open() (Window)
mergeProps()	restore()
minimize()	

Property summary for the Window object

Property	
appearanceOptions	resizable
bgColor (Window)	sizeState
dockingEnabled	sourceRect
drawRect	title (Window)
fileName (Window)	titlebarOptions
image (Window)	type (Window)
movie	visible
name	windowBehind
picture (Window)	windowInFront
rect (Window)	

See also

`Cast Library`, `Member`, `Movie`, `Player`, `Sprite`, `window()`, `window`, `windowList`

Chapter 6: Media Types

The media types in Director® provide access to the functionality of the various media types, such as RealMedia®, DVD, Animated GIF, and so on, that are added to movies as cast members.

Literally, media types are not actually objects, but rather cast members that are of a specific type of media. When a media type is added to a movie as a cast member, it not only inherits the functionality of the core Member object, it also extends the Member object by providing additional functionality that is available only to the specified media type. For example, a RealMedia cast member has access to the Member object's methods and properties, but it also has additional methods and properties that are specific to RealMedia. All other media types also exhibit this behavior.

For an illustration of how the cast member media types relate to each other and to other objects in Director, see "Object model diagrams" on page 44.

Animated GIF

Represents an animated GIF cast member.

You can add an animated GIF cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#animgif)

// JavaScript syntax
_movie.newMember("animgif");
```

Some of the following methods or properties may apply only to sprites that are created from an animated GIF cast member.

Method summary for the Animated GIF media type

Method
resume()
rewind() (Animated GIF, Flash)

Property summary for the Animated GIF media type

Property
directToStage
frameRate
linked
path (Movie)
playBackMode

See also

Member

Bitmap

Represents a bitmap cast member.

You can use bitmap image objects to perform simple operations that affect the content of an entire bitmap cast member, such as changing the background and foreground colors of the member, or to perform fine manipulations of the pixels of an image, such as cropping, drawing, and copying pixels.

You can add a bitmap cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#bitmap)

// JavaScript syntax
_movie.newMember("bitmap");
```

Some of the following methods or properties may apply only to sprites that are created from a bitmap cast member.

Method summary for the Bitmap media type

Method
crop() (Image)
pictureP()

Property summary for the Bitmap media type

Property	
alphaThreshold	imageCompression
backColor	imageQuality
blend (Sprite)	palette
depth (Bitmap)	picture (Member)
dither	rect (Image)
foreColor	trimWhiteSpace
image (Image)	useAlpha

See also

Member

Button

Represents a button or check box cast member.

You can add a button cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#button)

// JavaScript syntax
_movie.newMember("button");
```

Property summary for the Button media type

Property
hilite

See also

`Member`

Color Palette

Represents the color palette associated with a bitmap cast member.

A color palette cast member does not have any methods or properties that can be accessed directly from it. The following methods and properties are merely associated with color palettes.

You can add a color palette cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#palette)

// JavaScript syntax
_movie.newMember("palette");
```

You can associate a bitmap cast member with a color palette cast member by setting the `palette` property of the bitmap cast member. The following example sets the `palette` property of the bitmap cast member `bmpMember` to the color palette cast member `colorPaletteMember`. The value of the `palette` property reflects the number of the color palette cast member.

```
-- Lingo syntax
member("bmpMember").palette = member("colorPaletteMember")

// JavaScript syntax
member("bmpMember").palette = member("colorPaletteMember");
```

After you associate a bitmap cast member with a color palette cast member, you cannot delete the color palette cast member until you remove its association with the bitmap cast member.

Method summary for the Color Palette media type

Method
color()

Property summary for the Color Palette media type

Property
depth (Bitmap)
palette
paletteMapping

See also

Bitmap, Member, palette

Cursor

Represents a cursor cast member.

You can add a cursor cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#cursor)

// JavaScript syntax
_movie.newMember("cursor");
```

Property summary for the Cursor media type

Property
castMemberList
cursorSize
hotSpot
interval

See also

Member

DVD

Represents a DVD cast member.

You can add a DVD cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#dvd)

// JavaScript syntax
_movie.newMember("dvd");
```

Some of the following methods or properties may apply only to sprites that are created from a DVD cast member.

Event summary for the DVD media type

The following DVD events are always be handled by a `DVDeventNotification` event handler. When one of these events occurs, the `DVDeventNotification` event handler receives the event as a parameter. Some of these events also contain additional information that is passed as a second or third parameter to `DVDeventNotification`. For more information on using the following events with the `DVDeventNotification` handler, see "on DVDeventNotification" on page 154.

Event	
angleChange	noFirstPlayChain
audioStreamChange	parentalLevelChange
buttonChange	playbackStopped
chapterAutoStop	playPeriodAutoStop
chapterStart	rateChange
diskEjected	stillOff
diskInserted	stillOn
domainChange	titleChange
error	UOPchange
karaokeMode	warning

Method summary for the DVD media type

Method	
activateAtLoc()	rootMenu()
activateButton()	selectAtLoc()
frameStep()	selectButton()
chapterCount()	selectButtonRelative()
pause() (DVD)	stop() (DVD)
play() (DVD)	subPictureType()
returnToTitle()	titleMenu()

Property summary for the DVD media type

Property	
angle (DVD)	duration (DVD)
angleCount	folder
aspectRatio	frameRate (DVD)
audio (DVD)	fullScreen
audioChannelCount	mediaStatus (DVD)
audioExtension	playRate (DVD)

Property (Continued)	
audioFormat	resolution (DVD)
audioSampleRate	selectedButton
audioStream	startTimeList
audioStreamCount	stopTimeList
buttonCount	subPicture
chapter	subPictureCount
chapterCount	title (DVD)
closedCaptions	titleCount
currentTime (DVD)	videoFormat
domain	volume (DVD)

See also

Member

Field

Represents a field cast member.

You can add a field cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#field)

// JavaScript syntax
_movie.newMember("field");
```

Method summary for the Field media type

Method	
charPosToLoc()	pointToItem()
lineHeight()	pointToLine()
linePosToLocV()	pointToParagraph()
locToCharPos()	pointToWord()
locVToLinePos()	scrollByLine()
pointToChar()	scrollByPage()

Property summary for the Field media type

Property	
alignment	fontStyle
autoTab	lineCount
border	margin
boxDropShadow	pageHeight
boxType	scrollTop
dropShadow	selEnd
editable	selStart
font	text
fontSize	wordWrap

See also

Member

Film Loop

Represents a film loop cast member.

You can add a film loop cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#filmloop)

// JavaScript syntax
_movie.newMember("filmloop");
```

Property summary for the Film Loop media type

Property
media
regPoint

See also

Member

Flash Component

Represents a Adobe® Flash® component that is embedded in a cast member or sprite that contains Flash content.

A Flash component provides prepackaged functionality that extends the existing functionality of cast members or sprites that contain Flash content. They are created and supported entirely by the Director development community.

Director supports the following Flash components:

Flash component	Description
Button	A resizable rectangular user interface button.
CheckBox	A fundamental part of any form or web application; can be used wherever you need to gather a set of `true` or `false` values that aren't mutually exclusive.
DateChooser	A calendar that allows a user to select a date.
Label	A single line of text.
List	A scrollable single- or multiple-selection list box.
NumericStepper	Allows a user to step through an ordered set of numbers.
RadioButton	A fundamental part of any form or web application; can be used wherever you want a user to make one choice from a group of options.
ScrollPane	Displays movie clips, JPEG files, and SWF files in a scrollable area.
TextArea	A multiline text field.
TextInput	A single-line component that wraps the native ActionScript TextField object.
Tree	Allows a user to view hierarchical data.

A Flash component has access to the same APIs that a regular Flash cast member or sprite does, in addition to the functionality that is specific to that component. For more information about using these Flash components, see the Using Director topics in the Director Help Panel.

You can add a Flash component cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#flashcomponent)

// JavaScript syntax
_movie.newMember("flashcomponent");
```

See also

Flash Movie, Member

Flash Movie

Represents a cast member or sprite that contains Flash content.

You can add a Flash movie cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#flash)

// JavaScript syntax
_movie.newMember("flash");
```

A Flash movie cast member or sprite can also contain Flash components. Flash components provide prepackaged functionality that extends the existing functionality of Flash movie cast members or sprites. For more information about the Flash components that Director supports, see "Flash Component" on page 116.

Some of the following methods or properties may apply only to sprites that are created from a Flash movie cast member.

Method summary for the Flash Movie media type

Method	
callFrame()	printAsBitmap()
clearAsObjects()	rewind() (Animated GIF, Flash)
clearError()	setCallback()
findLabel()	setFlashProperty()
flashToStage()	settingsPanel()
getFlashProperty()	setVariable()
getVariable()	showProps()
goToFrame()	stageToFlash()
hitTest()	stop() (Flash)
hold()	stream()
newObject()	tellTarget()
print()	

Property summary for the Flash Movie media type

Property	
actionsEnabled	originPoint
broadcastProps	originV
bufferSize	playBackMode
buttonsEnabled	playing
bytesStreamed	posterFrame
centerRegPoint	quality
clickMode	rotation
defaultRect	scale (Member)
defaultRectMode	scaleMode
eventPassMode	sound (Member)
fixedRate	static
flashRect	streamMode
frameCount	streamSize
imageEnabled	viewH
linked	viewPoint

Property (Continued)	
mouseOverButton	viewScale
originH	viewV
originMode	

See also

Flash Component, Member

Font

Represents a font cast member.

You can add a font cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#font)

// JavaScript syntax
_movie.newMember("font");
```

Property summary for the Font media type

Property
bitmapSizes
characterSet
fontStyle
originalFont
recordFont

See also

Member

Linked Movie

Represents a linked movie cast member.

You can add a linked movie cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#movie)

// JavaScript syntax
_movie.newMember("movie");
```

Property summary for the Linked Movie media type

Property
scriptsEnabled

See also

Member

QuickTime

Represents a QuickTime® cast member.

You can add a QuickTime cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#quicktimemedia)

// JavaScript syntax
_movie.newMember("quicktimemedia");
```

Some of the following methods or properties may apply only to sprites that are created from a QuickTime cast member.

Method summary for the QuickTime media type

Method	
enableHotSpot()	qtRegisterAccessKey()
getHotSpotRect()	qtUnRegisterAccessKey()
nudge()	setTrackEnabled()
ptToHotSpotID()	swing()
QuickTimeVersion()	

Property summary for the QuickTime media type

Property	
audio (RealMedia)	scale (Member)
currentTime (QuickTime, AVI)	staticQuality
fieldOfView	tilt
hotSpotEnterCallback	trackCount (Member)
hotSpotExitCallback	trackCount (Sprite)
invertMask	trackEnabled
isVRMovie	trackNextKeyTime
loopBounds	trackNextSampleTime

Property (Continued)	
mask	trackPreviousKeyTime
motionQuality	trackPreviousSampleTime
mouseLevel	trackStartTime (Member)
node	trackStartTime (Sprite)
nodeEnterCallback	trackStopTime (Member)
nodeExitCallback	trackStopTime (Sprite)
nodeType	trackText
pan (QTVR property)	trackType (Member)
percentStreamed (Member)	trackType (Sprite)
playRate	translation
preLoad (Member)	triggerCallback
rotation	warpMode

See also

`Member`

RealMedia

Represents a RealMedia cast member.

You can add a RealMedia cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#realmedia)

// JavaScript syntax
_movie.newMember("realmedia");
```

Some of the following methods or properties may apply only to sprites that are created from a RealMedia cast member.

Method summary for the RealMedia media type

Method
pause() (RealMedia, SWA, Windows Media)
play() (RealMedia, SWA, Windows Media)
realPlayerNativeAudio()
realPlayerPromptToInstall()

Method (Continued)
realPlayerVersion()
seek()
stop() (RealMedia, SWA, Windows Media)

Property summary for the RealMedia media type

Property	
audio (RealMedia)	password
currentTime (RealMedia)	pausedAtStart (RealMedia, Windows Media)
displayRealLogo	percentBuffered
duration (RealMedia, SWA)	soundChannel (RealMedia)
image (RealMedia)	state (RealMedia)
lastError	userName (RealMedia)
mediaStatus (RealMedia, Windows Media)	video (RealMedia, Windows Media)

See also

`Member`

Shockwave 3D

Represents a Adobe® Shockwave® 3D cast member.

A Shockwave 3D (or simply 3D) cast member differs from other cast members in that a 3D cast member contains a complete 3D world. A 3D world contains a set of objects that are unique to 3D cast members, and that enable you to add 3D functionality to a movie.

You can add a 3D cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#shockwave3d)

// JavaScript syntax
_movie.newMember("shockwave3d");
```

For more information on the objects and APIs that are available to 3D cast members, see "3D Objects" on page 132.

See also

`Member`

Shockwave Audio

Represents a Shockwave Audio cast member.

You can add a Shockwave Audio cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#swa)

// JavaScript syntax
_movie.newMember("swa");
```

Event summary for the Shockwave Audio media type

Event
on cuePassed

Method summary for the Shockwave Audio media type

Method
getError() (Flash, SWA)
getErrorString()
isPastCuePoint()
pause() (RealMedia, SWA, Windows Media)
play() (RealMedia, SWA, Windows Media)
preLoadBuffer()
stop() (RealMedia, SWA, Windows Media)

Property summary for the Shockwave Audio media type

Property	
bitRate	percentStreamed (Member)
bitsPerSample	preLoadTime
channelCount	sampleRate
copyrightInfo (SWA)	sampleSize
cuePointNames	soundChannel (SWA)
cuePointTimes	state (Flash, SWA)
duration (RealMedia, SWA)	streamName
loop (Member)	URL
mostRecentCuePoint	volume (Member)
numChannels	

See also

Member

Sound

Represents a cast member that is used to store and refer to sound samples.

Sound samples are controlled by the core Sound and Sound Channel objects. A sound cast member does not have any APIs of its own, and uses the APIs of the Sound and Sound Channel objects to control its behavior.

You can add a sound cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#sound)

// JavaScript syntax
_movie.newMember("sound");
```

For more information on the objects and APIs you can use to control sound samples, see "Sound" on page 102 and "Sound Channel" on page 103.

See also

`Member`

Text

Represents a text cast member.

You can add a text cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#text)

// JavaScript syntax
_movie.newMember("text");
```

Event summary for the Text media type

Event
on hyperlinkClicked

Method summary for the Text media type

Method
count()
pointInHyperlink()
pointToChar()
pointToItem()
pointToLine()
pointToParagraph()
pointToWord()

Property summary for the Text media type

Property	
antiAlias	hyperlink
antiAliasThreshold	hyperlinkRange
bottomSpacing	hyperlinks
charSpacing	hyperlinkState
firstIndent	kerning
fixedLineSpace	kerningThreshold
font	RTF
fontStyle	selectedText
HTML	useHypertextStyles

See also

`Member`

Vector Shape

Represents a vector shape cast member.

You can add a vector shape cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#vectorshape)

// JavaScript syntax
_movie.newMember("vectorshape");
```

Some of the following methods or properties may apply only to sprites that are created from a vector shape cast member.

Method summary for the Vector Shape media type

Method
addVertex()
deleteVertex()
moveVertex()
moveVertexHandle()
newCurve()
showProps()

Property summary for the Vector Shape media type

Property	
antiAlias	imageEnabled
backgroundColor	originH
broadcastProps	originMode
centerRegPoint	originPoint
closed	originV
curve	regPointVertex
defaultRect	scale (Member)
defaultRectMode	scaleMode
endColor	strokeColor
fillColor	strokeWidth
fillCycles	vertex
fillDirection	vertexList
fillMode	viewH
fillOffset	viewPoint
fillScale	viewScale
flashRect	viewV
gradientType	

See also

Member

Windows Media

Represents a Windows Media® cast member.

You can add a Windows Media cast member to a movie by using the Movie object's `newMember()` method.

```
-- Lingo syntax
_movie.newMember(#windowsmedia)

// JavaScript syntax
_movie.newMember("windowsmedia");
```

Some of the following methods or properties may apply only to sprites that are created from a Windows Media cast member.

Method summary for the Windows Media media type

Method
pause() (RealMedia, SWA, Windows Media)
play() (RealMedia, SWA, Windows Media)
playFromToTime()
rewind() (Windows Media)
stop() (RealMedia, SWA, Windows Media)

Property summary for the Windows Media media type

Property	
audio (Windows Media)	pausedAtStart (RealMedia, Windows Media)
directToStage	playRate (Windows Media)
duration (Member)	video (RealMedia, Windows Media)
height	volume (Windows Media)
loop (Windows Media)	width
mediaStatus (RealMedia, Windows Media)	

See also

Member

Chapter 7: Scripting Objects

The scripting objects, also known as Xtra extensions, in Director provide access to the functionality of the software components that are installed with Director® and extend core Director functionary. The preexisting Xtra extensions provide capabilities such as importing filters and connecting to the Internet. If you know the C programming language, you can create your own custom Xtra extensions.

For an illustration of how the scripting objects relate to each other and to other objects in Director, see "Object model diagrams" on page 44.

Fileio

Enables you to perform file input and output operations.

You can create a reference to a Fileio object by using the `new` operator.

```
-- Lingo syntax
objFileio = new xtra("fileio")

// JavaScript syntax
var objFileio = new xtra("fileio");
```

Method summary for the Fileio object

Method	
closeFile()	readFile()
createFile()	readLine()
delete()	readToken()
deleteFile()	readWord()
displayOpen()	setFilterMask()
displaySave()	setFinderInfo()
error()	setNewLineConversion()
fileName()	setPosition()
getFinderInfo()	status()
getLength()	
getOSDirectory()	version()
getPosition()	writeChar()
openFile()	writeReturn()
readChar()	writeString()

MUI Xtra

The MUI Xtra provides fully functional dialog boxes set up the way that you specify. These dialog boxes don't require the memory or disk footprint of a MIAW that simulates a dialog box.

You can create a reference to an MUI Xtra object by using the `new` operator.

```
-- Lingo syntax
objMui = new xtra("Mui")

// JavaScript syntax
var objMui = new xtra("Mui");
```

Method summary for the XML Parser object

Method
Alert()
fileOpen()
fileSave()
GetItemPropList
getURL()
GetWidgetList()
GetWindowPropList
Initialize
ItemUpdate()
run
stop
WindowOperation

NetLingo

Enables you to perform network operations such as obtaining or streaming media from a network, checking network availability, checking the progress of a network operation, and so on.

You can create a reference to a NetLingo object by using the `new` operator.

```
-- Lingo syntax
objNetLingo = new xtra("netlingo")

// JavaScript syntax
var objNetLingo = new xtra("netlingo");
```

Method summary for the NetLingo object

Method	
browserName()	netDone()
cacheDocVerify()	netError()
cacheSize()	netLastModDate()
clearCache	netMIME()
downloadNetThing	netStatus
externalEvent()	netTextResult()
getLatestNetID	postNetText
getNetText()	preloadNetThing()
getStreamStatus()	proxyServer
gotoNetMovie	tellStreamStatus()
gotoNetPage	URLEncode
netAbort	

SpeechXtra

Enables you to add text-to-speech functionality to a movie.

You can create a reference to a SpeechXtra object by using the `new` operator.

```
-- Lingo syntax
objSpeech = new xtra("speechxtra")

// JavaScript syntax
var objSpeech = new xtra("speechxtra");
```

Method summary for the SpeechXtra object

Method	
voiceCount()	voiceSet()
voiceGet()	voiceSetPitch()
voiceGetAll()	voiceSetRate()
voiceGetPitch()	voiceSetVolume()
voiceGetRate()	voiceSpeak()
voiceGetVolume()	voiceState()
voiceInitialize()	voiceStop()
voicePause()	voiceWordPos()
voiceResume()	

XML Parser

Enables you to perform XML parsing.

You can create a reference to an XML Parser object by using the `new` operator.

```
-- Lingo syntax
objXml = new xtra("xmlparser")

// JavaScript syntax
var objXml = new xtra("xmlparser");
```

Method summary for the XML Parser object

Method
count()
doneParsing()
getError() (XML)
ignoreWhiteSpace()
makeList()
makeSubList()
parseString()
parseURL()

Property summary for the XML Parser object

Property
attributeName
attributeValue
child (XML)
name (XML)

Chapter 8: 3D Objects

The 3D objects enable you to add 3D functionality to a movie. These objects are exposed to both Lingo and JavaScript syntax within Director, projectors, and the Adobe® Shockwave® Player.

You access these 3D objects through Shockwave 3D (or simply 3D) cast members. You can also create 3D sprites from the 3D cast members. Both 3D cast members and 3D sprites contain functionality that is specific to 3D cast members and sprites. They also have access to the functionality of non-3D cast members and sprites, whose APIs are specified by the core `Member` and `Sprite` objects, respectively.

A 3D cast member differs from a non-3D cast member in that a 3D cast member contains a complete 3D world. A 3D world contains the objects that provide access to 3D functionality. All objects in a 3D world are based on a basic object known as a node. The simplest form of a node in a 3D world is a Group object; a Group object is essentially the most basic node. All other objects in a 3D world are based on a Group object, which means that the other objects inherit the functionality of a Group object in addition to containing functionality that is specific to those objects.

For an illustration of how the 3D objects relate to each other and to other objects in Director, see "Object model diagrams" on page 44.

Director® ships with two Xtra extensions that enable access to the 3D objects:

- 3D Asset Xtra (3DAuth.x32 in Windows®, 3D Auth Xtra on Mac®) provides support for the 3D media window inside Director.
- 3D Media Xtra (Shockwave 3D Asset.x32 in Windows, 3D Asset Xtra on Mac) provides support for the 3D media itself.

To access the 3D objects during authoring or at runtime, your movie must contain the 3D Asset Xtra.

Camera

Represents a Camera object.

A camera controls how a 3D sprite views the 3D world. A 3D sprite displays a particular camera's view into the world.

You can create a reference to a camera by using the `camera` property of the 3D `Member` object. The `camera` property gets the camera at a specified index position in the list of cameras. In Lingo, you use the `camera` property directly from the 3D `Member` object to create a reference. In JavaScript syntax, you must use the `getPropRef()` method to create a reference.

The following example creates a reference to the second camera of the 3D cast member "family room", and assigns it to the variable `myCamera`.

```
-- Lingo syntax
myCamera = member("family room").camera[2]

// JavaScript syntax
var myCamera = member("family room").getPropRef("camera", 2);
```

Method summary for the Camera object

Method
addBackdrop
addOverlay
insertBackdrop
insertOverlay
removeBackdrop
removeOverlay

Property summary for the Camera object

Property	
backdrop	fog.far (fog)
backdrop[].blend (3D)	fog.near (fog)
backdrop[].loc (backdrop and overlay)	hither
backdrop[].regPoint (3D)	orthoHeight
backdrop[].rotation (backdrop and overlay)	overlay
backdrop[].scale (3D)	overlay[].blend (3D)
backdrop[].source	overlay[].loc (backdrop and overlay)
backdrop.count (3D)	overlay[].regPoint (3D)
child (3D)	overlay[].rotation (backdrop and overlay)
colorBuffer.clearAtRender	overlay[].scale (3D)
colorBuffer.clearValue	overlay[].source
fieldOfView (3D)	overlay.count (3D)
fog.color()	projection
fog.decayMode	rootNode
fog.enabled (fog)	yon

See also

`Group`, `Light`, `Model`, `Model Resource`, `Motion`, `Shader`, `Texture`

Group

Represents a model that does not have a resource or any shaders.

A group is the most basic node, and is merely a point in space that is represented by a transform. You can assign children and parents to this node in order to group models, lights, cameras, or other groups.

The most basic group is called a world, which is essentially synonymous with a 3D cast member.

You can create a reference to a group by using the `group` property of the 3D `Member` object. The `group` property gets the group at a specified index position in the list of groups. In Lingo, you use the `group` property directly from the 3D `Member` object to create a reference. In JavaScript syntax, you must use the `getPropRef()` method to create a reference.

The following example creates a reference to the first group of the 3D cast member `space`, and assigns it to the variable `myGroup`.

```
-- Lingo syntax
myGroup = member("space").group[1]

// JavaScript syntax
var myGroup = member("space").getPropRef("group", 1);
```

Method summary for the Group object

Method	
addChild	pointAt
addToWorld	registerScript()
clone	removeFromWorld
cloneDeep	rotate
getWorldTransform()	scale (command)
isInWorld()	translate

Property summary for the Group object

Property
name (3D)
parent
pointAtOrientation
transform (property)
userData
worldPosition

See also

`Camera`, `Light`, `Model`, `Model Resource`, `Motion`, `Shader`, `Texture`

Light

Represents a light in a 3D world.

Lights are used to light a 3D world. Without lights, the objects within the world cannot be seen.

You can create a reference to a light by using the `light` property of the 3D `Member` object. The `light` property gets the light at a specified index position in the list of lights. In Lingo, you use the `light` property directly from the 3D `Member` object to create a reference. In JavaScript syntax, you must use the `getPropRef()` method to create a reference.

The following example creates a reference to the third light of the 3D cast member "film room" and assigns it to the variable `myLight`.

```
-- Lingo syntax
myLight = member("film room").light[3]

// JavaScript syntax
var myLight = member("film room").getPropRef("light", 3);
```

Property summary for the Light object

Property
attenuation
color (light)
specular (light)
spotAngle
spotDecay
type (light)

See also

`Camera`, `Group`, `Model`, `Model Resource`, `Motion`, `Shader`, `Texture`

Member

Represents a Shockwave 3D cast member.

A Shockwave 3D (or simply 3D) cast member contains a complete 3D world. A 3D world contains the set of objects you use to add 3D functionality to a movie.

You can create a reference to a 3D cast member by using either the top level `member()` function, or by using the `member` property of the Movie or Sprite object. These are the same techniques you can use to create a reference to a non-3D cast member.

- Use the top level `member()` function.

```
-- Lingo syntax
3dMember = member("magic")

// JavaScript syntax
var 3dMember = member("magic");
```

- Use the Sprite object's `member` property.

```
-- Lingo syntax
3dMember = sprite(1).member;

// JavaScript syntax
```

```
var 3dMember = sprite(1).member;
```

Method summary for the Member object

Method	
camera()	model
cloneModelFromCastmember	modelResource
cloneMotionFromCastmember	motion()
deleteCamera	newCamera
deleteGroup	newGroup
deleteLight	newLight
deleteModel	newMesh
deleteModelResource	newModel
deleteMotion	newModelResource
deleteShader	newShader
deleteTexture	newTexture
extrude3D	resetWorld
group()	revertToWorldDefaults
light()	shader()
loadFile()	texture()

Property summary for the Member object

Property	
ambientColor	loop (3D)
animationEnabled	model
bevelDepth	modelResource
bevelType	motion
bytesStreamed (3D)	percentStreamed (3D)
camera	preLoad (3D)
cameraPosition	reflectivity
cameraRotation	shader
diffuseColor	smoothness
directionalColor	specularColor
directionalPreset	state (3D)
directToStage	streamSize (3D)
displayFace	texture

Property (Continued)	
displayMode	textureMember
group	textureType
light	tunnelDepth

See also

Camera, Group, Light, Model, Model Resource, Motion, Shader, Sprite, Texture

Model

Represents a visible object that a user sees within a 3D world.

A model makes use of a model resource and occupies a specific position and orientation with a 3D world. A model resource is an element of 3D geometry that can be used to draw 3D models. A model also defines the appearance of the model resource, such as what textures and shaders are used. For more information about the relationship between models and model resources, see the Using Director topics in the Director Help Panel.

You can create a reference to a model by using the `model` property of the 3D `Member` object. The `model` property gets the model at a specified index position in the list of models. In Lingo, you use the `model` property directly from the 3D `Member` object to create a reference. In JavaScript syntax, you must use the `getPropRef()` method to create a reference.

The following example creates a reference to the second model of the 3D cast member `Transportation` and assigns it to the variable `myModel`.

```
-- Lingo syntax
myModel = member("Transportation").model[2]

// JavaScript syntax
var myModel = member("Transportation").getPropRef("model", 2);
```

A model also contains modifiers that control how the model is rendered or how its animation behaves. Modifiers are attached to a model by using the `addModifier()` method. After a modifier has been attached to a model, its properties can be manipulated with script.

The following modifiers are available to a model:

Modifier	Description
Bones player	Modifies a model's geometry over time.
Collision	Allows a model to be notified of and respond to collisions.
Inker	Adds silhouette, crease, and boundary edges to an existing model.
Keyframe player	Modifies a model's `transform` properties over time.
Level of detail (LOD)	Provides per-model control over the number of polygons used to render a model, based on the model's distance from a camera. The LOD modifier is also available to model resources.

Modifier	Description
Mesh deform	Alters an existing model resource's geometry at runtime.
Subdivision surfaces (SDS)	Causes the model to be rendered with additional geometric detail in the area of the model that the camera is currently looking at.
Toon	Changes a model's rendering to imitate a cartoon drawing style.

For more information about the methods, properties, and events available to the modifiers, see the Using Director topics in the Director Help Panel.

Model Resource

Represents an element of 3D geometry that is used to draw 3D models.

A model makes use of a model resource and occupies a specific position and orientation with a 3D world. A model also defines the appearance of the model resource, such as what textures and shaders are used.

For more information about the relationship between models and model resources, and about using models and model resources, see the Using Director topics in the Director Help Panel.

You can create a reference to a model resource by using the `modelResource` property of the 3D `Member` object. The `modelResource` property gets the model resource at a specified index position in the list of model resources. In Lingo, you use the `modelResource` property directly from the 3D `Member` object to create a reference. In JavaScript syntax, you must use the `getPropRef()` method to create a reference.

The following example creates a reference to the second model resource of the 3D cast member `wheels` and assigns it to the variable `myModelResource`.

```
-- Lingo syntax
myModelResource = member("wheels").modelResource[2]

// JavaScript syntax
var myModelResource = member("wheels").getPropRef("modelResource", 2);
```

Motion

Represents a predefined animation sequence that involve the movement of a model or a model component.

Individual motions can be set to play by themselves or with other motions. For example, a running motion can be combined with a jumping motion to simulate a person jumping over a puddle.

You can create a reference to a motion by using the `motion` property of the 3D `Member` object. The `motion` property gets the motion at a specified index position in the list of motions. In Lingo, you use the `motion` property directly from the 3D `Member` object to create a reference. In JavaScript syntax, you must use the `getPropRef()` method to create a reference.

The following example creates a reference to the fourth motion of the 3D cast member `athlete` and assigns it to the variable `myMotion`.

```
-- Lingo syntax
myMotion = member("athlete").motion[4]
```

```
// JavaScript syntax
var myMotion = member("athlete").getPropRef("motion", 4);
```

Renderer Services

Represents the global object that contains a property list whose values impact common rendering properties for all 3D cast members and sprites.

You can access the global renderer services object using the top level `getRendererServices()` function.

The following example accesses the renderer property of the global renderer services object and assigns the value to the variable `myRenderer`.

```
-- Lingo syntax
myRenderer = getRendererServices().renderer

// JavaScript syntax
var myRenderer = getRendererServices().renderer;
```

Method summary for the Renderer Services object

Method
getHardwareInfo()

Property summary for the Renderer Services object

Property
modifiers
primitives
renderer
rendererDeviceList
textureRenderFormat

See also

`Member`, `Sprite`

Shader

Represents a model's surface color.

You can draw images on the surface of a model by applying one or more textures to each shader.

You can create a reference to a shader by using the `shader` property of the 3D `Member` object. The `shader` property gets the shader at a specified index position in the list of shaders. In Lingo, you use the `shader` property directly from the 3D `Member` object to create a reference. In JavaScript syntax, you must use the `getPropRef()` method to create a reference.

The following example creates a reference to the second shader of the 3D cast member `triangle` and assigns it to the variable `myShader`.

```
-- Lingo syntax
myShader = member("triangle").shader[2]

// JavaScript syntax
var myShader = member("triangle").getPropRef("shader", 2);
```

Sprite

Represents a 3D sprite created from a Shockwave 3D cast member.

You can create a reference to a 3D sprite by using the top level `sprite()` function, the Movie object's `sprite` property, or the Sprite Channel object's `sprite` property. These are the same techniques you can use to create a reference to a non-3D sprite.

- Use the top level `sprite()` function.

```
-- Lingo syntax
3dSprite = sprite(1)

// JavaScript syntax
var 3dSprite = sprite(1);
```

- Use the Movie object's `sprite` property.

```
-- Lingo syntax
3dSprite = _movie.sprite["willowTree"]

// JavaScript syntax
var 3dSprite = _movie.sprite["willowTree"];
```

- Use the Sprite Channel object's `sprite` property.

```
-- Lingo syntax
3dSprite = channel(3).sprite

// JavaScript syntax
var 3dSprite = channel(3).sprite;
```

Method summary for the Sprite object

Method
addCamera
cameraCount()
deleteCamera

Property summary for the Sprite object

Property
antiAliasingEnabled
backColor
camera
directToStage

See also

`Camera`, `Member`

Texture

Represents the texture applied to a shader.

You can create a reference to a texture by using the `texture` property of the 3D `Member` object. The `texture` property gets the texture at a specified index position in the list of textures. In Lingo, you use the `texture` property directly from the 3D `Member` object to create a reference. In JavaScript syntax, you must use the `getPropRef()` method to create a reference.

The following example creates a reference to the first texture of the 3D cast member `triangle` and assigns it to the variable `myTexture`.

```
-- Lingo syntax
myTexture = member("triangle").texture[1]

// JavaScript syntax
var myTexture = member("triangle").getPropRef("texture", 1);
```

Chapter 9: Constants

This section provides an alphabetical list of all the constants available in Director®.

The majority of these constants apply only to Lingo. JavaScript syntax does contain some constants that are similar to the Lingo constants listed here; therefore, where appropriate, JavaScript syntax usage and examples are provided to help you map the functionality of Lingo constants with their closest counterparts in JavaScript syntax. For more information about JavaScript syntax constants, see one of the many third-party resources on the subject.

" (string)

Usage

```
--Lingo syntax
"

// JavaScript syntax
"
```

Description

String constant; when used before and after a string, quotation marks indicate that the string is a literal—not a variable, numerical value, or script element. Quotation marks must always surround literal names of cast members, casts, windows, and external files.

Example

This statement uses quotation marks to indicate that the string “San Francisco” is a literal string, the name of a cast member:

```
--Lingo syntax
put member("San Francisco").loaded

// JavaScript syntax
put(member("San Francisco").loaded);
```

See also

QUOTE

BACKSPACE

Usage

```
-- Lingo syntax
BACKSPACE

// JavaScript syntax
51 // value of _key.keyCode
```

Description

Constant; represents the Backspace key. This key is labeled Backspace (Windows®) and Delete (Mac®).

Example

This `on keyDown` handler checks whether the Backspace key was pressed and, if it was, calls the handler `clearEntry`:

```
--Lingo syntax
on keyDown
    if (_key.key = BACKSPACE) then clearEntry
    _movie.stopEvent()
end keyDown

// JavaScript syntax
function keyDown() {
    if (_key.keyCode == 51) {
        clearEntry();
        _movie.stopEvent();
    }
}
```

EMPTY

Usage

```
--Lingo syntax
EMPTY

// JavaScript syntax
""
```

Description

Character constant; represents the empty string, `""`, a string with no characters.

Example

This statement erases all characters in the field cast member Notice by setting the field to `EMPTY`:

```
--Lingo syntax
member("Notice").text = EMPTY

// JavaScript syntax
member("Notice").text = "";
```

ENTER

Usage

```
--Lingo syntax
ENTER

// JavaScript syntax
3 // value of _key.keyCode
```

Description

Character constant; represents Enter (Windows) or Return (Mac) for a carriage return.

On PC keyboards, the element `ENTER` refers only to Enter on the numeric keypad.

For a movie that plays back as an applet, use RETURN to specify both Return in Windows and Enter on the Mac.

Example

This statement checks whether Enter is pressed and if it is, sends the playhead to the frame addSum:

```
-- Lingo syntax
on keyDown
    if (_key.key = ENTER) then _movie.go("addSum")
end

// JavaScript syntax
function keyDown() {
    if (_key.keyCode == 3) {
        _movie.go("addSum");
    }
}
```

See also

RETURN (constant)

FALSE

Usage

```
-- Lingo syntax
FALSE

// JavaScript syntax
false
```

Description

Constant; applies to an expression that is logically FALSE, such as 2 > 3. When treated as a number value, FALSE has the numerical value of 0. Conversely, 0 is treated as FALSE.

Example

This statement turns off the soundEnabled property by setting it to FALSE:

```
-- Lingo syntax
_sound.soundEnabled = FALSE

// JavaScript syntax
_sound.soundEnabled = false;
```

See also

if, not, TRUE

PI

Usage

```
-- Lingo syntax
PI

// JavaScript syntax
Math.PI
```

Description

Constant; returns the value of pi (π), the ratio of a circle's circumference to its diameter, as a floating-point number. The value is rounded to the number of decimal places set by the `floatPrecision` property.

Example

This statement uses the `PI` constant as part of an equation for calculating the area of a circle:

```
-- Lingo syntax
vRadius = 3
vArea = PI*power(vRadius, 2)
trace(vArea) -- results in 28.2743

// JavaScript syntax
var vRadius = 3;
vArea = Math.PI*Math.pow(vRadius, 2);
trace(vArea); // results in 28.274333882308138
```

QUOTE

Usage

```
--Lingo syntax
QUOTE

// JavaScript syntax
\"
```

Description

Constant; represents the quotation mark character and refers to the literal quotation mark character in a string, because the quotation mark character itself is used by Lingo scripts to delimit strings.

Example

This statement inserts quotation mark characters in a string:

```
-- Lingo syntax
put("Can you spell" && QUOTE & "Adobe" & QUOTE & "?")

// JavaScript syntax
put("Can you spell \"Adobe\"?");
```

The result is a set of quotation marks around the word Adobe®:

```
Can you spell "Adobe"?
```

RETURN (constant)

Usage

```
-- Lingo syntax
RETURN

// JavaScript syntax
36 // value of _key.keyCode
\n // when used in a string
```

Description

Constant; represents a carriage return.

Example

This statement causes a paused movie to continue when the user presses the carriage return:

```
-- Lingo syntax
if (_key.key = RETURN) then _movie.go(_movie.frame + 1)

// JavaScript syntax
if (_key.keyCode == 36) {
    _movie.go(_movie.frame + 1);
}
```

This statement uses the `RETURN` character constant to insert a carriage return between two lines in an alert message:

```
-- Lingo syntax
_player.alert("Last line in the file." & RETURN & "Click OK to exit.")

// JavaScript syntax
_player.alert("Last line in the file." + "\n" + " Click OK to exit");
```

In Windows, it is standard practice to place an additional line-feed character at the end of each line. This statement creates a two-character string named CRLF that provides the additional line feed:

```
CRLF = RETURN & numToChar(10)
```

SPACE

Usage

```
-- Lingo syntax
SPACE

// JavaScript syntax
49 // value of _key.keyCode
```

Description

Constant; read-only, value that represents the space character.

Example

This statement displays “Age Of Aquarius” in the Message window:

```
-- Lingo syntax
put("Age"&SPACE&"Of"&SPACE&"Aquarius")
```

TAB

Usage

```
-- Lingo syntax
TAB

// JavaScript syntax
48 // value of _key.keyCode
```

Description

Constant; represents the Tab key.

Example

This statement checks whether the character typed is the tab character and calls the handler `doNextField` if it is:

```
-- Lingo syntax
if (_key.key = TAB) then doNextField

// JavaScript syntax
if (_key.keyCode == 48) {
    doNextField();
}
```

These statements move the playhead forward or backward, depending on whether the user presses Tab or Shift+Tab:

```
-- Lingo syntax
if (_key.key = TAB) then
    if (_key.shiftDown) then
        _movie.go(_movie.frame - 1)
    else
        _movie.go(_movie.frame + 1)
    end if
end if

// JavaScript syntax
if (_key.keyCode == 48) {
    if (_key.shiftDown) {
        _movie.go(_movie.frame - 1);
    } else {
        _movie.go(_movie.frame + 1);
    }
}
```

See also

BACKSPACE, EMPTY, RETURN (constant)

TRUE

Usage

```
-- Lingo syntax
TRUE

// JavaScript syntax
true
```

Description

Constant; represents the value of a logically true expression, such as `2 < 3`. It has a traditional numerical value of 1, but any nonzero integer evaluates to `TRUE` in a comparison.

Example

This statement turns on the `soundEnabled` property by setting it to `TRUE`:

```
-- Lingo syntax
_sound.soundEnabled = TRUE
```

```
// JavaScript syntax
_sound.soundEnabled = true;
```

See also
FALSE, if

VOID

Usage

```
-- Lingo syntax
VOID

// JavaScript syntax
null
```

Description

Constant; indicates the value VOID.

Example

This statement checks whether the value in the variable currentVariable is VOID:

```
-- Lingo syntax
if currentVariable = VOID then
    put("This variable has no value")
end if

// JavaScript syntax
if (currentVariable == undefined) {
    put("This variable has no value");
}
```

See also
voidP()

Chapter 10: Events and Messages

This section provides an alphabetical list of all the events and messages available in Director®.

on activateApplication

Usage

```
-- Lingo syntax
on activateApplication
    statement(s)
end

// JavaScript syntax
function activateApplication() {
    statement(s);
}
```

Description

Built-in handler; runs when the projector is brought to the foreground. This handler is useful when a projector runs in a window and the user can send it to the background to work with other applications. When the projector is brought back to the foreground, this handler runs. Any MIAWs running in the projector can also make use of this handler.

During authoring, this handler is called only if Animate in Background is turned on in General Preferences.

On Windows®, this handler is not called if the projector is merely minimized and no other application is brought to the foreground.

Example

This handler plays a sound each time the user brings the projector back to the foreground:

```
-- Lingo syntax
on activateApplication
    sound(1).queue(member("openSound"))
    sound(1).play()
end

// JavaScript syntax
function activateApplication() {
    sound(1).queue(member("openSound"));
    sound(1).play();
}
```

See also

on deactivateApplication, activeCastLib, on deactivateWindow

on activateWindow

Usage

```
-- Lingo syntax
on activateWindow
        statement(s)
end

// JavaScript syntax
function activateWindow()
    statement(s);
}
```

Description

System message and event handler; contains statements that run in a movie when the user clicks the inactive window and the window comes to the foreground.

You can use an `on activateWindow` handler in a script that you want executed every time the movie becomes active.

Clicking the main movie (the main Stage) does not generate an `on activateWindow` handler.

Example

This handler plays the sound Hurray when the window that the movie is playing in becomes active:

```
-- Lingo syntax
on activateWindow
    sound(2).play(member("Hurray"))
end

// JavaScript syntax
function activateWindow() {
    sound(2).play(member("Hurray"));
}
```

See also

activeWindow, close(), on deactivateWindow, frontWindow, on moveWindow, open() (Window)

on beginSprite

Usage

```
-- Lingo syntax
on beginSprite
    statement(s)
end

// JavaScript syntax
function beginSprite() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run when the playhead moves to a frame that contains a sprite that was not previously encountered. Like `endSprite`, this event is generated only one time, even if the playhead loops on a frame, since the trigger is a sprite not previously encountered by the playhead. The event is generated before `prepareFrame`.

Director creates instances of any behavior scripts attached to the sprite when the `beginSprite` message is sent.

The object reference `me` is passed to this event if it is used in a behavior. The message is sent to behaviors and frame scripts.

If a sprite begins in the first frame that plays in the movie, the `beginSprite` message is sent after the `prepareMovie` message but before the `prepareFrame` and `startMovie` messages.

Note: *Be aware that some sprite properties, such as the* `rect` *sprite property, may not be accessible in a beginSprite handler. This is because the property needs to be calculated, which is not done until the sprite is drawn.*

The `go`, `play`, and `updateStage` commands are disabled in an `on beginSprite` handler.

Example

This handler plays the sound cast member Stevie Wonder when the sprite begins:

```
-- Lingo syntax
on beginSprite me
    sound(1).play(member("Stevie Wonder"))
end

// JavaScript syntax
function beginSprite() {
    sound(1).play(member("Stevie Wonder"));
}
```

See also

on endSprite, on prepareFrame, scriptInstanceList

on closeWindow

Usage

```
-- Lingo syntax
on closeWindow
    statement(s)
end

// JavaScript syntax
function closeWindow() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run when the user closes the window for a movie by clicking the window's close box.

The `on closeWindow` handler is a good place to put Lingo commands that you want executed every time the movie's window closes.

Example

This handler tells Director to `forget` the current window when the user closes the window that the movie is playing in:

```
-- Lingo syntax
on closeWindow
    -- perform general housekeeping here
    window(1).forget()
end

// JavaScript syntax
function closeWindow() {
    // perform general housekeeping here
    window(1).forget();
}
```

on cuePassed

Usage

```
-- Lingo syntax
on cuePassed({me,} channelID, cuePointNumber,cuePointName)
    statement(s)
end

// JavaScript syntax
function cuePassed(channelID, cuePointNumber,cuePointName) {
    statement(s);
}
```

Description

System message and event handler; contains statements that run each time a sound or sprite passes a cue point in its media.

- `me` The optional `me` parameter is the `scriptInstanceRef` value of the script being invoked. You must include this parameter when using the message in a behavior. If this parameter is omitted, the other arguments will not be processed correctly.
- *`channelID`* The number of the sound or sprite channel for the file where the cue point occurred.
- *`cuePointNumber`* The ordinal number of the cue point that triggers the event in the list of the cast member's cue points.
- *`cuePointName`* The name of the cue point that was encountered.

The message is passed—in order—to sprite, cast member, frame, and movie scripts. For the sprite to receive the event, it must be the source of the sound, like a QuickTime® movie or SWA cast member. Use the `isPastCuePoint` property to check cues in behaviors on sprites that don't generate sounds.

Example

This handler placed in a Movie or Frame script reports any cue points in sound channel 1 to the Message window:

```
-- Lingo syntax
on cuePassed channel, number, name
    if (channel = #Sound1) then
        put("CuePoint" && number && "named" && name && "occurred in sound 1")
    end if
```

```
end

// JavaScript syntax
function cuePassed(channel, number, name) {
    if (channel == symbol("Sound1")) {
        put("CuePoint " + number + " named " + name + "occurred in sound 1");
    }
}
```

See also

scriptInstanceList, cuePointNames, cuePointTimes, isPastCuePoint()

on deactivateApplication

Usage

```
-- Lingo syntax
on deactivateApplication
    statement(s)
end

// JavaScript syntax
function deactivateApplication() {
    statement(s);
}
```

Description

Built-in handler; runs when the projector is sent to the background. This handler is useful when a projector runs in a window and the user can send it to the background to work with other applications. Any MIAWs running in the projector can also make use of this handler.

During authoring, this handler is called only if Animate in Background is turned on in General Preferences.

On Windows, this handler is not called if the projector is merely minimized and no other application is brought to the foreground.

Example

This handler plays a sound each time the user sends the projector to the background:

```
-- Lingo syntax
on deactivateApplication
    sound(1).queue(member("closeSound"))
    sound(1).play()
end

// JavaScript syntax
function deactivateApplication() {
    sound(1).queue(member("closeSound"));
    sound(1).play();
}
```

See also

add (3D texture), activeCastLib, on deactivateWindow

on deactivateWindow

Usage

```
-- Lingo syntax
on deactivateWindow
    statement(s)
end

// JavaScript syntax
function deactivateWindow() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run when the window that the movie is playing in is deactivated. The `on deactivate` event handler is a good place for Lingo that you want executed whenever a window is deactivated.

Example

This handler plays the sound Snore when the window that the movie is playing in is deactivated:

```
-- Lingo syntax
on deactivateWindow
    sound(2).play(member("Snore"))
end

// JavaScript syntax
function deactivateWindow() {
    sound(2).play(member("Snore"));
}
```

on DVDeventNotification

Usage

```
-- Lingo syntax
on DVDeventNotification objectRef, event {, eventArg1} {, eventArg2} {, eventArg3}
    statement(s)
end DVDeventNotification

// JavaScript syntax
function DVDeventNotification (objectRef, event {, eventArg1} {, eventArg2} {, eventArg3}) {
    statement(s);
}
```

Description

Author-specified DVD event handler. Contains statements that run in response to events that occur while a DVD is playing.

This handler can be used to track all DVD events. In the script examples above, *`objectRef`*, the first parameter passed to the DVDeventNotification handler, is a reference to the DVDeventNotification object itself. The actual event that occurs is always passed as the second parameter, *`event`*. Some events contain additional information about them that is passed as a third parameter, *`eventArg1`*. In some cases, a fourth and fifth parameter, *`eventArg2`* and *`eventArg3`*, may contain additional event information.

The following table lists the events that can occur while a DVD is playing.

Event	Description
angleChange	Occurs when either the number of available angles changed or the current user angle number changed. The following additional information is passed to `DVDeventNotification` when this event occurs: • *`eventArg2`* - An integer that indicates the number of available angles. When the number of available angles is 1, the current video is not multiangle. • *`eventArg3`* - An integer that indicates the current user angle number.
audioStreamChange	Occurs when the current user audio stream number changed for the main title. The following additional information is passed to `DVDeventNotification` when this event occurs: • *`eventArg2`* - An integer that indicates the new user audio stream number. Stream `0xFFFFFFFF` indicates that no stream is selected.
buttonChange	Occurs when either the number of available buttons changed or the currently selected button number changed. The following additional information is passed to `DVDeventNotification` when this event occurs: • *`eventArg2`* - An integer that indicates the number of available buttons. • *`eventArg3`* - An integer that indicates the currently selected button number. Selected button number 0 implies that no button is selected.
chapterAutoStop	Occurs when playback stopped as a result of an automatic stop.
chapterStart	Occurs when playback of a new program in the `title` domain starts. The following additional information is passed to `DVDeventNotification` when this event occurs: • *`eventArg2`* - An integer that indicates the new chapter number.
diskEjected	Occurs when a DVD is ejected.
diskInserted	Occurs when a DVD is inserted.
domainChange	Occurs when the DVD player's domain changes. The following additional information is passed to `DVDeventNotification` when this event occurs: • *`eventArg1`*. A value that indicates the new domain. The new domain will be one of the following values. • `firstPlay`. The DVD Navigator is performing default initialization of a DVD. • `videoManagerMenu`. The DVD Navigator is displaying menus for the whole disc. • `videoTitleSetMenu`. The DVD Navigator is displaying menus for the current title set. • `title`. The DVD Navigator is displaying the current title. • `stop`. The DVD Navigator is in the `stop` domain.

Event	Description
error	Occurs when a DVD error condition is encountered. The following additional information is passed to `DVDeventNotification` when this event occurs: • *`eventArg2`*. A value that indicates error condition. The error condition will be one of the following values. • `copyProtectFail`. Key exchange for DVD copy protection failed. Playback is stopped. • `invalidDVD1_0Disc`. DVD-Video disc is authored incorrectly for specification version 1.*x*. Playback is stopped. • `invalidDiscRegion`. DVD-Video disc cannot be played because the disc is not authored to play in the system region. • `lowParentalLevel`. Player parental level is lower than the lowest parental level available in the DVD content. Playback is stopped. • `macrovisionFail`. Macrovision distribution failed. Playback stopped. • `incompatibleSystemAndDecoderRegions`. No discs can be played because the system region does not match the decoder region. • `incompatibleDiscAndDecoderRegions`. The disc cannot be played because the disc is not authored to be played in the decoder's region. • `unexpected`. Something unexpected happened; perhaps content is authored incorrectly. Playback is stopped.
karaokeMode	Occurs when the audio mode is set to `karaoke`.
noFirstPlayChain	Occurs when the DVD disc does not have a FP_PGC (First Play Program Chain) and that the DVD Navigator will not automatically load any PGC and start playback.
parentalLevelChange	Occurs when the parental level of the authored content is about to change. The following additional information is passed to `DVDeventNotification` when this event occurs: • *`eventArg2`*. An integer that indicates the new parental level set in the player.
playbackStopped	Occurs when playback stops. The DVD Navigator has completed playback of the PGC and did not find any other branching instruction for subsequent playback.
playPeriodAutoStop	Occurs when playback stopped as a result of an automatic stop.
rateChange	Occurs when the playback rate changes. The following additional information is passed to `DVDeventNotification` when this event occurs: • *`eventArg2`*. An integer that indicates the new playback rate. A value that is less than (<) 0 indicates reverse playback mode. A value that is greater than (>) 0 indicates forward playback mode. This value is the actual playback rate multiplied by 10,000.
stillOff	Occurs at the end of any still (PGC, Cell, or VOBU).
stillOn	Occurs at the beginning of any still (PGC, Cell, or VOBU). The following additional information is passed to `DVDeventNotification` when this event occurs: • *`eventArg2`* - A boolean that indicates whether buttons are available. Zero (0) indicates buttons are available. One (1) indicates no buttons are available. • *`eventArg3`* - An integer or address that indicates the number of seconds the still will last. `0xFFFFFFFF` indicates an infinite still.

Event	Description
titleChange	Occurs when the current title number changes. The following additional information is passed to `DVDeventNotification` when this event occurs: • *`eventArg2`* - An integer or address that indicates the new title number.
UOPchange	Occurs when one of the available playback or search mechanisms has changed. The following additional information is passed to `DVDeventNotification` when this event occurs: • *`eventArg2`* - An integer or address that indicates which playback or search mechanisms the DVD disc explicitly disabled.
warning	Occurs when a DVD warning condition is encountered. The following additional information is passed to `DVDeventNotification` when this event occurs: • *`eventArg2`* - An integer or address that indicates the warning condition. The warning condition will be one of the following values. • `invalidDVD1_0Disc`. DVD-Video disc is authored incorrectly. Playback can continue, but unexpected behavior might occur. • `formatNotSupported`. A decoder would not support the current format. Playback of a stream might not function. • `illegalNavCommand`. The internal DVD Navigation command processor attempted to process an illegal command. • `open`. • `seek`. • `read`.

See also

DVD

on endSprite

Usage

```
-- Lingo syntax
on endSprite
    statement(s)
end

// JavaScript syntax
function endSprite() {
    statement(s);
}
```

Description

System message and event handler; contains Lingo that runs when the playhead leaves a sprite and goes to a frame in which the sprite doesn't exist. It is generated after `exitFrame`.

Place `on endSprite` handlers in a behavior script.

Director destroys instances of any behavior scripts attached to the sprite immediately after the `endSprite` event occurs.

The event handler is passed the behavior or frame script reference `me` if used in a behavior. This `endSprite` message is sent after the `exitFrame` message if the playhead plays to the end of the frame.

The `go()`, `play()`, and `updateStage()` methods are disabled in an `on endSprite` handler.

Example

This handler runs when the playhead exits a sprite:

```
-- Lingo syntax
on endSprite me
    -- clean up
    gNumberOfSharks = gNumberOfSharks - 1
    sound(5).stop()
end

// JavaScript syntax
function endSprite() {
    // clean up
    gNumberOfSharks--;
    sound(5).stop();
}
```

See also

on beginSprite, on exitFrame

on enterFrame

Usage

```
-- Lingo syntax
on enterFrame
    statement(s)
end

// JavaScript syntax
function enterFrame() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run each time the playhead enters the frame.

Place `on enterFrame` handlers in behavior, frame, or movie scripts, as follows:

- To assign the handler to an individual sprite, put the handler in a behavior attached to the sprite.
- To assign the handler to an individual frame, put the handler in the frame script.
- To assign the handler to every frame (unless you explicitly instruct the movie otherwise), put the `on enterFrame` handler in a movie script. The handler executes every time the playhead enters a frame unless the frame script has its own handler. If the frame script has its own handler, the `on enterFrame` handler in the frame script overrides the `on enterFrame` handler in the movie script.

The order of frame events is `stepFrame`, `prepareFrame`, `enterFrame`, and `exitFrame`.

This event is passed the object reference `me` if used in a behavior.

Example

This handler turns off the puppet condition for sprites 1 through 5 each time the playhead enters the frame:

```
-- Lingo syntax
on enterFrame
    repeat with i = 1 to 5
        _movie.puppetSprite(i, FALSE)
    end repeat
end

// JavaScript syntax
function enterFrame() {
    for (i=1;i<=5;i++) {
        _movie.puppetSprite(i, false);
    }
}
```

on EvalScript

Usage

```
-- Lingo syntax
on EvalScript aParam
    statement(s)
end

// JavaScript syntax
function EvalScript(aParam) {
    statement(s);
}
```

Description

System message and event handler; in a movie with Adobe® Shockwave® content, contains statements that run when the handler receives an `EvalScript` message from a browser. The parameter is a string passed in from the browser.

- The `EvalScript` message can include a string that Director can interpret as a Lingo statement. Lingo cannot accept nested strings. If the handler you are calling expects a string as a parameter, pass the parameter as a symbol.
- The `on EvalScript` handler is called by the `EvalScript()` scripting method from JavaScript or VBScript in a browser.

Include only those behaviors in `on EvalScript` that you want users to control; for security reasons, don't give complete access to behaviors.

Note: *If you place a return at the end of your EvalScript handler, the value returned can be used by JavaScript in the browser.*

Example

This shows how to make the playhead jump to a specific frame depending on what frame is passed in as the parameter:

```
-- Lingo syntax
on EvalScript aParam
    _movie.go(aParam)
end
```

```
// JavaScript syntax
function EvalScript(aParam) {
    _movie.go(aParam);
}
```

This handler runs the statement `_movie.go(aParam)` if it receives an `EvalScript` message that includes dog, cat, or tree as an argument:

```
-- Lingo syntax
on EvalScript aParam
    case aParam of
        "dog", "cat", "tree": _movie.go(aParam)
    end case
end

// JavaScript syntax
function EvalScript(aParam) {
    switch(aParam) {
        case "dog", "cat", "tree": _movie.go(aParam);
    }
}
```

A possible calling statement for this in JavaScript would be `EvalScript ("dog")`.

This handler takes an argument that can be a number or symbol:

```
-- Lingo syntax
on EvalScript aParam
    if word 1 of aParam = "myHandler" then
        _movie.go(aParam)
    end if
end

// JavaScript syntax
function EvalScript(aParam) {
    if (aParam.indexOf("myHandler",0)) {
        _movie.go(aParam);
    }
}
```

The following handler normally requires a string as its argument. The argument is received as a symbol and then converted to a string within the handler by the `string` function:

```
-- Lingo syntax
on myHandler aParam
    _movie.go(string(aParam))
end

// JavaScript syntax
function myHandler(aParam) {
    _movie.go(aParam.toString());
}
```

See also

externalEvent(), return (keyword)

on exitFrame

Usage

```
-- Lingo syntax
on exitFrame
    statement(s)
end

// JavaScript syntax
function exitFrame() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run each time the playhead exits the frame that the `on exitFrame` handler is attached to. The `on exitFrame` handler is a useful place for Lingo that resets conditions that are no longer appropriate after leaving the frame.

Place `on exitFrame` handlers in behavior, frame, or movie scripts, as follows:

- To assign the handler to an individual sprite, put the handler in a behavior attached to the sprite.
- To assign the handler to an individual frame, put the handler in the frame script.
- To assign the handler to every frame unless explicitly instructed otherwise, put the handler in a movie script. The `on exitFrame` handler then executes every time the playhead exits the frame unless the frame script has its own `on exitFrame` handler. When the frame script has its own `on exitFrame` handler, the `on exitFrame` handler in the frame script overrides the one in the movie script.

This event is passed the sprite script or frame script reference `me` if it is used in a behavior. The order of frame events is `prepareFrame`, `enterFrame`, and `exitFrame`.

Example

This handler turns off all puppet conditions when the playhead exits the frame:

```
-- Lingo syntax
on exitFrame me
    repeat with i = 48 down to 1
        sprite(i).scripted = FALSE
    end repeat
end

// JavaScript syntax
function exitFrame() {
    for (i=48; i>=1; i--);
        sprite(i).scripted = false;
    }
}
```

This handler branches the playhead to a specified frame if the value in the global variable `vTotal` exceeds 1000 when the playhead exits the frame:

```
// JavaScript syntax
function exitFrame() {
    if (_global.vTotal > 1000) {
        _movie.go("Finished");
    }
}
```

See also
on enterFrame

on getBehaviorDescription

Usage

```
-- Lingo syntax
on getBehaviorDescription
    statement(s)
end

// JavaScript syntax
function getBehaviorDescription() {
    statement(s);
}
```

Description

System message and event handler; contains Lingo that returns the string that appears in a behavior's description pane in the Behavior Inspector when the behavior is selected.

The description string is optional.

Director sends the `getBehaviorDescription` message to the behaviors attached to a sprite when the Behavior inspector opens. Place the `on getBehaviorDescription` handler within a behavior.

The handler can contain embedded Return characters for formatting multiple-line descriptions.

Example

This statement displays "Vertical Multiline textField Scrollbar" in the description pane:

```
-- Lingo syntax
on getBehaviorDescription
    return "Vertical Multiline textField Scrollbar"
end

// JavaScript syntax
function getBehaviorDescription() {
    return "Vertical Multiline textField Scrollbar";
}
```

See also

on getPropertyDescriptionList, on getBehaviorTooltip, on runPropertyDialog

on getBehaviorTooltip

Usage

```
-- Lingo syntax
on getBehaviorTooltip
    statement(s)
end
```

```
// JavaScript syntax
function getBehaviorTooltip() {
    statement(s);
}
```

Description

System message and event handler; contains Lingo that returns the string that appears in a tooltip for a script in the Library palette.

Director sends the `getBehaviorTooltip` message to the script when the cursor stops over it in the Library palette. Place the `on getBehaviorTooltip` handler within the behavior.

The use of the handler is optional. If no handler is supplied, the cast member name appears in the tooltip.

The handler can contain embedded Return characters for formatting multiple-line descriptions.

Example

This statement displays "Jigsaw puzzle piece" in the description pane:

```
-- Lingo syntax
on getBehaviorTooltip
    return "Jigsaw puzzle piece"
end

// JavaScript syntax
function getBehaviorTooltip() {
    return "Jigsaw puzzle piece";
}
```

See also

on getPropertyDescriptionList, on getBehaviorDescription, on runPropertyDialog

on getPropertyDescriptionList

Usage

```
-- Lingo syntax
on getPropertyDescriptionList
    statement(s)
end

// JavaScript syntax
function getPropertyDescriptionList() {
    statement(s);
}
```

Description

System message and event handler; contains Lingo that generates a list of definitions and labels for the parameters that appear in a behavior's Parameters dialog box.

Place the `on getPropertyDescriptionList` handler within a behavior script. Behaviors that don't contain an `on getPropertyDescriptionList` handler don't appear in the Parameters dialog box and can't be edited from the Director interface.

The `on getPropertyDescriptionList` message is sent when any action that causes the Behavior Inspector to open occurs: either when the user drags a behavior to the Score or the user double-clicks a behavior in the Behavior inspector.

The `#default`, `#format`, and `#comment` settings are mandatory for each parameter. The following are possible values for these settings:

`#default`	The parameter's initial setting.
`#format`	`#integer #float #string #symbol #member #bitmap #filmloop #field #palette #picture #sound #button #shape #movie #digitalvideo #script #richtext #ole #transition #xtra #frame #marker #ink #boolean`
`#comment`	A descriptive string that appears to the left of the parameter's editable field in the Parameters dialog box.
#range	A range of possible values that can be assigned to a property. The range is specified as a linear list with several values or as a minimum and maximum in the form of a property list: `[#min: minValue, #max: maxValue]`.

Example

The following handler defines a behavior's parameters that appear in the Parameters dialog box. Each statement that begins with `addProp` adds a parameter to the list named description. Each element added to the list defines a property and the property's `#default`, `#format`, and `#comment` values:

```
on getPropertyDescriptionList
    description = [:]
    description.addProp(#dynamic, [#default:1, #format:#boolean, #comment:"Dynamic"])
    description.addProp(#fieldNum, [#default:1, #format:#integer,#comment:"Scroll which
sprite:"])
    description.addProp(#extentSprite, [#default:1,#format:#integer,#comment: "Extend
Sprite:"])
    description.addProp(#proportional, [#default:1,#format:#boolean, #comment:
"Proportional:"])
    return description
end
```

See also

addProp, on getBehaviorDescription, on runPropertyDialog

on hyperlinkClicked

Usage

```
-- Lingo syntax
on hyperlinkClicked me, data, range
    statement(s)
end

// JavaScript syntax
function hyperlinkClicked(data, range) {
    statement(s);
}
```

Description

System message and event handler; used to determine when a hyperlink is actually clicked.

This event handler has the following parameters:

- `me` Used in a behavior to identify the sprite instance
- `data` The hyperlink data itself; the string entered in the Text inspector when editing the text cast member
- `range` The character range of the hyperlink in the text (It's possible to get the text of the range itself by using the syntax member Ref.char[range[1]..range[2]]

This handler should be attached to a sprite as a behavior script. Avoid placing this handler in a cast member script.

Example

This behavior shows a link examining the hyperlink that was clicked, jump to a URL if needed, then output the text of the link itself to the message window:

```
property spriteNum
on hyperlinkClicked(me, data, range)
    if data starts "http://" then
        gotoNetPage(data)
    end if
    currentMember = sprite(spriteNum).member
    anchorString = currentMember.char[range[1]..range[2]]
    put("The hyperlink on"&&anchorString&&"was just clicked.")
end
// JavaScript syntax
function hyperlinkClicked(data, range) {
    var st = data.slice(0,7);
    var ht = "http://";
    if (st = ht) {
        gotoNetPage(data);
    }
    var currentMember = sprite(this.spriteNum).member;
    var r1 = currentMember.getPropRef("char", range[1]).hyperlinkRange;
    var a = r1[1] - 1;
    var b = r1[2];
    var st = new String(currentMember.text);
    var anchorString = st.slice(a, b);
    put("The hyperlink on " + anchorString + " was just clicked.");
}
```

on idle

Usage

```
-- Lingo syntax
on idle
    statement(s)
end

// JavaScript syntax
function idle() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run whenever the movie has no other events to handle and is a useful location for Lingo statements that you want to execute as frequently as possible, such as statements that update values in global variables and displays current movie conditions.

Because statements in `on idle` handlers run frequently, it is good practice to avoid placing Lingo that takes a long time to process in an `on idle` handler.

It is often preferable to put `on idle` handlers in frame scripts instead of movie scripts to take advantage of the `on idle` handler only when appropriate.

Director can load cast members from an internal or external cast during an `idle` event. However, it cannot load linked cast members during an `idle` event.

The `idle` message is only sent to frame scripts and movie scripts.

Example

This handler updates the time being displayed in the movie whenever there are no other events to handle:

```
-- Lingo syntax
on idle
    member("Time").text = _system.time()
end idle

// JavaScript syntax
function idle() {
    member("Time").text = _system.time();
}
```

See also

idleHandlerPeriod

on isOKToAttach

Usage

```
-- Lingo syntax
on isOKToAttach me, aSpriteType, aSpriteNum
    statement(s)
end

// JavaScript syntax
function isOKToAttach(aSpriteType, aSpriteNum) {
    statement(s)
}
```

Description

Built-in handler; you can add this handler to a behavior in order to check the type of sprite the behavior is being attached to and prevent the behavior from being attached to inappropriate sprite types.

When the behavior is attached to a sprite, the handler executes and Director passes to it the type of the sprite and its sprite number. The `me` argument contains a reference to the behavior that is being attached to the sprite.

This handler runs before the `on getPropertyDescriptionList` handler.

The Lingo author can check for two types of sprites. `#graphic` includes all graphic cast members, such as shapes, bitmaps, digital video, text, and so on. `#script` indicates the behavior was attached to the script channel. In this case, the *`spriteNum`* is 1.

For each of these sprite types, the handler must return `TRUE` or `FALSE`. A value of `TRUE` indicates that the behavior can be attached to the sprite. A value of `FALSE` prevents the behavior from being attached to the sprite.

If the behavior contains no `on isOKToAttach` handler, then the behavior can be attached to any sprite or frame.

This handler is called during the initial attachment of the behavior to the sprite or script channel and also when attaching a new behavior to a sprite using the Behavior inspector.

Example

This statement checks the sprite type the behavior is being attached to and returns `TRUE` for any graphic sprite except a shape and `FALSE` for the script channel:

```
-- Lingo syntax
on isOKToAttach me, aSpriteType, aSpriteNum
    case aSpriteType of
        #graphic: -- any graphic sprite type
            return sprite(aSpriteNum).member.type <> #shape
            -- works for everything but shape cast members
        #script: --the frame script channel
            return FALSE -- doesn't work as a frame script
    end case
end

// JavaScript syntax
function isOKToAttach(aSpriteType, aSpriteNum) {
    switch (aSpriteType) {
        case symbol("graphic"): // any graphic sprite type
            return sprite(aSpriteNum).member.type != symbol("shape");
            // works for everything but shape cast members
        case symbol("script"): // the frame script channel
            return false; // doesn't work as a frame script
    }
}
```

on keyDown

Usage

```
-- Lingo syntax
on keyDown
    statement(s)
end

// JavaScript syntax
function keyDown() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run when a key is pressed.

When a key is pressed, Director searches these locations, in order, for an `on keyDown` handler: primary event handler, editable field sprite script, field cast member script, frame script, and movie script. For sprites and cast members, `on keyDown` handlers work only for editable text and field members. A `keyDown` event on a different type of cast member, such as a bitmap, has no effect. (If pressing a key should have the same response throughout the movie, set `keyDownScript`.)

Director stops searching when it reaches the first location that has an `on keyDown` handler, unless the handler includes the `pass` command to explicitly pass the `keyDown` message on to the next location.

The `on keyDown` event handler is a good place to put Lingo that implements keyboard shortcuts or other interface features that you want to occur when the user presses keys.

When the movie plays back as an applet, an `on keyDown` handler always traps key presses, even if the handler is empty. If the user is typing in an editable field, an `on keyDown` handler attached to the field must include the `pass` command for the key to appear in the field.

Where you place an `on keyDown` handler can affect when it runs.

- To apply the handler to a specific editable field sprite, put the handler in a sprite script.
- To apply the handler to an editable field cast member in general, put the handler in a cast member script.
- To apply the handler to an entire frame, put the handler in a frame script.
- To apply the handler throughout the entire movie, put the handler in a movie script.

You can override an `on keyDown` handler by placing an alternative `on keyDown` handler in a location that Lingo checks before it gets to the handler you want to override. For example, you can override an `on keyDown` handler assigned to a cast member by placing an `on keyDown` handler in a sprite script.

Example

This handler checks whether the Return key was pressed and if it was, sends the playhead to another frame:

```
-- Lingo syntax
on keyDown
    if (_key.key = RETURN) then _movie.go("AddSum")
end keyDown

// JavaScript syntax
function keyDown() {
    if (_key.keyCode == 36) {
        _movie.go("AddSum");
    }
}
```

See also

charToNum(), keyDownScript, keyUpScript, key, keyCode, keyPressed()

on keyUp

Usage

```
-- Lingo syntax
on keyUp
    statement(s)
end

// JavaScript syntax
function keyUp() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run when a key is released. The `on keyUp` handler is similar to the `on keyDown` handler, except this event occurs after a character appears if a field or text sprite is editable on the screen.

When a key is released, Lingo searches these locations, in order, for an `on keyUp` handler: primary event handler, editable field sprite script, field cast member script, frame script, and movie script. For sprites and cast members, `on keyUp` handlers work only for editable strings. A `keyUp` event on a different type of cast member, such as a bitmap, has no effect. If releasing a key should always have the same response throughout the movie, set `keyUpScript`.

Lingo stops searching when it reaches the first location that has an `on keyUp` handler, unless the handler includes the `pass` command to explicitly pass the `keyUp` message on to the next location.

The `on keyUp` event handler is a good place to put Lingo that implements keyboard shortcuts or other interface features that you want to occur when the user releases keys.

When the movie plays back as an applet, an `on keyUp` handler always traps key presses, even if the handler is empty. If the user is typing in an editable field, an `on keyUp` handler attached to the field must include the `pass` command for the key to appear in the field.

Where you place an `on keyUp` handler can affect when it runs, as follows:

- To apply the handler to a specific editable field sprite, put it in a behavior.
- To apply the handler to an editable field cast member in general, put it in a cast member script.
- To apply the handler to an entire frame, put it in a frame script.
- To apply the handler throughout the entire movie, put it in a movie script.

You can override an `on keyUp` handler by placing an alternative `on keyUp` handler in a location that Lingo checks before it gets to the handler you want to override. For example, you can override an `on keyUp` handler assigned to a cast member by placing an `on keyUp` handler in a sprite script.

Example

This handler checks whether the Return key was released and if it was, sends the playhead to another frame:

```
-- Lingo syntax
on keyUp
    if (_key.key = RETURN) then _movie.go("AddSum")
end keyUp

// JavaScript syntax
function keyUp() {
    if (_key.keyCode == 36) {
        _movie.go("AddSum");
    }
}
```

See also

on keyDown, keyDownScript, keyUpScript

on mouseDown (event handler)

Usage

```
-- Lingo syntax
on mouseDown
    statement(s)
end

// JavaScript syntax
function mouseDown() {
```

```
    statement(s);
}
```

Description

System message and event handler; contains statements that run when the mouse button is pressed.

When the mouse button is pressed, Lingo searches the following locations, in order, for an `on mouseDown` handler: primary event handler, sprite script, cast member script, frame script, and movie script. Lingo stops searching when it reaches the first location that has an `on mouseDown` handler, unless the handler includes the `pass` command to explicitly pass the `mouseDown` message on to the next location.

To have the same response throughout the movie when pressing the mouse button, set `mouseDownScript` or put a mouseDown handler in a Movie script.

The `on mouseDown` event handler is a good place to put Lingo that flashes images, triggers sound effects, or makes sprites move when the user presses the mouse button.

Where you place an `on mouseDown` handler can affect when it runs.

- To apply the handler to a specific sprite, put it in a sprite script.
- To apply the handler to a cast member in general, put it in a cast member script.
- To apply the handler to an entire frame, put it in a frame script.
- To apply the handler throughout the entire movie, put it in a movie script.

You can override an `on mouseDown` handler by placing an alternative `on mouseDown` handler in a location that Lingo checks before it gets to the handler you want to override. For example, you can override an `on mouseDown` handler assigned to a cast member by placing an `on mouseDown` handler in a sprite script.

If used in a behavior, this event is passed the sprite script or frame script reference `me`.

Example

This handler checks whether the user clicks anywhere on the Stage and sends the playhead to another frame if a click occurs:

```
-- Lingo syntax
on mouseDown
    if (_mouse.clickOn = 0) then _movie.go("AddSum")
end

// JavaScript syntax
function mouseDown() {
    if (_mouse.clickOn == 0) {
        _movie.go("AddSum");
    }
}
```

This handler, assigned to a sprite script, plays a sound when the sprite is clicked:

```
-- Lingo syntax
on mouseDown
    sound(1).play(member("Crickets"))
end

// JavaScript syntax
function mouseDown() {
    sound(1).play(member("Crickets"));
}
```

See also
clickOn, mouseDownScript, mouseUpScript

on mouseEnter

Usage

```
-- Lingo syntax
on mouseEnter
    statement(s)
end

// JavaScript syntax
function mouseEnter() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run when the mouse pointer first contacts the active area of the sprite. The mouse button does not have to be pressed.

If the sprite is a bitmap cast member with matte ink applied, the active area is the portion of the image that is displayed; otherwise, the active area is the sprite's bounding rectangle.

If used in a behavior, this event is passed the sprite script or frame script reference `me`.

Example

This example is a simple button behavior that switches the bitmap of the button when the mouse rolls over and then off the button:

```
-- Lingo syntax
property spriteNum

on mouseEnter me
    -- Determine current cast member and switch to next in cast
    currentMember = sprite(spriteNum).member.number
    sprite(spriteNum).member = currentMember + 1
end

on mouseLeave me
    -- Determine current cast member and switch to previous in cast
    currentMember = sprite(spriteNum).member.number
    sprite(spriteNum).member = currentMember - 1
end

// JavaScript syntax
var spriteNum;

function mouseEnter() {
    // Determine current cast member and switch to next in cast
    currentMember = sprite(spriteNum).member.number;
    sprite(spriteNum).member = currentMember + 1;
}

function mouseLeave() {
    // Determine current cast member and switch to previous in cast
    currentMember = sprite(spriteNum).member.number;
```

```
    sprite(spriteNum).member = currentMember - 1;
}
```

See also

on mouseLeave, on mouseWithin

on mouseLeave

Usage

```
-- Lingo syntax
on mouseLeave
    statement(s)
end

// JavaScript syntax
function mouseLeave() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run when the mouse leaves the active area of the sprite. The mouse button does not have to be pressed.

If the sprite is a bitmap cast member with the matte ink applied, the active area is the portion of the image that is displayed; otherwise, the active area is the sprite's bounding rectangle.

If used in a behavior, this event is passed the sprite script or frame script reference `me`.

Example

This statement shows a simple button behavior that switches the bitmap of the button when the mouse pointer rolls over and then back off the button:

```
-- Lingo syntax
property spriteNum

on mouseEnter me
    -- Determine current cast member and switch to next in cast
    currentMember = sprite(spriteNum).member.number
    sprite(spriteNum).member = currentMember + 1
end

on mouseLeave me
    -- Determine current cast member and switch to previous in cast
    currentMember = sprite(spriteNum).member.number
    sprite(spriteNum).member = currentMember - 1
end

// JavaScript syntax
var spriteNum;

function mouseEnter() {
    // Determine current cast member and switch to next in cast
    currentMember = sprite(spriteNum).member.number;
    sprite(spriteNum).member = currentMember + 1;
}
```

```
function mouseLeave() {
    // Determine current cast member and switch to previous in cast
    currentMember = sprite(spriteNum).member.number;
    sprite(spriteNum).member = currentMember - 1;
}
```

See also

on mouseEnter, on mouseWithin

on mouseUp (event handler)

Usage

```
-- Lingo syntax
on mouseUp
    statement(s)
end

// JavaScript syntax
function mouseUp() {
    statement(s);
}
```

Description

System message and event handler; contains statements that are activated when the mouse button is released.

When the mouse button is released, Lingo searches the following locations, in order, for an `on mouseUp` handler: primary event handler, sprite script, cast member script, frame script, and movie script. Lingo stops searching when it reaches the first location that has an `on mouseUp` handler, unless the handler includes the `pass` command to explicitly pass the `mouseUp` message on to the next location.

To create the same response throughout the movie when the user releases the mouse button, set `the mouseUpScript`.

An `on mouseUp` event handler is a good place to put Lingo that changes the appearance of objects—such as buttons—after they are clicked. You can do this by switching the cast member assigned to the sprite after the sprite is clicked and the mouse button is released.

Where you place an `on mouseUp` handler can affect when it runs, as follows:

- To apply the handler to a specific sprite, put it in a sprite script.
- To apply the handler to a cast member in general, put it in a cast member script.
- To apply the handler to an entire frame, put it in a frame script.
- To apply the handler throughout the entire movie, put it in a movie script.

You can override an `on mouseUp` handler by placing an alternative `on mouseUp` handler in a location that Lingo checks before it gets to the handler you want to override. For example, you can override an `on mouseUp` handler assigned to a cast member by placing an `on mouseUp` handler in a sprite script.

If used in a behavior, this event is passed the sprite script or frame script reference `me`.

Example

This handler, assigned to sprite 10, switches the cast member assigned to sprite 10 when the user releases the mouse button after clicking the sprite:

```
-- Lingo syntax
on mouseUp
    sprite(10).member = member("Dimmed")
end

// JavaScript syntax
function mouseUp() {
    sprite(10).member = member("Dimmed");
}
```

See also

on mouseDown (event handler)

on mouseUpOutside

Usage

```
-- Lingo syntax
on mouseUpOutside me
    statement(s)
end

// JavaScript syntax
function mouseUpOutside() {
    statement(s);
}
```

Description

System message and event handler; sent when the user presses the mouse button on a sprite but releases it (away from) the sprite.

Example

This statement plays a sound when the user clicks the mouse button over a sprite and then releases it outside the bounding rectangle of the sprite:

```
-- Lingo syntax
on mouseUpOutside me
    sound(1).play(member("Professor Long Hair"))
end

// JavaScript syntax
function mouseUpOutside() {
    sound(1).play(member("Professor Long Hair"));
}
```

See also

on mouseEnter, on mouseLeave, on mouseWithin

on mouseWithin

Usage

```
-- Lingo syntax
on mouseWithin
    statement(s)
```

```
end

// JavaScript syntax
function mouseWithin() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run when the mouse is within the active area of the sprite. The mouse button does not have to be pressed.

If the sprite is a bitmap cast member with the matte ink applied, the active area is the portion of the image that is displayed; otherwise, the sprite's bounding rectangle is the active area.

If used in a behavior, this event is passed the sprite script or frame script reference `me`.

Example

This statement displays the mouse location when the mouse pointer is over a sprite:

```
-- Lingo syntax
on mouseWithin
    member("Display").text = string(_mouse.mouseH)
end

// JavaScript syntax
function mouseWithin() {
    member("Display").text = _mouse.mouseH.toString();
}
```

See also

`on mouseEnter`, `on mouseLeave`

on moveWindow

Usage

```
-- Lingo syntax
on moveWindow
    statement(s)
end

// JavaScript syntax
function moveWindow() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run when a window is moved, such as by dragging a movie to a new location on the Stage, and is a good place to put Lingo that you want executed every time a movie's window changes location.

Example

This handler displays a message in the Message window when the window a movie is playing in moves:

```
-- Lingo syntax
on moveWindow
```

```
    put("Just moved window containing" && _movie.name)
end

// JavaScript syntax
function moveWindow() {
    put("Just moved window containing " + _movie.name);
}
```

See also

activeWindow, name (3D), windowList

on openWindow

Usage

```
-- Lingo syntax
on openWindow
    statement(s)
end

// JavaScript syntax
function openWindow() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run when Director opens the movie as a movie in a window and is a good place to put Lingo that you want executed every time the movie opens in a window.

Example

This handler plays the sound file Hurray when the window that the movie is playing in opens:

```
-- Lingo syntax
on openWindow
    sound(2).play(member("Hurray"))
end

// JavaScript syntax
function openWindow() {
    sound(2).play(member("Hurray"));
}
```

on prepareFrame

Usage

```
-- Lingo syntax
on prepareFrame
    statement(s)
end

// JavaScript syntax
function prepareFrame {
    statement(s);
}
```

Description

System message and event handler; contains statements that run immediately before the current frame is drawn.

Unlike `beginSprite` and `endSprite` events, a `prepareFrame` event is generated each time the playhead enters a frame.

The `on prepareFrame` handler is a useful place to change sprite properties before the sprite is drawn.

If used in a behavior, the `on prepareFrame` handler receives the reference `me`.

The `go`, `play`, and `updateStage` commands are disabled in an `on prepareFrame` handler.

Example

This handler sets the `locH` property of the sprite that the behavior is attached to:

```
-- Lingo syntax
on prepareFrame me
    sprite(me.spriteNum).locH= _mouse.mouseH
end

// JavaScript syntax
function prepareFrame() {
    sprite(spriteNum).locH= _mouse.mouseH;
}
```

See also

`on enterFrame`

on prepareMovie

Usage

```
-- Lingo syntax
on prepareMovie
    statement(s)
end

// JavaScript syntax
function prepareMovie() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run after the movie preloads cast members but before the movie does the following:

- Creates instances of behaviors attached to sprites in the first frame that plays.
- Prepares the first frame that plays, including drawing the frame, playing any sounds, and executing transitions and palette effects.

New global variables used for sprite behaviors in the first frame should be initialized in the `on prepareMovie` handler. Global variables already set by the previous movie do not need to be reset.

An `on prepareMovie` handler is a good place to put Lingo that creates global variables, initializes variables, plays a sound while the rest of the movie is loading into memory, or checks and adjusts computer conditions such as color depth.

The go, play, and updateStage commands are disabled in an on prepareMovie handler.

Example

This handler creates a global variable when the movie starts:

```
-- Lingo syntax
on prepareMovie
    global currentScore
    currentScore = 0
end

// JavaScript syntax
function prepareMovie() {
    _global.currentScore = 0;
}
```

See also

on enterFrame, on startMovie

on resizeWindow

Usage

```
-- Lingo syntax
on resizeWindow
    statement(s)
end

// JavaScript syntax
function resizeWindow() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run when a movie is running as a movie in a window (MIAW) and the user resizes the window by dragging the window's resize box or one of its edges.

An on resizeWindow event handler is a good place to put Lingo related to the window's dimensions, such as Lingo that positions sprites or crops digital video.

Example

This handler moves sprite 3 to the coordinates stored in the variable centerPlace when the window that the movie is playing in is resized:

```
-- Lingo syntax
on resizeWindow centerPlace
    sprite(3).loc = centerPlace
end

// JavaScript syntax
function resizeWindow(centerPlace) {
    sprite(3).loc = centerPlace;
}
```

See also

drawRect, sourceRect

on rightMouseDown (event handler)

Usage

```
-- Lingo syntax
on rightMousedown
    statement(s)
end

// JavaScript syntax
function rightMouseDown() {
    statement(s);
}
```

Description

System message and event handler; in Windows, specifies statements that run when the right mouse button is pressed. On Mac computers, the statements run when the mouse button and Control key are pressed simultaneously and the `emulateMultiButtonMouse` property is set to `TRUE`; if this property is set to `FALSE`, this event handler has no effect on the Mac.

Example

This handler opens the window Help when the user clicks the right mouse button in Windows:

```
-- Lingo syntax
on rightMousedown
    window("Help").open()
end

// JavaScript syntax
function rightMouseDown() {
    window("Help").open();
}
```

on rightMouseUp (event handler)

Usage

```
-- Lingo syntax
on rightMouseUp
    statement(s)
end

// JavaScript syntax
function rightMouseUp() {
    statement(s);
}
```

Description

System message and event handler; in Windows, specifies statements that run when the right mouse button is released. On Mac computers, the statements run if the mouse button is released while the Control key is pressed and the `emulateMultiButtonMouse` property is set to `TRUE`; if this property is set to `FALSE`, this event handler has no effect on the Mac.

Example

This handler opens the Help window when the user releases the right mouse button in Windows:

```
-- Lingo syntax
on rightMouseUp
    window("Help").open()
end

// JavaScript syntax
function rightMouseUp() {
    window("Help").open();
}
```

on runPropertyDialog

Usage

```
-- Lingo syntax
on runPropertyDialog me, currentInitializerList
    statement(s)
end

// JavaScript syntax
function runPropertyDialog(currentInitializerList) {
    statement(s);
}
```

Description

System message and event handler; contains Lingo that defines specific values for a behavior's parameters in the Parameters dialog box. The `runPropertyDialog` message is sent whenever the behavior is attached to a sprite, or when the user changes the initial property values of a sprite's behavior.

The current settings for a behavior's initial properties are passed to the handler as a property list. If the `on runPropertyDialog` handler is not defined within the behavior, Director runs a behavior customization dialog box based on the property list returned by the `on getPropertyDescriptionList` handler.

Example

The following handler overrides the behavior's values set in the Parameters dialog box for the behavior. New values are contained in the list `currentInitializerList`. Normally, the Parameters dialog box allows the user to set the mass and gravitational constants. However, this handler assigns these parameters constant values without displaying a dialog box:

```
-- Lingo syntax
property mass
property gravitationalConstant

on runPropertyDialog me, currentInitializerList
    --force mass to 10
    currentInitializerList.setaProp(#mass, 10)
    -- force gravitationalConstant to 9.8
    currentInitializerList.setaProp(#gravitationalConstant, 9.8)
    return currentInitializerList
end

// JavaScript syntax
function runPropertyDialog(currentInitializerList) {
    //force mass to 10
    currentInitializerList.setaProp("mass", 10)
    //force gravitationalConstant to 9.8
```

```
    currentInitializerList.setaProp("gravitationalConstant", 9.8)
    return(currentInitializerList)
}
```

See also

on getBehaviorDescription, on getPropertyDescriptionList

on savedLocal

Usage

```
-- Lingo syntax
on savedLocal
    statement(s)
end

// JavaScript syntax
function savedLocal() {
    statement(s);
}
```

Description

System message and event handler; this property is provided to allow for enhancements in future versions of Shockwave Player.

See also

allowSaveLocal

on sendXML

Usage

```
-- Lingo syntax
on sendXML "sendxmlstring", "window", "postdata"
    statement(s)
end

// JavaScript syntax
function sendXML(sendxmlstring, window, postdata) {
    statement(s);
}
```

Description

Event handler; functions much like the `getURL` scripting method, which is also available using the Adobe® Flash® Asset Xtra extension. The `on sendXML` handler is called in Lingo when the *`XMLobject`*`.send` ActionScript method is executed in a Flash sprite or Flash XML object.

In ActionScript, the *`XMLobject`*`.send` method passes two parameters in addition to the XML data in the XML object. These parameters are as follows:

- *`url`* the URL to send the XML data to. Usually this is the URL of a server script that is waiting to process the XML data.
- *`window`* the browser window in which to display the server's response data.

The ActionScript *XMLobject*.send method can be called in Director either by a Flash sprite or by a global Flash XML object created in Lingo. When this happens, the Lingo on sendXML handler is called, and the same parameters are passed to the handler.

The following Lingo illustrates how the parameters are received by the on sendXML handler:

```
on sendXML me, theURL, targetWindow, XMLdata
```

These parameters correlate with the *XMLobject*.send parameters as follows:

- theURL the URL to send the XML data to.
- targetWindow the browser window in which to display the server's response.
- XMLdata the XML data in the Flash XML object.

By creating an on sendXML handler in your Director movie, you enable it to process *XMLobject*.send events generated in a Flash sprite or a global Flash object.

Flash sprites can also load external XML data or parse internal XML data. The Flash Asset Xtra extension handles these functions in the same way as Flash 5 or Flash MX content in your browser.

Example

This Lingo command gets the *XMLobject*.send method information from a Flash sprite and then directs the browser to the URL and transmits the XML data to the URL:

```
-- Lingo syntax
on sendXML me, theURL, targetWindow, xmlData
    gotoNetPage(theURL, targetWindow)
    postNetText(theURL, xmlData)
end

// JavaScript syntax
function sendXML(theURL, targetWindow, xmlData) {
    gotoNetPage(theURL, targetWindow);
    postNetText(theURL, xmlData);
}
```

on startMovie

Usage

```
-- Lingo syntax
on startMovie
    statement(s)
end

// JavaScript syntax
function startMovie() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run just before the playhead enters the first frame of the movie. The startMovie event occurs after the prepareFrame event and before the enterFrame event.

An on startMovie handler is a good place to put Lingo that initializes sprites in the first frame of the movie.

Example

This handler makes sprites invisible when the movie starts:

```
-- Lingo syntax
on startMovie
    repeat with counter = 10 to 50
        sprite(counter).visible = 0
    end repeat
end startMovie

// JavaScript syntax
function startMovie() {
    for(counter=10;counter<=50;counter++) {
        sprite(counter).visible = 0;
    }
}
```

See also

on prepareMovie

on stepFrame

Usage

```
-- Lingo syntax
on stepFrame
    statement(s)
end

// JavaScript syntax
function stepFrame() {
    statement(s);
}
```

Description

System message and event handler; works in script instances in `actorList` because these are the only objects that receive `on stepFrame` messages. This event handler is executed when the playhead enters a frame or the Stage is updated.

An `on stepFrame` handler is a useful location for Lingo that you want to run frequently for a specific set of objects. Assign the objects to `actorList` when you want Lingo in the `on stepFrame` handler to run; remove the objects from `actorList` to prevent Lingo from running. While the objects are in `actorList`, the objects' `on stepFrame` handlers run each time the playhead enters a frame or the `updateStage` command is issued.

The `stepFrame` message is sent before the `prepareFrame` message.

Assign objects to `actorList` so they respond to `stepFrame` messages. Objects must have an `on stepFrame` handler to use this built-in functionality with `actorList`.

The `go`, `play`, and `updateStage` commands are disabled in an `on stepFrame` handler.

Example

If the child object is assigned to `actorList`, the `on stepFrame` handler in this parent script updates the position of the sprite that is stored in the `mySprite` property each time the playhead enters a frame:

```
-- Lingo syntax
```

```
property mySprite

on new me, theSprite
    mySprite = theSprite
    return me
end

on stepFrame me
    sprite(mySprite).loc = point(random(640),random(480))
end

// JavaScript syntax
// define a constructor class that contains the mySprite property
function Frame(theSprite) {
    this.mySprite = theSprite;
}

function stepFrame() {
    var myFrame = new Frame(sprite(spriteName).spriteNum);
    sprite(myFrame.mySprite).loc = point(random(640),random(480));
end
```

on stopMovie

Usage

```
-- Lingo syntax
on stopMovie
    statement(s)
end

// JavaScript syntax
function stopMovie() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run when the movie stops playing.

An `on stopMovie` handler is a good place to put Lingo that performs cleanup tasks—such as closing resource files, clearing global variables, erasing fields, and disposing of objects—when the movie is finished.

An `on stopMovie` handler in a MIAW is called only when the movie plays through to the end or branches to another movie. It isn't called when the window is closed or when the window is deleted by the `forget window` command.

Example

This handler clears a global variable when the movie stops:

```
-- Lingo syntax
global gCurrentScore

on stopMovie
    gCurrentScore = 0
end

// JavaScript syntax
_global.gCurrentScore;
```

```
function stopMovie() {
    _global.gCurrentScore = 0;
}
```

See also

on prepareMovie

on streamStatus

Usage

```
-- Lingo syntax
on streamStatus URL, state, bytesSoFar, bytesTotal, error
    statement(s)
end

// JavaScript syntax
function streamStatus(URL, state, bytesSoFar, bytesTotal, error) {
    statement(s);
}
```

Description

System message and event handler; called periodically to determine how much of an object has been downloaded from the Internet. The handler is called only if `tellStreamStatus (TRUE)` has been called, and the handler has been added to a movie script.

The `on streamStatus` event handler has the following parameters:

URL	Displays the Internet address of the data being retrieved.
state	Displays the state of the stream being downloaded. Possible values are `Connecting`, `Started`, `InProgress`, `Complete`, and `Error`.
bytesSoFar	Displays the number of bytes retrieved from the network so far.
bytesTotal	Displays the total number of bytes in the stream, if known. The value may be 0 if the HTTP server does not include the content length in the MIME header.
error	Displays an empty string ("") if the download has not finished; OK (*OK*) if the download completed successfully; displays an error code if the download was unsuccessful.

These parameters are automatically filled in by Director with information regarding the progress of the download. The handler is called by Director automatically, and there is no way to control when the next call will be. If information regarding a particular operation is needed, call `getStreamStatus()`.

You can initiate network streams using Lingo commands, by linking media from a URL, or by using an external cast member from a URL. A `streamStatus` handler will be called with information about all network streams.

Place the `streamStatus` handler in a movie script.

Example

This handler determines the state of a streamed object and displays the URL of the object:

```
-- Lingo syntax
on streamStatus URL, state, bytesSoFar, bytesTotal
    if state = "Complete" then
        put(URL && "download finished")
```

```
    end if
end streamStatus

// JavaScript syntax
function streamStatus(URL, state, bytesSoFar, bytesTotal) {
    if (state == "Complete") {
        put(URL + " download finished");
    }
}
```

See also

getStreamStatus(), tellStreamStatus()

on timeOut

Usage

```
-- Lingo syntax
on timeOut
    statement(s)
end

// JavaScript syntax
function timeOut() {
    statement(s);
}
```

Description

System message and event handler; contains statements that run when the keyboard or mouse is not used for the time period specified in `timeOutLength`. Always place an `on timeOut` handler in a movie script.

To have a timeout produce the same response throughout a movie, use `the timeoutScript` to centrally control timeout behavior.

Example

The following handler plays the movie Attract Loop after users do nothing for the time set in the `timeoutLength` property. It can be used to respond when users leave the computer.

```
-- Lingo syntax
on timeOut
    _movie.play("Attract Loop")
end timeOut

// JavaScript syntax
function timeOut() {
    _movie.play("Attract Loop");
}
```

trayIconMouseDoubleClick

Usage

```
-- Lingo syntax
on trayIconMouseDoubleClick
    statement(s)
```

```
end

// JavaScript syntax
function trayIconMouseDoubleClick() {
    statement(s);
}
```

Description

Movie and Window event handler (Microsoft Windows only). Contains statements that run when a user double-clicks the system tray icon.

The `trayIconMouseDoubleClick` event is sent to the handler only if the `systemTrayIcon` property is set to `TRUE`.

Example

The following handler pauses a movie when a user double-clicks the system tray icon.

```
-- Lingo syntax
on trayIconMouseDoubleClick
    _movie.delay(500)
end

// JavaScript syntax
function trayIconMouseDoubleClick() {
    _movie.delay(500);
}
```

See also

Movie, systemTrayIcon, trayIconMouseDown, trayIconRightMouseDown, Window

trayIconMouseDown

Usage

```
-- Lingo syntax
on trayIconMouseDown
    statement(s)
end

// JavaScript syntax
function trayIconMouseDown() {
    statement(s);
}
```

Description

Movie and Window event handler (Microsoft Windows only). Contains statements that run when a user single-clicks the system tray icon.

The `trayIconMouseDown` event is sent to the handler only if the `systemTrayIcon` property is set to `TRUE`.

Example

The following handler pauses a movie when a user clicks the mouse when the mouse is over the system tray icon.

```
-- Lingo syntax
on trayIconMouseDown
    _movie.delay(500)
end
```

```
// JavaScript syntax
function trayIconMouseDown() {
    _movie.delay(500);
}
```

See also

Movie, systemTrayIcon, trayIconMouseDoubleClick, trayIconRightMouseDown, Window

trayIconRightMouseDown

Usage

```
-- Lingo syntax
on trayIconRightMouseDown
    statement(s)
end

// JavaScript syntax
function trayIconRightMouseDown() {
    statement(s);
}
```

Description

Movie and Window event handler (Microsoft Windows only). Contains statements that run when a user right-clicks the system tray icon.

The `trayIconRightMouseDown` event is sent to the handler only if the `systemTrayIcon` property is set to `TRUE`.

Example

The following handler pauses a movie when a user right-clicks the system tray icon.

```
-- Lingo syntax
on trayIconRightMouseDown
    _movie.delay(500)
end

// JavaScript syntax
function trayIconRightMouseDown() {
    _movie.delay(500);
}
```

See also

Movie, systemTrayIcon, trayIconMouseDoubleClick, trayIconMouseDown, Window

on zoomWindow

Usage

```
-- Lingo syntax
on zoomWindow
    statement(s)
end

// JavaScript syntax
```

```
function zoomWindow() {
    statement(s);
}
```

Description

System message and event handler; contains statements that execute whenever a movie running as a movie in a window (MIAW) is resized. This happens when the user clicks the Minimize or Maximize button (Windows) or the Zoom button (Mac). The operating system determines the dimensions after resizing the window.

An `on zoomWindow` event handler is a good place to put Lingo that rearranges sprites when window dimensions change.

Example

This handler moves sprite 3 to the coordinates stored in the variable `centerPlace` when the window that the movie is playing in is resized:

```
-- Lingo syntax
on zoomWindow
    centerPlace = point(10, 10)
    sprite(3).loc = centerPlace
end

// JavaScript syntax
function zoomWindow() {
    var centerPlace = point(10, 10);
    sprite(3).loc = centerPlace;
}
```

See also

drawRect, sourceRect, on resizeWindow

Chapter 11: Keywords

This section provides an alphabetical list of all the keywords available in Adobe® Director®.

These keywords apply only to Lingo. JavaScript syntax does contain some keywords and constructs that are similar in function to the following Lingo keywords, but they are not documented here. For more information about JavaScript syntax keywords and constructs, see "Director Scripting Essentials" on page 4.

\ (continuation)

Usage

```
-- Lingo syntax
first part of a statement on this line
second part of the statement
third part of the statement
```

Description

Continuation symbol; when used as the last character in a line, indicates that the statement continues on the next line. Lingo then interprets the lines as one continuous statement.

Example

This statement uses the \ character to wrap the statement on to two lines:

```
-- Lingo syntax
if sprite("mySprite").member = member("myMember") then _player.alert("The sprite was created
from myMember")
```

case

Usage

```
-- Lingo syntax
case expression of
    expression1: Statement
    expression2: Statement(s)
    expression3, expression4: Statement
    {otherwise: Statement(s)}
end case
```

Description

Keyword; starts a multiple branching logic structure that is easier to write than repeated `if...then` statements.

Lingo compares the value in *`case expression`* to the expressions in the lines beneath it, starting at the beginning and continuing through each line in order, until Lingo encounters an expression that matches *`case expression`*.

When Lingo finds a matching expression, it executes the corresponding statement or statements that follow the colon after the matching expression. When only one statement follows the matching expression, the matching expression and its corresponding statement may appear on the same line. Multiple statements must appear on indented lines immediately below the matching expression.

When more than one possible match could cause Lingo to execute the same statements, the expressions must be separated by commas. (The syntax line containing *expression3* and *expresssion4* is an example of such a situation.)

After Lingo encounters the first match, it stops testing for additional matches.

If the optional `otherwise` statement is included at the end of the case structure, the statements following `otherwise` are executed if there are no matches.

Example

The following handler tests which key the user pressed most recently and responds accordingly

- If the user pressed A, the movie goes to the frame labeled Apple.
- If the user pressed B or C, the movie performs the specified transition and then goes to the frame labeled Oranges.
- If the user pressed any other key, the computer beeps.

```
on keyDown
    case (_key.key) of
        "a": _movie.go("Apple")
        "b", "c":
             _movie.puppetTransition(99)
             _movie.go("Oranges")
        otherwise: _sound.beep()
    end case
end keyDown
```

This case statement tests whether the cursor is over sprite 1, 2, or 3 and runs the corresponding Lingo if it is:

```
case _movie.rollOver() of
    1: sound(1).play(member("Horn"))
    2: sound(1).play(member("Drum"))
    3: sound(1).play(member("Bongos"))
end case
```

char...of

Usage

```
-- Lingo syntax
textMemberExpression.char[whichCharacter]
char whichCharacter of fieldOrStringVariable
textMemberExpression.char[firstCharacter..lastCharacter]
char firstCharacter to lastCharacter of fieldOrStringVariable
```

Description

Keyword; identifies a character or a range of characters in a chunk expression. A chunk expression is any character, word, item, or line in any source of text (such as field cast members and variables) that holds a string.

- An expression using *whichCharacter* identifies a specific character.
- An expression using *firstCharacter* and *lastCharacter* identifies a range of characters.

The expressions must be integers that specify a character or range of characters in the chunk. Characters include letters, numbers, punctuation marks, spaces, and control characters such as Tab and Return.

You can test but not set the `char...of` keyword. Use the `put...into` command to modify the characters in a string.

Example

This statement displays the first character of the string $9.00:

```
//Lingo
put(("$9.00").char[1..1])
-- "$"

// Javascript
trace(("$9.00").substring(0,1))
-- "$"
```

This statement displays the entire string $9.00:

```
//Lingo
put(("$9.00").char[1..5])
-- "$9.00"

// Javascript
trace(("$9.00").substring(0))
-- "$9.00"
```

This statement changes the first five characters of the second word in the third line of a text cast member:

```
//Lingo
member("quiz").line[3].word[2].char[1..5] = "?????"

// Javascript
var s = member(1).getPropRef("line",3).getProp("word",2)
s=s.replace(s.substring(0),"????")
member(1).getPropRef("line",3).setProp("word",2,s)
```

See also

mouseMember, mouseItem, mouseLine, mouseWord

end

Usage

```
-- Lingo syntax
end
```

Description

Keyword; marks the end of handlers and multiple-line control structures.

Example

The following mouseDown handler ends with an end mouseDown statement.

```
on mouseDown
     _player.alert("The mouse was pressed")
end mouseDown
```

end case

Usage

```
-- Lingo syntax
end case
```

Description

Keyword; ends a `case` statement.

Example

This handler uses the end case keyword to end the case statement:

```
on keyDown
    case (_key.key) of
        "a": _movie.go("Apple")
        "b", "c":
             _movie.puppetTransition(99)
             _movie.go("Oranges")
        otherwise: _sound.beep()
    end case
end keyDown
```

See also

case

exit

Usage

```
-- Lingo syntax
exit
```

Description

Keyword; instructs Lingo to leave a handler and return to where the handler was called. If the handler is nested within another handler, Lingo returns to the main handler.

Example

The first statement of this script checks whether the monitor is set to black and white and then exits if it is:

```
on setColors
    if _system.colorDepth = 1 then exit
    sprite(1).foreColor = 35
end
```

See also

abort, halt(), quit(), pass, return (keyword)

exit repeat

Usage

```
-- Lingo syntax
exit repeat
```

Description

Keyword; instructs Lingo to leave a repeat loop and go to the statement following the `end repeat` statement but to remain within the current handler or method.

The `exit repeat` keyword is useful for breaking out of a repeat loop when a specified condition—such as two values being equal or a variable being a certain value—exists.

Example

The following handler searches for the position of the first vowel in a string represented by the variable testString. As soon as the first vowel is found, the exit repeat command instructs Lingo to leave the repeat loop and go to the statement return i:

```
on findVowel testString
    repeat with i = 1 to testString.char[testString.char.count]
        if "aeiou" contains testString.char[i] then exit repeat
    end repeat
    return i
end
```

See also
repeat while, repeat with

field

Usage

```
field(whichField)
```

Description

Keyword; refers to the field cast member specified by *whichField*.

- When *whichField* is a string, it is used as the cast member name.
- When *whichField* is an integer, it is used as the cast member number.

Character strings and chunk expressions can be read from or placed in the field.

The term `field` was used in earlier versions of Director and is maintained for backward compatibility. For new movies, use `member` to refer to field cast members.

Example

This statement places the characters 5 through 10 of the field name entry in the variable `myKeyword`:

```
myKeyword = field("entry").char[5..10]
```

This statement checks whether the user entered the word *desk* and, if so, goes to the frame `deskBid`:

```
if member("bid") contains "desk" then _movie.go("deskBid")
```

See also
char...of, item...of, line...of, word...of

global

Usage

```
global variable1 {, variable2} {, variable3}...
```

Description

Keyword; defines a variable as a global variable so that other handlers or movies can share it.

Every handler that examines or changes the content of a global variable must use the `global` keyword to identify the variable as global. Otherwise, the handler treats the variable as a local variable, even if it is declared to be global in another handler.

Note: *To ensure that global variables are available throughout a movie, declare and initialize them in the* `prepareMovie` *handler. Then, if you leave and return to the movie from another movie, your global variables will be reset to the initial values unless you first check to see that they aren't already set.*

A global variable can be declared in any handler or script. Its value can be used by any other handlers or scripts that also declare the variable as global. If the script changes the variable's value, the new value is available to every other handler that treats the variable as global.

A global variable is available in any script or movie, regardless of where it is first declared; it is not automatically cleared when you navigate to another frame, movie, or window.

Any variables manipulated in the Message window are automatically global, even though they are not explicitly declared as such.

Movies with Shockwave® content playing on the Internet cannot access global variables within other movies, even movies playing on the same HTML page. The only way movies can share global variables is if an embedded movie navigates to another movie and replaces itself through either `goToNetMovie` or `go movie`.

Example

The following example sets the global variable StartingPoint to an initial value of 1 if it doesn't already contain a value. This allows navigation to and from the movie without loss of stored data.

```
//Lingo
global gStartingPoint

on prepareMovie
    if voidP(gStartingPoint) then gStartingPoint = 1
end

// Javascript
function prepareMovie(){
    if (_global.gStartingPoint==null) then _global.gStartingPoint = 1
}
```

See also

`showGlobals()`, `property`, `gotoNetMovie`

if

Usage

```
if logicalExpression then statement
if logicalExpression then statement
else statement
end if
if logicalExpression then
        statement(s)
end if
if logicalExpression then
        statement(s)
else
        statement(s)
```

```
end if
if logicalExpression1 then
        statement(s)
else if logicalExpression2 then
        statement(s)
else if logicalExpression3 then
        statement(s)
end if
if logicalExpression1 then
        statement(s)
else logicalExpression2
end if
```

Description

Keyword; `if...then` structure that evaluates the logical expression specified by *`logicalExpression`*.

- If the condition is `TRUE`, Lingo executes the statement(s) that follow `then`.
- If the condition is `FALSE`, Lingo executes the statement(s) following `else`. If no statements follow `else`, Lingo exits the `if...then` structure.
- All parts of the condition must be evaluated; execution does not stop at the first condition that is met or not met. Thus, faster code may be created by nesting `if...then` statements on separate lines instead of placing them all on the first line to be evaluated.

When the condition is a property, Lingo automatically checks whether the property is `TRUE`. You don't need to explicitly add the phrase `= TRUE` after the property.

The `else` portion of the statement is optional. To use more than one *`then-statement`* or *`else-statement`*, you must end with the form `end if`.

The `else` portion always corresponds to the previous `if` statement; thus, sometimes you must include an `else nothing` statement to associate an `else` keyword with the proper `if` keyword.

Note: *A quick way to determine in the script window if a script is paired properly is to press Tab. This forces Director to check the open Script window and show the indentation for the contents. Any mismatches will be immediately apparent.*

Example

This statement checks whether the carriage return was pressed and then continues if it was:

```
if the key = RETURN then go the frame + 1
```

This handler checks whether the Command and Q keys were pressed simultaneously and, if so, executes the subsequent statements:

```
on keyDown
    if (_key.commandDown) and (_key.key = "q") then
        cleanUp
        quit
    end if
end keyDown
```

Compare the following two constructions and the performance results. The first construction evaluates both conditions, and so must determine the time measurement, which may take a while. The second construction evaluates the first condition; the second condition is checked only if the first condition is TRUE.

```
spriteUnderCursor = rollOver()
if (spriteUnderCursor > 25) and MeasureTimeSinceIStarted() then
    _player.alert("You found the hidden treasure!")
end if
```

The alternate, and faster, construction would be as follows:

```
spriteUnderCursor = rollOver()
if (spriteUnderCursor > 25) then
    if MeasureTimeSinceIStarted() then
        _player.alert("You found the hidden treasure!")
    end if
end if
```

See also

case

INF

Usage

```
-- Lingo syntax
INF
```

Description

Return value; indicates that a specified Lingo expression evaluates as an infinite number.

See also

NAN

item...of

Usage

```
-- Lingo syntax
textMemberExpression.item[whichItem]
item whichItem of fieldOrStringVariable
textMemberExpression.item[firstItem..lastItem]
item firstItem to lastItem of fieldOrStringVariable
```

Description

Keyword; specifies an item or range of items in a chunk expression. An item in this case is any sequence of characters delimited by the current delimiter as determined by the `itemDelimiter` property.

The terms *`whichItem`*, *`firstItem`*, and *`lastItem`* must be integers or integer expressions that refer to the position of items in the chunk.

Chunk expressions refer to any character, word, item, or line in any source of strings. Sources of strings include field and text cast members and variables that hold strings.

When the number that specifies the last item is greater than the item's position in the chunk expression, the actual last item is specified instead.

Example

This statement looks for the third item in the chunk expression that consists of names of colors and then displays the result in the Message window:

```
put("red,yellow,blue green,orange".item[3])
-- "blue green"
```

The result is the entire chunk "blue green" because this is the entire chunk between the commas.

The following statement looks for the third through fifth items in the chunk expression. Because there are only four items in the chunk expression, only the third item is used and fourth items are returned. The result appears in the Message window.

```
put("red,yellow,blue green,orange".item[3..5])
-- "blue green, orange"
put item 5 of "red, yellow, blue green, orange"
-- ""
```

The following statement inserts the item Desk as the fourth item in the second line of the field cast member All Bids:

```
member("All Bids").line[2].item[4] = "Desk"
```

See also
char...of, itemDelimiter, number of members, word...of

line...of

Usage

```
-- Lingo syntax
textMemberExpression.line[whichLine]
line whichLine of fieldOrStringVariable
textMemberExpression.line[firstLine..lastLine]
line firstLine to lastLine of fieldOrStringVariable
```

Description

Keyword; specifies a line or a range of lines in a chunk expression. A line chunk is any sequence of characters delimited by carriage returns, not by line breaks caused by text wrapping.

The expressions *whichLine*, *firstLine*, and *lastLine* must be integers that specify a line in the chunk.

Chunk expressions refer to any character, word, item, or line in any source of characters. Sources of characters include field cast members and variables that hold strings.

Example

This statement assigns the first four lines of the variable Action to the field cast member To Do:

```
member("To Do").text = Action.line[1..4]
```

This statement inserts the word and after the second word of the third line of the string assigned to the variable Notes:

```
put "and" after Notes.line[3].word[2]
```

See also
char...of, item...of, word...of, number of members

loop (keyword)

Usage

```
-- Lingo syntax
_movie.goLoop()
```

Description

Keyword; refers to the marker.

Example

This handler loops the movie between the previous marker and the current frame:

```
on exitFrame
    _movie.goLoop()
end exitFrame
```

me

Usage

```
-- Lingo syntax
me
```

Description

Special variable; used within parent scripts and behaviors to refer to the current object that is an instance of the parent script or the behavior or a variable that contains the memory address of the object.

The term has no predefined meaning in Lingo. The term `me` is used by convention.

To see an example of `me` used in a completed movie, see the Parent Scripts movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

The following statement sets the object `myBird1` to the script named Bird. The `me` keyword accepts the parameter script Bird and is used to return that parameter.

```
myBird1 = new script("Bird")
```

This is the `on new` handler of the Bird script:

```
on new me
    return me
end
```

The following two sets of handlers make up a parent script. The first set uses `me` to refer to the child object. The second set uses the variable `myAddress` to refer to the child object. In all other respects, the parent scripts are the same.

This is the first set:

```
property myData

on new me, theData
    myData = theData
    return me
end
```

```
on stepFrame me
    ProcessData me
end
```

This is the second set:

```
property myData

on new myAddress, theData
    myData = theData
    return myAddress
end

on stepFrame myAddress
    ProcessData myAddress
end
```

See also

new(), ancestor

menu

Usage

```
-- Lingo syntax
menu: menuName
itemName | script
itemName | script
...
```

or

```
menu: menuName
itemName | script
itemName | script
...
[more menus]
```

Description

Keyword; in conjunction with the `installMenu` command, specifies the actual content of custom menus. Field cast members contain menu definitions; refer to them by the cast member name or number.

The `menu` keyword is followed immediately by a colon, a space, and the name of the menu. In subsequent lines, specify the menu items for that menu. You can set a script to execute when the user chooses an item by placing the script after the vertical bar symbol (|). A new menu is defined by the subsequent occurrence of the `menu` keyword.

Note: *Menus are not available in Shockwave Player.*

On the Mac, you can use special characters to define custom menus. These special characters are case-sensitive. For example, to make a menu item bold, the letter *B* must be uppercase.

Special symbols should follow the item name and precede the vertical bar symbol (|). You can also use more than one special character to define a menu item. Using <B<U, for example, sets the style to Bold and Underline.

Avoid special character formatting for cross-platform movies because not all Windows® computers support it.

Symbol	Example	Description
@	menu: @	*On the Mac®, creates the Apple® symbol and enables Mac menu bar items when you define an Apple menu.
!Ã	!ÃEasy Select	*On the Mac, checks the menu with a check mark (Option+v).
<B	Bold<B	*On the Mac, sets the menu item's style to Bold.
<I	Italic<I	*On the Mac, sets the style to Italic.
<U	Underline<U	*On the Mac, sets the style to Underline.
<O	Outline<O	*On the Mac, sets the style to Outline.
<S	Shadow<S	*On the Mac, sets the style to Shadow.
|	Open/O | go to frame "Open"	Associates a script with the menu item.
/	Quit/Q	Defines a command-key equivalent.
(	Save(	Disables the menu item.
(-	(-	Creates a disabled line in the menu.

*. Identifies formatting tags that work only on the Mac.

Example

This example is the text of a field cast member named CustomMenu2 which can be used to specify the content of a custom File menu. To install this menu, use "`installMenu member("CustomMenu2")`" while the movie is running. The Convert menu item runs the custom handler `convertThis`.

```
menu: File
Open/O | _movie.go("Open")
Close/W | _movie.go("Close")
Convert/C | convertThis
    (-
Quit/Q | _movie.go("Quit")
```

See also

installMenu, name, number (menu items), checkMark, enabled, script

NAN

Usage

```
-- Lingo syntax
NAN
```

Description

Return value; Indicates that a specified Lingo expression is not a number.

This statement attempts to display the square root of -1, which is not a number, in the Message window:

```
-- Lingo syntax
put((-1).sqrt) -- NAN
```

See also
INF

next

Usage

```
-- Lingo syntax
next
```

Description

Keyword; refers to the next marker in the movie and is equivalent to the phrase `the marker (+ 1)`.

Example

This statement sends the playhead to the next marker in the movie:

```
go next
```

This handler moves the movie to the next marker in the Score when the right arrow key is pressed and to the previous marker when the left arrow key is pressed:

```
on keyUp
    if (_key.keyCode = 124) then _movie.goNext()
    if (_key.keyCode = 123) then _movie.goPrevious()
end keyUp
```

See also
loop (keyword), goPrevious()

next repeat

Usage

```
-- Lingo syntax
next repeat
```

Description

Keyword; sends Lingo to the next step in a repeat loop in a script. This function differs from that of the `exit repeat` keyword.

Exampleg

This repeat loop displays only odd numbers in the Message window:

```
repeat with i = 1 to 10
    if (i mod 2) = 0 then next repeat
    put(i)
end repeat
```

on

Usage

```
-- Lingo syntax
on handlerName {argument1}, {arg2}, {arg3} ...
    statement(s)
end handlerName
```

Description

Keyword; indicates the beginning of a handler, a collection of Lingo statements that you can execute by using the handler name. A handler can accept arguments as input values and returns a value as a function result.

Handlers can be defined in behaviors, movie scripts, and cast member scripts. A handler in a cast member script can be called only by other handlers in the same script. A handler in a movie script can be called from anywhere.

You can use the same handler in more than one movie by putting the handler's script in a shared cast.

otherwise

Usage

```
-- Lingo syntax
otherwise statement(s)
```

Description

Keyword; precedes instructions that Lingo performs when none of the earlier conditions in a `case` statement are met.

This keyword can be used to alert users of out-of-bound input or invalid type, and can be very helpful in debugging during development.

Example

The following handler tests which key the user pressed most recently and responds accordingly:

- If the user pressed A, B, or C, the movie performs the corresponding action following the of keyword.
- If the user pressed any other key, the movie executes the statement that follows the otherwise keyword. In this case, the statement is a simple alert.

```
//Lingo
on keyDown
     case (_key.key) of
        "a": _movie.go("Apple")
        "b", "c":
             _movie.puppetTransition(99)
             _movie.go("Oranges")
        otherwise: _player.alert("That is not a valid key.")
    end case
end keyDown

// Javascript
function keyDown()
{
    switch(_key.key)
    {
        case "a":
```

```
            _movie.go("Apple");
            break;
            case "b":
            case "c":
            _movie.puppetTransition(99);
            _movie.go("Oranges");
            break;
            default:
            _player.alert("That is not a valid key.");
        }
    }
```

property

Usage

```
-- Lingo syntax
property {property1}{, property2} {,property3} {...}
```

Description

Keyword; declares the properties specified by *property1*, *property2*, and so on as property variables.

Declare property variables at the beginning of the parent script or behavior script. You can access them from outside the parent script or behavior script by using the `the` operator.

Note: *The `spriteNum` property is available to all behaviors and simply needs to be declared to be accessed.*

You can refer to a property within a parent script or behavior script without using the `me` keyword. However, to refer to a property of a parent script's ancestor, use the form `me.property`.

For behaviors, properties defined in one behavior script are available to other behaviors attached to the same sprite.

You can directly manipulate a child object's property from outside the object's parent scripts through syntax similar to that for manipulating other properties. For example, this statement sets the `motionStyle` property of a child object:

```
set the motionStyle of myBouncingObject to #frenetic
```

Use the `count` function to determine the number of properties within the parent script of a child object. Retrieve the name of these properties by using `getPropAt`. Add properties to an object by using `setaProp()`.

To see an example of `property` used in a completed movie, see the Parent Scripts movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement lets each child object created from a single parent script have its own location and velocity setting:

```
property location, velocity
```

This parent script handler declares pMySpriteNum a property to make it available:

```
-- script Elder
property pMyChannel
on new me, whichSprite
    me.pMyChannel = whichSprite
    return me
end
```

The original behavior script sets up the ancestor and passes the `spriteNum` property to all behaviors:

```
property spriteNum

property ancestor

on beginSprite me
    ancestor = new script("Elder", spriteNum)
end
```

See also
end, ancestor, spriteNum

put...after

Usage

```
-- Lingo syntax
put expression after chunkExpression
```

Description

Command; evaluates a Lingo expression, converts the value to a string, and inserts the resulting string after a specified chunk in a container, without replacing the container's contents. (If *chunkExpression* specifies a nonexistent target chunk, the string value is inserted as appropriate into the container.)

Chunk expressions refer to any character, word, item, or line in any container. Containers include field cast members; text cast members; variables that hold strings; and specified characters, words, items, lines, and ranges within containers.

Example

This statement adds the string "fox dog cat" after the contents of the field cast member Animal List:

```
put("fox dog cat") after member("Animal List")
```

The same can be accomplished using this statement:

```
put "fox dog cat" after member("Animal List").line[1]
```

See also
char...of, item...of, line...of, paragraph, word...of, put...before, put...into

put...before

Usage

```
-- Lingo syntax
put expression before chunkExpression
```

Description

Command; evaluates a Lingo expression, converts the value to a string, and inserts the resulting string before a specified chunk in a container, without replacing the container's contents. (If *chunkExpression* specifies a nonexistent target chunk, the string value is inserted as appropriate into the container.)

Chunk expressions refer to any character, word, item, or line in any container. Containers include field cast members; text cast members; variables that hold strings; and specified characters, words, items, lines, and ranges in containers.

Example

This statement sets the variable animalList to the string "fox dog cat" and then inserts the word *elk* before the second word of the list:

```
put "fox dog cat" into animalList
put "elk " before word 2 of animalList
```

The result is the string "fox elk dog cat".

The same can be accomplished using this syntax:

```
put "fox dog cat" into animalList
put "elk " before animalList.word[2]
```

See also

char...of, item...of, line...of, paragraph, word...of, put...after, put...into

put...into

Usage

```
-- Lingo syntax
put expression into chunkExpression
```

Description

Command; evaluates a Lingo expression, converts the value to a string, and uses the resulting string to replace a specified chunk in a container. (If *chunkExpression* specifies a nonexistent target chunk, the string value is inserted as appropriate into the container.)

Chunk expressions refer to any character, word, item, or line in any container. Containers include field cast members; text cast members; variables that hold strings; and specified characters, words, items, lines, and ranges in containers.

When a movie plays back as an applet, the `put...into` command replaces all text within a container, not chunks of text.

To assign values to variables, use the `set` command.

Example

This statement changes the second line of the field cast member Review Comments to "Reviewed by Agnes Gooch":

```
put "Reviewed by Agnes Gooch" into line 2 of member("Review Comments")
```

The same can be accomplished with a text cast member using this syntax:

```
put "Reviewed by Agnes Gooch" into member("Review Comments").line[2]

// Javascript
member("Review Comments").setProp("line",2,"Reviewed by Agnes Gooch")
repeat while:
```

See also

char...of, item...of, line...of, paragraph, word...of, put...before, put...after, set...to, set...=

repeat while

Usage

```
-- Lingo syntax
repeat while testCondition
    statement(s)
end repeat
```

Description

Keyword; repeatedly executes *statement(s)* so long as the condition specified by *testCondition* is TRUE. This structure can be used in Lingo that continues to read strings until the end of a file is reached, checks items until the end of a list is reached, or repeatedly performs an action until the user presses or releases the mouse button.

While in a repeat loop, Lingo ignores other events. To check the current key in a repeat loop, use the keyPressed property.

Only one handler can run at a time. If Lingo stays in a repeat loop for a long time, other events stack up waiting to be evaluated. Therefore, repeat loops are best used for short, fast operations or when users are idle.

If you need to process something for several seconds or more, evaluate the function in a loop with some type of counter or test to track progress.

If the stop condition is never reached or there is no exit from the repeat loop, you can force Director to stop by using Control+Alt+period (Windows) or Command+period (Mac).

Example

This handler starts the timer counting, resets the timer to 0, and then has the timer count up to 60 milliseconds:

```
on countTime
    _system.milliseconds
    repeat while _system.milliseconds < 60
    -- waiting for time
    end repeat
end countTime

// Javascript
function countTime()
{
    _system.milliseconds
    while (_system.milliseconds < 60)
    {
        -- waiting for time
    }
}
```

See also

exit, exit repeat, repeat with, keyPressed()

repeat with

Usage

```
-- Lingo syntax
repeat with counter = start to finish
    statement(s)
```

```
end repeat
```

Description

Keyword; executes the Lingo specified by *statement(s)* the number of times specified by *counter*. The value of *counter* is the difference between the value specified by *start* and the value specified by *finish*. The counter is incremented by 1 each time Lingo cycles through the repeat loop.

The `repeat with` structure is useful for repeatedly applying the same effect to a series of sprites or for calculating a series of numbers to some exponent.

While in a repeat loop, Lingo ignores other events. To check the current key in a repeat loop, use the `keyPressed` property.

Only one handler can run at a time. If Lingo stays in a repeat loop for a long time, other events stack up waiting to be evaluated. Therefore, repeat loops are best used for short, fast operations or when users are idle.

If you need to process something for several seconds or more, evaluate the function in a loop with some type of counter or test to track progress.

If the stop condition is never reached or there is no exit from the repeat loop, you can force Director to stop by using Control+Alt+period (Windows) or Command+period (Mac).

Example

This handler turns sprites 1 through 30 into puppets:

```
//Lingo
on puppetize
    repeat with channel = 1 to 30
    _movie.puppetSprite(channel, TRUE)
    end repeat
end puppetize

// Javascript
function puppetize()
{
    for(var channel=1;channel<30;channel++)
    {
    _movie.puppetSprite(channel, true);
    }
}
```

See also

exit, exit repeat, repeat while, repeat with...down to, repeat with...in list

repeat with...down to

Usage

```
-- Lingo syntax
repeat with variable = startValue down to endValue
```

Description

Keyword; counts down by increments of 1 from *startValue* to *endValue*.

Only one handler can run at a time. If Lingo stays in a repeat loop for a long time, other events stack up waiting to be evaluated. Therefore, repeat loops are best used for short, fast operations or when you know the user won't be doing other things.

While in a repeat loop, Lingo ignores other events. To check the current key in a repeat loop, use the `keyPressed` property.

If you need to process something for several seconds or more, evaluate the function in a loop with some type of counter or test to track progress.

If the stop condition is never reached or there is no exit from the repeat loop, you can force Director to stop by using Control+Alt+period (Windows) or Command+period (Mac).

Example

This handler contains a repeat loop that counts down from 20 to 15:

```
on countDown
    repeat with i = 20 down to 15
        sprite(6).member = 10 + i
        _movie.updateStage()
    end repeat
end
```

repeat with...in list

Usage

```
-- Lingo syntax
repeat with variable in someList
```

Description

Keyword; assigns successive values from the specified list to the variable.

While in a repeat loop, Lingo ignores other events except keypresses. To check the current key in a repeat loop, use the `keyPressed` property.

Only one handler can run at a time. If Lingo stays in a repeat loop for a long time, other events stack up waiting to be evaluated. Therefore, repeat loops are best used for short, fast operations or when users are idle.

If you need to process something for several seconds or more, evaluate the function in a loop with some type of counter or test to track progress.

If the stop condition is never reached or there is no exit from the repeat loop, you can force Director to stop by using Control+Alt+period (Windows) or Command+period (Mac).

Example

This statement displays four values in the Message window:

```
//Lingo
repeat with i in [1, 2, 3, 4]
    put(i)
end repeat

// Javascript
var nl = new Array()
nl =[1,2,3,4]
```

```
var i
for(i in nl)
{
    trace(nl[i])
}
```

return (keyword)

Usage

```
-- Lingo syntax
return expression
```

Description

Keyword; returns the value of *expression* and exits from the handler. The *expression* argument can be any Lingo value.

When calling a handler that serves as a user-defined function and has a return value, you must use parentheses around the argument lists, even if there are no arguments, as in the `diceRoll` function handler discussed under the entry for the `result` function.

The function of the `return` keyword is similar to that of the `exit` command, except that `return` also returns a value to whatever called the handler. The `return` command in a handler immediately exits from that handler, but it can return a value to the Lingo that called it.

The use of `return` in object-oriented scripting can be difficult to understand. It's easier to start by using `return` to create functions and exit handlers. Later, you will see that the `return me` line in an `on new` handler gives you a way to pass back a reference to an object that was created so it can be assigned to a variable name.

The `return` keyword isn't the same as the character constant `RETURN`, which indicates a carriage return. The function depends on the context.

To retrieve a returned value, use parentheses after the handler name in the calling statement to indicate that the named handler is a function.

To see an example of `return (keyword)` used in a completed movie, see the Parent Scripts movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This handler returns a random multiple of 5 between 5 and 100:

```
//Lingo
on getRandomScore
    theScore = 5 * random(20)
    return theScore
end getRandomScore

// Javascript
function getRandomScore()
{
    theScore = 5 * random(20);
    return theScore;
}
```

Call this handler with a statement similar to the following:

```
thisScore = getRandomScore()
```

In this example, the variable `thisScore` is assigned the return value from the function `getRandomScore()`. A parent script performs the same function: by returning the object reference, the variable name in the calling code provides a handle for subsequent references to that object.

See also

`result`, `RETURN (constant)`

set...to, set...=

Usage

```
-- Lingo syntax
lingoProperty = expression
variable = expression
```

Description

Command; evaluates an expression and puts the result in the property specified by *`lingoProperty`* or the variable specified by *`variable`*.

Example

This statement sets the name of member 3 to Sunset:

```
member(3).name = "Sunset"
```

The following statement sets the soundEnabled property to the opposite of its current state. When `soundEnabled` is `TRUE` (the sound is on), this statement turns it off. When `soundEnabled` is `FALSE` (the sound is off), this statement turns it on.

```
_sound.soundEnabled = not(_sound.soundEnabled)
```

This statement sets the variable vowels to the string "aeiou":

```
vowels = "aeiou"
```

See also
property

sprite...intersects

Usage

```
-- Lingo syntax
sprite(sprite1).intersects(sprite2)
sprite sprite1 intersects sprite2
```

Description

Keyword; operator that compares the position of two sprites to determine whether the quad of *`sprite1`* touches (`TRUE`) or does not touch (`FALSE`) the quad of *`sprite2`*.

If both sprites have matte ink, their actual outlines, not the quads, are used. A sprite's outline is defined by the nonwhite pixels that make up its border.

This is a comparison operator with a precedence level of 5.

Note: *The dot operator is required whenever sprite1 is not a simple expression—that is, one that contains a math operation.*

Example

This statement checks whether two sprites intersect and, if they do, changes the contents of the field cast member Notice to "You placed it correctly.":

```
// Lingo Syntax
if sprite i intersects j then put("You placed it correctly.") into member("Notice")

// Javascript
if (sprite(i).intersects(sprite(j)))
{
    member(“Notice”).text=”You placed it correctly”;
}
```

See also

sprite...within, quad

sprite...within

Usage

```
-- Lingo syntax
sprite(sprite1).within(sprite2)
sprite sprite1 within sprite2
```

Description

Keyword; operator that compares the position of two sprites and determines whether the quad of *sprite1* is entirely inside the quad of *sprite2* (TRUE) or not (FALSE).

If both sprites have matte ink, their actual outlines, not the quads, are used. A sprite's outline is defined by the nonwhite pixels that make up its border.

This is a comparison operator with a precedence level of 5.

Note: *The dot operator is required whenever sprite1 is not a simple expression—that is, one that contains a math operation.*

Example

This statement checks whether two sprites intersect and calls the handler doInside if they do:

```
//Lingo
if sprite(3).within(2) then doInside

// Javascript
if (sprite(3).within(2))
{
 doInside();
}
```

See also

sprite...intersects, quad

version

Usage

```
-- Lingo syntax
_player.productVersion
```

Description

Keyword; system variable that contains the version string for Director. The same string appears in the Mac Finder's Info window.

Example

This statement displays the version of Director in the Message window:

```
put(_player.productVersion)
```

word...of

Usage

```
-- Lingo syntax
member(whichCastMember).word[whichWord]
textMemberExpression.word[whichWord]
chunkExpression.word[whichWord]
word whichWord of fieldOrStringVariable
fieldOrStringVariable. word[whichWord]
textMemberExpression.word[firstWord..lastWord]
member(whichCastMember).word[firstWord..lastWord]
word firstWord to lastWord of chunkExpression
chunkExpression.word[whichWord..lastWord]
```

Description

Chunk expression; specifies a word or a range of words in a chunk expression. A word chunk is any sequence of characters delimited by spaces. (Any non-visible character, such as a tab or carriage return, is considered a space.)

The expressions *whichWord*, *firstWord*, and *lastWord* must evaluate to integers that specify a word in the chunk.

Chunk expressions refer to any character, word, item, or line in any source of characters. Sources of characters include field and text cast members and variables that hold strings.

To see an example of `word...of` used in a completed movie, see the Text movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

These statements set the variable named animalList to the string "fox dog cat" and then insert the word elk before the second word of the list:

```
animalList = "fox dog cat"
put "elk" before animalList.word[2]
```

The result is the string "fox elk dog cat".

This statement tells Director to display the fifth word of the same string in the Message window:

```
put "fox elk dog cat".word[5]
```

Because there is no fifth word in this string, the Message window displays two quotation marks (""), which indicate an empty string.

See also

`char...of`, `line...of`, `item...of`, `count()`, `number (words)`

Chapter 12: Methods

This section provides an alphabetical list of all the methods available in Director®.

abort

Usage

```
--Lingo syntax
abort

// JavaScript syntax
abort();
```

Description

Command; tells Lingo to exit the current handler and any handler that called it without executing any of the remaining statements in the handler. This differs from the `exit` keyword, which returns to the handler from which the current handler was called.

The `abort` command does not quit Director.

Parameters

None.

Example

This statement instructs Lingo to exit the handler and any handler that called it when the amount of free memory is less than 50K:

```
-- Lingo syntax
if the freeBytes < 50*1024 then abort

// JavaScript syntax
if (_player.freeBytes < 50*1024) {
    abort()
}
```

See also

exit, halt(), quit()

abs()

Usage

```
--Lingo syntax
abs (numericExpression)

// JavaScript syntax
Math.abs (numericExpression)
```

Description

The `abs()` function has several uses. It can simplify the tracking of mouse and sprite movement by converting coordinate differences (which can be either positive or negative numbers) into distances (which are always positive numbers). The `abs()` function is also useful for handling mathematical functions, such as `sqrt()` and `log()`.

In JavaScript syntax, use the Math object's `abs()` function.

Parameters

`numericExpression` Required. An integer or floating-point number from which an absolute value is calculated. If *`numericExpression`* is an integer, the absolute value is also an integer. If *`numericExpression`* is a floating-point number, the absolute value is also a floating-point number.

Example

This statement determines whether the absolute value of the difference between the current mouse position and the value of the variable `startV` is greater than 30 (since you wouldn't want to use a negative number for distance). If it is, the foreground color of sprite 6 is changed.

```
-- Lingo syntax
if (the mouseV - startV).abs > 30 then sprite(6).forecolor = 95

// JavaScript syntax
if ((_mouse.mouseV - Math.abs(_mouse.startV)) > 30) {
    sprite(6).foreColor = 95;
}
```

activateAtLoc()

Usage

```
-- Lingo syntax
dvdObjRef.activateAtLoc(point(x, y))

// JavaScript syntax
dvdObjRef.activateAtLoc(point(x, y));
```

Description

DVD method; activates the hilite of the embedded DVD menu item that is under a specified Stage location.

This method returns 0 if successful.

Parameters

`point(`*`x, y`*`)` Required. A point in Stage coordinates that specifies the location of the embedded DVD menu item.

Example

This statement activates the hilite of the menu item at a specified Stage location:

```
-- Lingo syntax
member("movie1").activateAtLoc(point(100, 200))

// JavaScript syntax
member("movie1").activateAtLoc(point(100, 200));
```

See also

DVD

activateButton()

Usage

```
-- Lingo syntax
dvdObjRef.activateButton()

// JavaScript syntax
dvdObjRef.activateButton();
```

Description

DVD method; activates the currently selected menu button.

This method returns 0 if successful.

Note: *This method is not supported in Mac®-Intel®.*

Parameters

None.

Example

This statement activates the menu button on a specified cast member:

```
-- Lingo syntax
sprite(1).member.activateButton()

// JavaScript syntax
sprite(1).member.activateButton();
```

See also

DVD

add

Usage

```
-- Lingo syntax
linearList.add(value)

// JavaScript syntax
array.push(value)
```

Description

List command; for linear lists only, adds a value to a linear list. For a sorted list, the value is placed in its proper order. For an unsorted list, the value is added to the end of the list.

This command returns an error when used on a property list.

Note: *Don't confuse the* `add` *command with the* `+` *operator used for addition or the* `&` *operator used to concatenate strings.*

Parameters

`value` Required. A value to add to the linear list.

Example

These statements add the value 2 to the list named bids. The resulting list is [3, 4, 1, 2].

```
-- Lingo syntax
bids = [3, 4, 1]
bids.add(2)

// JavaScript syntax
bids = new Array(3,4,1);
bids.push(2);
```

This statement adds 2 to the sorted linear list [1, 4, 5]. The new item remains in alphanumeric order because the list is sorted.

```
-- Lingo syntax
bids.add(2)

// JavaScript syntax
bids.push(2);
// to sort the list using JavaScript syntax
bids.sort();
```

See also

sort

add (3D texture)

Usage

```
--Lingo syntax
member(whichCastmember).model(whichModel).meshdeform.mesh[index].textureLayer.add()

// JavaScript syntax
member(whichCastmember).model(whichModel).meshdeform.mesh[index].textureLayer.add()
```

Description

3D `meshdeform` modifier command; adds an empty texture layer to the model's mesh.

You can copy texture coordinates between layers using the following code:

```
modelReference.meshdeform.texturelayer[a].texturecoordinatelist =
modelReference.meshdeform.texturelayer[b].texturecoordinatelist
```

Parameters

None.

Example

This statement creates a new texture layer for the first mesh of the model named Ear.

```
--Lingo syntax
member("Scene").model("Ear").meshdeform.mesh[1].textureLayer.add()

// JavaScript syntax
member("Scene").getprop("model","Ear").meshdeform.mesh[1].textureLayer.add();
```

See also

meshDeform (modifier), textureLayer, textureCoordinateList

addAt

Usage

```
list.AddAt(position, value)
```

Description

List command; for linear lists only, adds a value at a specified position in the list.

This command returns an error when used with a property list.

Parameters

position Required. An integer that specifies the position in the list to which the value specified by *value* is added.

value Required. A value to add to the list.

Example

This statement adds the value 8 to the fourth position in the list named bids, which is [3, 2, 4, 5, 6, 7]:

```
--Lingo
bids = [3, 2, 4, 5, 6, 7]
bids.addAt(4,8)

// Javascript
bids = list(3, 2, 4, 5, 6, 7)
bids.addAt(4,8)
```

The resulting value of bids is [3, 2, 4, 8, 5, 6, 7].

addBackdrop

Usage

```
-- Lingo syntax
sprite(whichSprite).camera{(index)}.addBackdrop(texture, locWithinSprite, rotation)
member(whichCastmember).camera(whichCamera).addBackdrop(texture, locWithinSprite,
rotation)

// JavaScript syntax
sprite(whichSprite).camera{(index)}.addBackdrop(texture, locWithinSprite, rotation);
member(whichCastmember).camera(whichCamera).addBackdrop(texture, locWithinSprite,
rotation);
```

Description

3D camera command; adds a backdrop to the end of the camera's list of backdrops.

Parameters

texture Required. The texture to apply to the backdrop.

locWithinSprite Required. A 2D loc at which the backdrop is displayed in the 3D sprite. This location is measured from the upper left corner of the sprite.

rotation Required. An integer that specifies the number of degrees to rotate the texture.

Example

The first line of this statement creates a texture named Rough from the cast member named Cedar and stores it in the variable `t1`. The second line applies the texture as a backdrop at the point (220, 220) within sprite 5. The texture has a rotation of 0 degrees. The last line applies the same texture as a backdrop for camera 1 of the cast member named Scene at the point (20, 20) with a rotation of 45 degrees.

```
t1 = member("Scene").newTexture("Rough", #fromCastMember,member("Cedar"))
sprite(5).camera.addBackdrop(t1, point(220, 220), 0)
member("Scene").camera[1].addBackdrop(t1, point(20, 20), 45)

// Javascript
var t1 = member("Scene").newTexture("Rough", symbol("fromCastMember"),member("Cedar"))
sprite(5).camera.addBackdrop(t1, point(220, 220), 0) ;
member("Scene").getPropRef("camera",1).addBackdrop(t1, point(20, 20), 45) ;
```

See also

removeBackdrop

addCamera

Usage

```
-- Lingo syntax
sprite(whichSprite).addCamera(whichCamera, index)
-- JavaScript syntax
sprite(whichSprite).addCamera(whichCamera, index);
```

Description

3D command; adds a camera to the list of cameras for the sprite. The view from each camera is displayed on top of the view from cameras with lower *index* positions. You can set the `rect` property of each camera to display multiple views within the sprite.

Parameters

whichCamera Required. A reference to the camera to add to the list of cameras for the sprite.

index Required. An integer that specifies the index in the list of cameras at which *whichCamera* is added. If *index* is greater than the value of `cameraCount()`, the camera is added to the end of the list.

Example

This statement inserts the camera named FlightCam at the fifth index position of the list of cameras of sprite 12:

```
--Lingo syntax
sprite(12).addCamera(member("scene").camera("FlightCam"), 5)

// JavaScript syntax
sprite(12).addCamera(member("scene").getPropRef("camera", i), 5);
// where i is the number index for the camera "FlightCam".
```

See also

cameraCount(), deleteCamera

addChild

Usage

```
-- Lingo syntax
member(whichCastmember).node(whichParentNode).addChild(member(whichCastmember).node(whichC
hildNode) {,#preserveWorld})

// JavaScript syntax
member(whichCastmember).node(whichParentNode).addChild(member(whichCastmember).node(whichC
hildNode) {,symbol(preserveWorld)})
```

Description

3D command; adds a node to the list of children of another node, and removes it from the list of children of its former parent.

An equivalent to this method would be to set the `parent` property of the child node to the parent node.

Parameters

`addMemberRef` Required. A reference to the cast member that contains the node to add.

`addNodeRef` Required. A reference to the node to add. This node can be a model, group, camera, or light.

`symPreserveParentOrWorld` Optional. A reference to the camera to add to the list of cameras for the sprite. Valid values are `#preserveWorld` or `#preserveParent`. When the child is added with `#preserveParent` specified, the parent-relative transform of the child remains unchanged and the child jumps to that transform in the space of its new parent. The child's world transform is recalculated. When the child is added with `#preserveWorld` specified, the world transform of the child remains unchanged and the child does not jump to its transform in the space of its new parent. Its parent-relative transform is recalculated.

Example

This statement adds the model named Tire to the list of children of the model named Car.

```
-- Lingo syntax
member("3D").model("Car").addChild(member("3D").model("Tire"))

// JavaScript syntax
member("3D").getProp("model" , i ).addChild(member("3D").getProp("model" , j));
// where i is the number index for model "Car" and j is the number index for model "Tire".
```

This statement adds the model named Bird to the list of children of the camera named MyCamera and uses the #preserveWorld argument to maintain Bird's world position.

```
-- Lingo syntax
member("3D").camera("MyCamera").addChild(member("3D").model
("Bird"), #preserveWorld)

// JavaScript syntax
member("3D").getPropRef("camera",j).addChild(member("3D").getProp("model",i),symbol("prese
rveWorld"))
// where i the number index of the model "Bird" and j is the number index of the camera
"MyCamera"
```

See also

parent, addToWorld, removeFromWorld

addModifier

Usage

```
-- Lingo syntax
member(whichCastmember).model(whichModel).addModifier(#modifierType)

// JavaScript syntax
member(whichCastmember).model(whichModel).addModifier(symbol(modifierType));
```

Description

3D model command; adds a specified modifier to the model. There is no default value for this command.

Parameters

symbolModType Required. A symbol that specifies the modifier to add. Possible modifiers are as follows:

- `#bonesPlayer`
- `#collision`
- `#inker`
- `#keyframePlayer`
- `#lod` (level of detail)
- `#meshDeform`
- `#sds`
- `#toon`

For more detailed information about each modifier, see the individual modifier entries.

Example

This statement adds the toon modifier to the model named Box.

```
-- Lingo syntax
member("shapes").model("Box").addModifier(#toon)

// JavaScript syntax
member("shapes").getPropRef("model" , a ).addModifier(symbol("toon"));
// where a is the number index for the "Box" model.
```

See also

bonesPlayer (modifier), collision (modifier), inker (modifier), keyframePlayer (modifier), lod (modifier), meshDeform (modifier), sds (modifier), toon (modifier), getRendererServices(), removeModifier, modifier, modifier[], modifiers

addOverlay

Usage

```
-- Lingo syntax
sprite(whichSprite).camera{(index)}.addOverlay(texture, locWithinSprite, rotation)
member(whichCastmember).camera(whichCamera).addOverlay(texture, locWithinSprite, rotation)

// JavaScript syntax
sprite(whichSprite).camera{(index)}.addOverlay(texture, locWithinSprite, rotation)
```

```
member(whichCastmember).camera(whichCamera).addOverlay(texture, locWithinSprite, rotation)
```

Description

3D camera command; adds an overlay to the end of a camera's list of overlays.

Parameters

texture Required. The texture to apply to the overlay.

locWithinSprite Required. A 2D loc at which the overlay is displayed in the 3D sprite. This location is measured from the upper left corner of the sprite.

rotation Required. An integer that specifies the number of degrees to rotate the texture.

Example

The first line of this statement creates a texture named Rough from the cast member named Cedar and stores it in the variable `t1`. The second line applies the texture as an overlay at the point (220, 220) within sprite 5. The texture has a rotation of 0 degrees. The last line of the statement applies the same texture as an overlay for camera 1 of the cast member named Scene at the point (20, 20). The texture has a rotation of 45 degrees.

```
-- Lingo syntax
t1 = member("Scene").newTexture("Rough", #fromCastMember,member("Cedar"))
sprite(5).camera.addOverlay(t1, point(220, 220), 0)
member("Scene").camera[1].addOverlay(t1, point(20, 20), 45)

// JavaScript syntax
t1 = member("Scene").newTexture("Rough", symbol("fromCastMember"),member("Cedar"));
sprite(5).camera.addOverlay(t1, point(220, 220), 0);
member("Scene").getPropRef("camera",1).addOverlay(t1, point(20, 20), 45);
```

See also

removeOverlay

addProp

Usage

```
list.addProp(property, value)
addProp list, property, value
```

Description

Property list command; for property lists only, adds a specified property and its value to a property list.

For an unsorted list, the value is added to the end of the list. For a sorted list, the value is placed in its proper order.

If the property already exists in the list, both Lingo and JavaScript syntax create a duplicate property. You can avoid duplicate properties by using the `setaProp()` command to change the new entry's property.

This command returns an error when used with a linear list.

Parameters

property Required. The property to add to the list.

value Required. The value of the property to add to the list.

Example

This statement adds the property named `kayne` and its assigned value 3 to the property list named `bids`, which contains [#gee: 4, #ohasi: 1]. Because the list is sorted, the new entry is placed in alphabetical order:

```
--Lingo
bids.addProp(#kayne, 3)

// Javascript
bids.addProp("kayne",3)
```

The result is the list [#gee: 4, #kayne: 3, #ohasi: 1].

This statement adds the entry `kayne`: 7 to the list named `bids`, which now contains [#gee: 4, #kayne: 3, #ohasi: 1]. Because the list already contains the property `kayne`, Lingo creates a duplicate property:

```
--Lingo
bids.addProp(#kayne, 7)

// Javascript
bids.addProp("kayne",7)
```

The result is the list [#gee: 4, #kayne: 3, #kayne: 7, #ohasi: 1].

addToWorld

Usage

```
-- Lingo syntax
member(whichCastmember).model(whichModel).addToWorld()
member(whichCastmember).group(whichGroup).addToWorld()
member(whichCastmember).camera(whichCamera).addToWorld()
member(whichCastmember).light(whichLight).addToWorld()

// JavaScript syntax
member(whichCastmember).model(whichModel).addToWorld()
member(whichCastmember).group(whichGroup).addToWorld()
member(whichCastmember).camera(whichCamera).addToWorld()
memberwhichCastmember).light(whichLight).addToWorld()
```

Description

3D command; inserts the model, group, camera, or light into the 3D world of the cast member as a child of the group named World.

When a model, group, camera, or light is created or cloned, it is automatically added to the world. Use the `removeFromWorld` command to take a model, group, camera, or light out of the 3D world without deleting it. Use the `isInWorld()` command to test whether a model, group, camera, or light has been added or removed from the world.

Parameters

None.

Example

This statement adds the model named gbCyl to the 3D world of the cast member named Scene.

```
-- Lingo syntax
member("Scene").model("gbCyl").addToWorld()
```

```
// JavaScript syntax
member("Scene").getProp("model","gbCyl").addToWorld();
```

See also
isInWorld(), removeFromWorld

addVertex()

Usage

```
-- Lingo syntax
memberObjRef.addVertex(indexToAddAt, pointToAddVertex {,[ horizControlLocV, \
vertControlLocV ], [ horizControlLocH, vertControlLocV ]})

// JavaScript syntax
memberObjRef.addVertex(indexToAddAt, pointToAddVertex {,[ horizControlLocV, vertControlLocV
], [ horizControlLocH, vertControlLocV ]});
```

Description

Vector shape command; adds a new vertex to a vector shape cast member in the position specified.

The horizontal and vertical positions are relative to the origin of the vertex shape cast member.

When using the final two optional parameters, you can specify the location of the control handles for the vertex. The control handle location is offset relative to the vertex, so if no location is specified, it will be located at 0 horizontal offset and 0 vertical offset.

Parameters

indexToAddAt Required. An integer that specifies the index at which the member is added.

pointToAddVertex Required. A point that specifies the position at which the member is added.

horizControlLocH Optional. An integer that specifies the location of the horizontal portion of the horizontal control handle.

horizControlLocV Optional. An integer that specifies the location of the vertical portion of the horizontal control handle.

vertControlLocH Optional. An integer that specifies the location of the horizontal portion of the vertical control handle.

vertControlLocV Optional. An integer that specifies the location of the vertical portion of the vertical control handle.

Example

This line adds a vertex point in the vector shape Archie between the two existing vertex points, at the position 25 horizontal and 15 vertical:

```
-- Lingo syntax
member("Archie").addVertex(2, point(25, 15))

// JavaScript syntax
member("Archie").addVertex(2, point(25, 15));
```

See also
vertexList, moveVertex(), deleteVertex(), originMode

alert()

Usage

```
-- Lingo syntax
_player.alert(displayString)

// JavaScript syntax
_player.alert(displayString);
```

Description

Player method; causes a system beep and displays an alert dialog box containing a specified string.

The alert message must be a string. If you want to include a number variable in an alert, convert the variable to a string before passing it to `alert()`.

Parameters

`displayString` Required. A string that represents the text displayed in the alert dialog box. The string can contain up to 255 characters.

Example

The following statement produces an alert stating that there is no CD-ROM drive connected:

```
-- Lingo syntax
_player.alert("There is no CD-ROM drive connected.")

// JavaScript syntax
_player.alert("There is no CD-ROM drive connected.");
```

This statement produces an alert stating that a file was not found:

```
-- Lingo syntax
_player.alert("The file" && QUOTE & filename & QUOTE && "was not found.")

// JavaScript syntax
_player.alert("The file \"" + filename + "\" was not found.");
```

See also
Player

Alert()

Usage

```
Alert( MUIObject,alertPropertiesList)
```

Description

This command displays an alert dialog box created from an instance of the MUI Xtra. This feature is in addition to the simple alerts generated by the alert command.

The MUI Xtra provides modal alerts. The alert can be moveable or non-moveable. To create the alert, create a MUI Xtra object, and then issue the alert command with a list containing definitions of alert properties as the second parameter.

The following are properties that you must specify and their possible values:

Property	Possible values	Specifies
#buttons	#Ok #OkCancel #AbortRetryIgnore #YesNoCancel #YesNo #RetryCancel	The set of buttons that appear in the alert. The buttons appear in the order that they are named in each symbol.
#default	The ordinal number of the button that becomes the default. For example, if the alert's buttons are OK and Cancel, 2 specifies the Cancel button. Specify 0 for no default.	Which button is the default.
#icon	#stop #note #caution #question #error	The type of icon that appears in the alert. Specify 0 for no icon.
#message	A string	The message that appears in the alert
#movable	TRUE FALSE	Whether the alert is moveable
#title	A string	The alert's title

You must explicitly specify each of the alert's properties. The MUI Xtra doesn't provide a list of default alert properties. Lingo returns a value for the button that the user clicks.

Alerts can be almost as big as the screen; you can display a lengthy description if appropriate.

The following statements create and display an alert dialog box.

- The first statement creates an instance of the MUI Xtra, which is the object used as the dialog box.
- The second statement sets up a list of the alert's properties.
- The final statements use the Alert command to display the alert and report which buttons the user clicks.

Example

```
-- Lingo syntax
set alertObj = new(xtra "MUI")
set alertInitList = [ #buttons : #YesNo, ¬
#title : "Alert Test", #message : "This shows Yes and No buttons",\
#movable : TRUE]
if objectP ( alertObj ) then
       set result = Alert( alertObj, alertInitList )
       case result of
       1 : -- the user clicked yes
       2 : -- the user clicked no
       otherwise : -- something is seriously wrong
   end case
end if
```

append

Usage

```
list.append(value)
append list, value
```

Description

List command; for linear lists only, adds the specified value to the end of a linear list. This differs from the `add` command, which adds a value to a sorted list according to the list's order.

This command returns a script error when used with a property list.

Properties

value Required. The value to add to the end of the linear list.

Example

This statement adds the value 2 at the end of the sorted list named bids, which contains [1, 3, 4], even though this placement does not match the list's sorted order:

```
--Lingo
set bids = [1, 3, 4]
bids.append(2)

// Javascript
bids = list(1, 3, 4)
bids.append(2)
```

The resulting value of bids is [1, 3, 4, 2].

See also

add (3D texture), sort

appMinimize()

Usage

```
-- Lingo syntax
_player.appMinimize()

// JavaScript syntax
_player.appMinimize();
```

Description

Player method; in Microsoft Windows, causes a projector to minimize to the Windows Task Bar. On the Mac, causes a projector to be hidden.

On the Mac, reopen a hidden projector from the Mac application menu.

This method is useful for projectors and MIAWs that play back without a title bar.

Parameters

None.

Example

```
--Lingo syntax
on mouseUp me
    _player.appMinimize()
end


// JavaScript syntax
function mouseUp() {
     _player.appMinimize();
}
```

See also
Player

atan()

Usage

```
-- Lingo syntax
(number).atan
atan (number)

// JavaScript syntax
Math.atan(number);
```

Description

Math function (Lingo only); calculates the arctangent, which is the angle whose tangent is a specified number. The result is a value in radians between pi/2 and +pi/2.

In JavaScript syntax, use the Math object's `atan()` function.

Parameters

None.

Example

This statement displays the arctangent of 1:

```
(1).atan
```

The result, to four decimal places, is 0.7854, or approximately pi/4.

Most trigonometric functions use radians, so you may want to convert from degrees to radians.

This handler lets you convert between degrees and radians:

```
-- Lingo syntax
on DegreesToRads degreeValue
    return degreeValue * PI/180
end

// JavaScript syntax
function DegreesToRads(degreeValue) {
    return degreeValue * PI/180
}
```

The handler displays the conversion of 30 degrees to radians in the Message window:

```
put DegreesToRads(30)
-- 0.5236
```

See also
cos(), PI, sin()

beep()

Usage

```
-- Lingo syntax
_sound.beep({intBeepCount})

// JavaScript syntax
_sound.beep({intBeepCount});
```

Description

Sound method; causes the computer's speaker to beep the number of times specified by *intBeepCount*. If *intBeepCount* is missing, the beep occurs once.

- In Windows, the beep is the sound assigned in the Sounds Properties dialog box.
- For the Mac, the beep is the sound selected from Alert Sounds on the Sound control panel. If the volume on the Sound control panel is set to 0, the menu bar flashes instead.

Parameters

intBeepCount Optional. An integer that specifies the number of times the computer's speakers should beep.

Example

```
-- Lingo syntax
on mouseUp me
    _sound.beep(1)
end mouseUp
// JavaScript syntax
function mouseUp() {
    _sound.beep(1);
}
```

See also

Sound

beginRecording()

Usage

```
-- Lingo syntax
_movie.beginRecording()

// JavaScript syntax
_movie.beginRecording();
```

Description

Movie method; starts a Score generation session.

When you call `beginRecording()`, the playhead automatically advances one frame and begins recording in that frame. To avoid this behavior and begin recording in the frame in which `beginRecording()` is called, place a statement such as `_movie.go(_movie.frame - 1)` between the calls to `beginRecording()` and `endRecording()`.

Only one update session in a movie can be active at a time.

Every call to `beginRecording()` must be matched by a call to `endRecording()`, which ends the Score generation session.

Parameters

None.

Example

When used in the following handler, the `beginRecording` keyword begins a Score generation session that animates the cast member Ball by assigning the cast member to sprite channel 20 and then moving the sprite horizontally and vertically over a series of frames. The number of frames is determined by the argument `numberOfFrames`.

```
-- Lingo syntax
on animBall(numberOfFrames)
    _movie.beginRecording()
    horizontal = 0
    vertical = 100
    repeat with i = 1 to numberOfFrames
        _movie.go(i)
        sprite(20).member = member("Ball")
        sprite(20).locH = horizontal
        sprite(20).locV = vertical
        sprite(20).foreColor = 255
        horizontal = horizontal + 3
        vertical = vertical + 2
        _movie.updateFrame()
    end repeat
    _movie.endRecording()
end animBall

// JavaScript syntax
function animBall(numberOfFrames) {
    _movie.beginRecording();
    var horizontal = 0;
    var vertical = 100;
    for (var i = 1; i <= numberOfFrames; i++) {
        _movie.go(1);
        sprite(20).member = member("Ball")
        sprite(20).locH = horizontal;
        sprite(20).locV = vertical;
        sprite(20).foreColor = 255;
        horizontal = horizontal + 3;
        vertical = vertical + 2;
        _movie.updateFrame();
    }
    _movie.endRecording();
}
```

See also

endRecording(), Movie

bitAnd()

Usage

```
bitAnd(integer1, integer2)
```

Description

Function (Lingo only); converts the two specified integers to 32-bit binary numbers and returns a binary number whose digits are 1's in the positions where both numbers had a 1, and 0's in every other position. The result is the new binary number, which Lingo displays as a base 10 integer.

Integer	**Binary number (abbreviated)**
6	00110
7	00111
Result	
6	00110

In JavaScript syntax, use the bitwise operator "&".

Parameters

integer1 Required. The first integer.

integer2 Required. The second integer.

Example

This statement compares the binary versions of the integers 6 and 7 and returns the result as an integer:

```
--Lingo
put bitAnd(6, 7)
-- 6



// Javascript
trace ( 6 & 7)
// 6
```

See also

bitNot(), bitOr(), bitXor()

bitNot()

Usage

```
(integer).bitNot
bitNot(integer)
```

Description

Function (Lingo only); converts the specified integer to a 32-bit binary number and reverses the value of each binary digit, replacing 1's with 0's and 0's with 1's. The result is the new binary number, which Lingo displays as a base 10 integer.

Integer	Binary number
1	00000000000000000000000000000001
Result	
-2	11111111111111111111111111111110

In JavaScript syntax, use the bitwise operator "~".

Parameters

None.

Example

This statement inverts the binary representation of the integer 1 and returns a new number.

```
--Lingo
put (1).bitNot
-- -2

// Javascript
trace(~1)
// -2
```

See also

bitAnd(), bitOr(), bitXor()

bitOr()

Usage

```
bitOr(integer1, integer2)
```

Description

Function (Lingo only); converts the two specified integers to 32-bit binary numbers and returns a binary number whose digits are 1's in the positions where either number had a 1, and 0's in every other position. The result is the new binary number, which Lingo displays as a base 10 integer.

Integer	Binary number (abbreviated)
5	0101
6	0110
Result	
7	0111

In JavaScript syntax, use the bitwise operator "|".

Parameters

integer1 Required. The first integer.

integer2 Required. The second integer.

Example

This statement compares the 32-bit binary versions of 5 and 6 and returns the result as an integer:

```
-- Lingo
put bitOr(5, 6)
-- 7

// Javascript
trace(5|6)
// 7
```

See also

bitNot(), bitAnd(), bitXor()

bitXor()

Usage

```
bitXor(integer1, integer2)
```

Description

Function; converts the two specified integers to 32-bit binary numbers and returns a binary number whose digits are 1's in the positions where the given numbers' digits do not match, and 0's in the positions where the digits are the same. The result is the new binary number, which Lingo displays as a base 10 integer.

Integer	**Binary number (abbreviated)**
5	0101
6	0110
Result	
3	0011

In JavaScript syntax, use the bitwise operator "^".

Parameters

integer1 Required. The first integer.

integer2 Required. The second integer.

Example

This statement compares the 32-bit binary versions of 5 and 6 and returns the result as an integer:

```
-- Lingo
put bitXor(5, 6)
-- 3

// Javascript
trace(5^6)
// 3
```

See also

bitNot(), bitOr(), bitAnd()

breakLoop()

Usage

```
-- Lingo syntax
soundChannelObjRef.breakLoop()

// JavaScript syntax
soundChannelObjRef.breakLoop();
```

Description

Sound Channel method; causes the currently looping sound in channel *soundChannelObjRef* to stop looping and play through to its `endTime`.

If there is no current loop, this method has no effect.

Parameters

None.

Example

This handler causes the background music looping in sound channel 2 to stop looping and play through to its end:

```
-- Lingo syntax
on continueBackgroundMusic
    sound(2).breakLoop()
end

// JavaScript syntax
function continueBackgroundMusic() {
    sound(2).breakLoop();
}
```

See also

endTime, Sound Channel

browserName()

Usage

```
browserName pathName
browserName()
browserName(#enabled, trueOrFalse)
```

Description

System property, command, and function; specifies the path or location of the browser. You can use the FileIO Xtra to display a dialog box that allows the user to search for a browser. The `displayOpen()` method of the FileIO Xtra is useful for displaying an Open dialog box.

The form `browserName()` returns the name of the currently specified browser. Placing a pathname, like one found using the FileIO Xtra, as an argument in the form `browserName(fullPathToApplication)` allows the property to be set. The form `browserName(#enabled, trueOrFalse)` determines whether the specified browser launches automatically when the `goToNetPage` command is issued.

This command is only useful playing back in a projector or in Director, and has no effect when playing back in a browser.

This property can be tested and set.

Example

This statement refers to the location of the Netscape® browser:

```
browserName "My Disk:My Folder:Netscape"
```

This statement displays the browser name in a Message window:

```
put browserName()
```

build()

Usage

```
-- Lingo syntax
member(whichCastmember).modelResource(whichModelResource).build()

// JavaScript syntax
member(whichCastmember).modelResource(whichModelResource).build();
```

Description

3D mesh command; constructs a mesh. This command is only used with model resources whose type is `#mesh`.

You must use the `build()` command in the initial construction of the mesh, after changing any of the `face` properties of the mesh, and after using the `generateNormals()` command.

Parameters

None.

Example

This example creates a simple model resource whose type is #mesh, specifies its properties, and then creates a new model using the model resource. The process is outlined in the following line-by-line explanation of the example code:

Line 1 creates a mesh called Plane, which has one face, three vertices, and a maximum of three colors. The number of normals and the number of texture coordinates are not set. The normals are created by the `generateNormals` command.

Line 2 defines the vectors that will be used as the vertices for Plane.

Line 3 assigns the vectors to the vertices of the first face of Plane.

Line 4 defines the three colors allowed by the newMesh command.

Line 5 assigns the colors to the first face of Plane. The third color in the color list is applied to the first vertex of Plane, the second color to the second vertex, and the first color to the third vertex. The colors will spread across the first face of Plane in gradients.

Line 6 creates the normals of Plane with the generateNormals() command.

Line 7 calls the build() command to construct the mesh.

```
-- Lingo syntax
nm = member("Shapes").newMesh("Plane",1,3,0,3,0)
nm.vertexList = [vector(0,0,0), vector(20,0,0), vector(20, 20, 0)]
nm.face[1].vertices = [1,2,3]
```

```
nm.colorList = [rgb(255,255,0), rgb(0, 255, 0), rgb(0,0,255)]
nm.face[1].colors = [3,2,1]
nm.generateNormals(#smooth)
nm.build()
nm = member("Shapes").newModel("TriModel", nm)

// JavaScript syntax
nm = member("Shapes").newMesh("Plane",1,3,0,3,0);
nm.vertexList = [vector(0,0,0), vector(20,0,0), vector(20, 20, 0)];
nm.face[1].vertices = [1,2,3];
nm.colorList = [rgb(255,255,0), rgb(0, 255, 0), rgb(0,0,255)];
nm.face[1].colors = [3,2,1];
nm.generateNormals(#smooth);
nm.build();
nm = member("Shapes").newModel("TriModel", nm);
```

See also

generateNormals(), newMesh, face[ ]

cacheDocVerify()

Usage

```
-- Lingo syntax
cacheDocVerify #setting
cacheDocVerify()

// JavaScript syntax
cacheDocVerify symbol(setting);
cacheDocVerify();
```

Description

Function; sets how often the contents of a page on the Internet are refreshed with information from the projector's cache.

The form `cacheDocVerify()` returns the current setting of the cache.

The `cacheDocVerify` function is valid only for movies running in Director or as projectors. This function is not valid for movies with Adobe® Shockwave® content because they use the network settings of the browser in which they run.

```
-- Lingo syntax
on resetCache
    current = cacheDocVerify()
    if current = #once then
        alert "Turning cache verification on"
        cacheDocVerify #always
    end if
end

// JavaScript syntax
function resetCache() {
    current = cacheDocVerify();
    if (current == symbol("once")) {
        alert("Turning cache verification on");
        cacheDocVerify(symbol("always"))
    }
}
```

Parameters

cacheSetting Optional. A symbol that specifies how often the contents of a page on the Internet are refreshed. Possible values are `#once` (default) and `#always`. Specifying `#once` tells a movie to get a file from the Internet once and then use the file from the cache without looking for an updated version on the Internet. Specifying `#always` tells a movie to try to get an updated version of the file each time the movie calls a URL.

See also

`cacheSize()`, `clearCache`

cacheSize()

Usage

```
-- Lingo syntax
cacheSize Size
cacheSize()

// JavaScript syntax
cacheSize(Size);
cacheSize();
```

Description

Function and command; sets the cache size of Director.

The `cacheSize` function is valid only for movies running in Director or as projectors. This function is not valid for movies with Shockwave content because they use the network settings of the browser in which they run.

Parameters

newCacheSize Optional. An integer that specifies the cache size, in kilobytes.

Example

This handler checks whether the browser's cache setting is less than 1 MB. If it is, the handler displays an alert and sets the cache size to 1 MB:

```
-- Lingo syntax
on checkCache if
    cacheSize()<1000 then
        alert "increasing cache to 1MB"
        cacheSize 1000
    end if
end

// JavaScript syntax
function checkCache() {
    if (cacheSize() < 1000) {
        alert("increasing cache to 1MB");
        cacheSize(1000);
    }
}
```

See also

`cacheDocVerify()`, `clearCache`

call

Usage

```
call #handlerName, script, {args...}
call (#handlerName, scriptInstance, {args...})
```

Description

Command; sends a message that invokes a handler in a specified script or list of scripts.

The `call` command can use a variable as the name of the handler. Messages passed using `call` are not passed to other scripts attached to the sprite, cast member scripts, frame scripts, or movie scripts.

Parameters

`symHandlerName` Required. A symbol that specifies the handler to activate.

`scriptInstance` Required. A reference to the script or list of scripts that contains the handler. If *`scriptInstance`* is a single script instance, an error alert occurs if the handler is not defined in the script's ancestor script. If *`scriptInstance`* is a list of script instances, the message is sent to each item in the list in turn; if the handler is not defined in the ancestor script, no alert is generated.

`args` Optional. Any optional parameters to be passed to the handler.

Example

This handler sends the message bumpCounter to the first behavior script attached to sprite 1:

```
-- Lingo syntax
on mouseDown me
    -- get the reference to the first behavior of sprite 1
    set xref = getAt (the scriptInstanceList of sprite 1,1)
    -- run the bumpCounter handler in the referenced script,
    -- with a parameter
    call (#bumpCounter, xref, 2)
end

// JavaScript syntax
function mouseDown() {
    // get the reference to the first behavior of sprite 1
    xref = getAt(sprite(1).script(1));
    // run the bumpCounter handler in the referenced script
    call(symbol("bumpcounter"), xref, 2);
}
```

The following example shows how a `call` statement can call handlers in a behavior or parent script and its ancestor.

- This is the parent script:

```
-- Lingo syntax
-- script Man
property ancestor

on new me
    set ancestor = new(script "Animal", 2)
    return me
end
on run me, newTool
    put "Man running with "&the legCount of me&" legs"
end
```

- This is the ancestor script:

```
-- script Animal
property legCount

on new me, newLegCount
    set legCount = newLegCount
    return me
end
on run me
    put "Animal running with "& legCount &" legs"
end
on walk me
    put "Animal walking with "& legCount &" legs"
end
```

- The following statements use the parent script and ancestor script.

This statement creates an instance of the parent script:

```
set m = new(script "man")
```

This statement makes the man walk:

```
call #walk, m
-- "Animal walking with 2 legs"
```

This statement makes the man run:

```
set msg = #run
call msg, m
-- "Man running with 2 legs and rock"
```

This statement creates a second instance of the parent script:

```
set m2 = new(script "man")
```

This statement sends a message to both instances of the parent script:

```
call msg, [m, m2]

-- "Man running with 2 legs "
-- "Man running with 2 legs "
```

callAncestor

Usage

```
callAncestor handlerName, script, {args...}
```

Description

Command; sends a message to a child object's ancestor script.

Ancestors can, in turn, have their own ancestors.

When you use `callAncestor`, the name of the handler can be a variable, and you can explicitly bypass the handlers in the primary script and go directly to the ancestor script.

Parameters

`symHandlerName` Required. A symbol that specifies the handler to activate.

scriptInstance Required. A reference to the script or list of scripts that contains the handler. If *scriptInstance* is a single script instance, an error alert occurs if the handler is not defined in the script's ancestor script. If *scriptInstance* is a list of script instances, the message is sent to each item in the list in turn; if the handler is not defined in the ancestor script, no alert is generated.

args Optional. Any optional parameters to be passed to the handler.

Example

This example shows how a `callAncestor` statement can call handlers in the ancestor of a behavior or parent script.

- This is the parent script:

```
-- script "man"
property ancestor

on new me, newTool
    set ancestor = new(script "Animal", 2)
    return me
end
on run me
    put "Man running with "&the legCount of me&"legs"
end
```

- This is the ancestor script:

```
-- script "animal"
property legCount

on new me, newLegCount
    set legCount = newLegCount
    return me
end
on run me
    put "Animal running with "& legCount &" legs"
end
on walk me
    put "Animal walking with "& legCount &" legs"
end
```

- The following statements use the parent script and ancestor script.

  This statement creates an instance of the parent script:

```
set m = new(script "man")
```

  This statement makes the man walk:

```
call #walk, m
-- "Animal walking with 2 legs"
```

  This statement makes the man run:

```
set msg = #run
callAncestor msg, m
-- "Animal running with 2 legs"
```

  This statement creates a second instance of the parent script:

```
set m2 = new(script "man")
```

  This statement sends a message to the ancestor script for both men:

```
callAncestor #run,[m,m2]
-- "Animal running with 2 legs"
```

```
   -- "Animal running with 2 legs"
```

See also
ancestor, new()

callFrame()

Usage

```
-- Lingo syntax
spriteObjRef.callFrame(flashFrameNameOrNum)

// JavaScript syntax
spriteObjRef.callFrame(flashFrameNameOrNum);
```

Description

Command; used to call a series of actions that reside in a frame of a Flash® movie sprite.

This command sends a message to the Flash® ActionScript engine and triggers the actions to execute in the Flash movie.

Parameters

flashFrameNameOrNum Required. A string or number that specifies the name or number of the frame to call.

Example

This Lingo executes the actions that are attached to frame 10 of the Flash movie in sprite 1:

```
-- Lingo syntax
sprite(1).callFrame(10)

// JavaScript syntax
sprite(1).callFrame(10);
```

camera()

Usage

```
member(whichCastMember).camera(whichCamera)
member(whichCastMember).camera[index]
member(whichCastMember).camera(whichCamera).whichCameraProperty
member(whichCastMember).camera[index].whichCameraProperty
sprite(whichSprite).camera{(index)}
sprite(whichSprite).camera{(index)}.whichCameraProperty
```

Description

3D element; an object at a vector position from which the 3D world is viewed.

Each sprite has a list of cameras. The view from each camera in the list is displayed on top of the view from camera with lower *index* positions. You can set the `rect (camera)` property of each camera to display multiple views within the sprite.

Cameras are stored in the camera palette of the cast member. Use the `newCamera` and `deleteCamera` commands to create and delete cameras in a 3D cast member.

The `camera` property of a sprite is the first camera in the list of cameras of the sprite. The camera referred to by `sprite(whichSprite).camera` is the same as `sprite(whichSprite).camera(1)`. Use the `addCamera` and `deleteCamera` commands to build the list of cameras in a 3D sprite.

Example

This statement sets the camera of sprite 1 to the camera named TreeCam of the cast member named Picnic.

```
sprite(1).camera = member("Picnic").camera("TreeCam")
```

This statement sets the camera of sprite 1 to camera 2 of the cast member named Picnic.

```
sprite(1).camera = member("Picnic").camera[2]
```

See also
`bevelDepth`, `overlay`, `modelUnderLoc`, `spriteSpaceToWorldSpace`, `fog`, `clearAtRender`

cameraCount()

Usage

```
-- Lingo syntax
sprite(whichSprite).cameraCount()

// JavaScript syntax
sprite(whichSprite).cameraCount();
```

Description

3D command; returns the number items in the list of cameras of the sprite.

Parameters

None.

Example

This statement shows that sprite 5 contains three cameras.

```
-- Lingo syntax
put sprite(5).cameraCount()
-- 3

// JavaScript syntax
put(sprite(5).cameraCount());
// 3
```

See also
`addCamera`, `deleteCamera`

cancelIdleLoad()

Usage

```
-- Lingo syntax
_movie.cancelIdleLoad(intLoadTag)

// JavaScript syntax
_movie.cancelIdleLoad(intLoadTag);
```

Description

Movie method; cancels the loading of all cast members that have the specified load tag.

Parameters

intLoadTag Required. An integer that specifies a group of cast members that have been queued for loading when the computer is idle.

Example

This statement cancels the loading of cast members that have an idle load tag of 20:

```
-- Lingo syntax
_movie.cancelIdleLoad(20)

// JavaScript syntax
_movie.cancelIdleLoad(20);
```

See also

idleLoadTag, Movie

castLib()

Usage

```
-- Lingo syntax
castLib(castNameOrNum)

// JavaScript syntax
castLib(castNameOrNum);
```

Description

Top level function; returns a reference to a specified cast library.

The default cast library number is 1. To specify a cast member in a cast library other than cast 1, set `castLib()` to specify the alternative cast library.

Parameters

castNameOrNum Required. A string that specifies the cast library name, or an integer that specifies the cast library number.

Example

This statement sets the variable `parts` to the second cast library:

```
-- Lingo syntax
parts = castLib(2)

// JavaScript syntax
var parts = castLib(2);
```

See also

Cast Library, castLibNum

channel() (Top level)

Usage

```
-- Lingo syntax
channel(soundChannelNameOrNum)

// JavaScript syntax
channel(soundChannelNameOrNum);
```

Description

Top level function; returns a reference to a Sound Channel object.

Parameters

soundChannelNameOrNum Required. A string that specifies the name of a sound channel, or an integer that specifies the index position of a sound channel.

Example

This statement sets the variable `newChannel` to sound channel 9:

```
-- Lingo syntax
newChannel = channel(9)

// JavaScript syntax
var newChannel = channel(9);
```

See also

Sound Channel

channel() (Sound)

Usage

```
-- Lingo syntax
_sound.channel(intChannelNum)

// JavaScript syntax
_sound.channel(intChannelNum);
```

Description

Sound method; returns a reference to a specified sound channel.

The functionality of this method is identical to the top level `sound()` method.

Parameters

intChannelNum Required. An integer that specifies the sound channel to reference.

Example

This statement sets the variable named `myChannel` to sound channel 2:

```
-- Lingo syntax
myChannel = _sound.channel(2)

// JavaScript syntax
var myChannel = _sound.channel(2);
```

See also
Sound, sound(), Sound Channel

chapterCount()

Usage

```
-- Lingo syntax
dvdObjRef.chapterCount({intTitle})

// JavaScript syntax
dvdObjRef.chapterCount({intTitle});
```

Description

DVD method; indicates the number of available chapters in a title.

Parameters

intTitle Optional. An integer that specifies the title that contains the chapters to count. If omitted, chapterCount() returns the number of available chapters in the current title.

Example

This statement returns the number of chapters in the current title:

```
-- Lingo syntax
trace (member(1).chapterCount)-- 17

// JavaScript syntax
trace (member(1). chapterCount);// 17
```

See also
chapterCount, DVD

charPosToLoc()

Usage

```
--Lingo syntax
memberObjRef.charPosToLoc(nthCharacter)

// JavaScript syntax
memberObjRef.charPosToLoc(nthCharacter);
```

Description

Field function; returns the point in the entire field cast member (not just the part that appears on the Stage) that is closest to a specified character. This is useful for determining the location of individual characters.

Values for charPosToLoc are in pixels from the top left corner of the field cast member. The *nthCharacter* parameter is 1 for the first character in the field, 2 for the second character, and so on.

Parameters

nthCharacter Required. The character to test.

Example

The following statement determines the point where the fiftieth character in the field cast member Headline appears and assigns the result to the variable `location`:

```
-- Lingo syntax
location = member("Headline").charPosToLoc(50)

// JavaScript syntax
var location = member("Headline").charPosToLoc(50);
```

chars()

Usage

```
chars(stringExpression, firstCharacter, lastCharacter)
```

Description

Function (Lingo only); identifies a substring of characters in an expression.

The expressions *`firstCharacter`* and *`lastCharacter`* must specify a position in the string.

If *`firstCharacter`* and *`lastCharacter`* are equal, then a single character is returned from the string. If *`lastCharacter`* is greater than the string length, only a substring up to the length of the string is identified. If *`lastCharacter`* is before *`firstCharacter`*, the function returns the value `EMPTY`.

To see an example of `chars()` used in a completed movie, see the Text movie in the Learning/Lingo folder inside the Director application folder.

In JavaScript syntax, use the String object's `substr()` function.

Parameters

`stringExpression` Required. A string that specifies the expression from which a substring is returned.

`firstCharacter` Required. An integer that specifies the point at which the substring starts.

`lastCharacter` Required. An integer that specifies the point at which the substring ends.

Example

This statement identifies the second character in the word *Adobe*:

```
put chars("Adobe", 2, 2)
-- "d"
```

This statement identifies the second through fifth characters of the word *Adobe*:

```
put chars("Adobe", 2, 5)
-- "dobe"
```

The following statement tries to identify the sixth through twentieth characters of the word Adobe. Because the word has only 10 characters, the result includes only the sixth through tenth characters.

```
put chars ("Adobe", 2, 20)
-- "dobe"
```

See also

char...of, length(), offset() (string function), number (characters)

charToNum()

Usage

```
(stringExpression).charToNum
charToNum(stringExpression)
```

Description

Function (Lingo only); returns the ASCII code that corresponds to the first character of an expression.

The `charToNum()` function is especially useful for testing the ASCII value of characters created by combining keys, such as the Control key and another alphanumeric key.

Director treats uppercase and lowercase letters the same if you compare them using the equal sign (=) operator; for example, the statement `put ("M" = "m")` returns the result 1 or `TRUE`.

Avoid problems by using `charToNum()` to return the ASCII code for a character and then use the ASCII code to refer to the character.

In JavaScript syntax, use the String object's `charCodeAt()` function.

Parameters

`stringExpression` Required. A string that specifies the expression to test.

Example

This statement displays the ASCII code for the letter A:

```
put ("A").charToNum
-- 65
```

The following comparison determines whether the letter entered is a capital A, and then navigates to either a correct sequence or incorrect sequence in the Score:

```
-- Lingo syntax
on CheckKeyHit theKey
    if (theKey).charToNum = 65 then
        go "Correct Answer"
    else
        go "Wrong Answer"
    end if
end

// JavaScript syntax
function CheckKeyHit(theKey) {
    if (theKey.charToNum() == 65)
        go("Correct Answer");
    } else {
        go("Wrong Answer");
    }
}
```

See also

`numToChar()`

clearAsObjects()

Usage

```
-- Lingo syntax
clearAsObjects()

// JavaScript syntax
clearAsObjects();
```

Description

Command; resets the global Flash Player used for ActionScript objects and removes any ActionScript objects from memory. The command does not clear or reset references to those objects stored in Lingo. Lingo references will persist but will refer to nonexistent objects. You must set each reference to `VOID` individually.

The `clearAsObjects()` command affects only global objects, such as the array created in this statement:

```
-- Lingo syntax
myGlobalArray = newObject(#array)

// JavaScript syntax
myGlobalArray = new Array();
```

The `clearAsObjects()` command has no effect on objects created within sprite references, such as the following:

```
myArray = sprite(2).newObject(#array)
```

Parameters

None.

Example

This statement clears all globally created ActionScript objects from memory:

```
-- Lingo syntax
clearAsObjects()

// JavaScript syntax
clearAsObjects();
```

See also

newObject(), setCallback()

clearCache

Usage

```
clearCache
```

Description

Command; clears the Director network cache.

The `clearCache` command clears only the cache, which is separate from the browser's cache.

If a file is in use, it remains in the cache until it is no longer in use.

Parameters

None.

Example

This handler clears the cache when the movie starts:

```
-- Lingo syntax
on startMovie
    clearCache
end

// JavaScript syntax
function startMovie() {
    clearCache();
}
```

See also

cacheDocVerify(), cacheSize()

clearError()

Usage

```
-- Lingo syntax
memberObjRef.clearError()

// JavaScript syntax
memberObjRef.clearError();
```

Description

Flash command; resets the error state of a streaming Flash cast member to 0.

When an error occurs while a cast member is streaming into memory, Director sets the cast member's `state` property to -1 to indicate that an error occurred. When this happens, you can use the `getError` function to determine what type of error occurred and then use the `clearError` command to reset the cast member's error state to 0. After you clear the member's error state, Director tries to open the cast member if it is needed again in the Director movie. Setting a cast member's `pathName`, `linked`, and `preload` properties also automatically clears the error condition.

Parameters

None.

Example

This handler checks to see if an out-of-memory error occurred for a Flash cast member named Dali, which was streaming into memory. If a memory error occurred, the script uses the `unloadCast` command to try to free some memory; it then branches the playhead to a frame in the Director movie named Artists, where the Flash movie sprite first appears, so Director can again try to play the Flash movie. If something other than an out-of-memory error occurred, the script goes to a frame named Sorry, which explains that the requested Flash movie can't be played.

```
-- Lingo syntax
on CheckFlashStatus
    if (member("Dali").getError() = #memory) then
        member("Dali").clearError()
        member("Dali").unload()
        unloadCast
    else
        _movie.go("Sorry")
```

```
    end if
end

// JavaScript syntax
function CheckFlashStatus() {
    var ge = member("Dali").getError();
    if (ge = "memory") {
        member("Dali").clearError();
        unloadCast;
        _movie.go("Artists");
    } else {
        _movie.go("Sorry");
    }
}
```

See also

state (Flash, SWA), getError() (Flash, SWA)

clearFrame()

Usage

```
-- Lingo syntax
_movie.clearFrame()

// JavaScript syntax
_movie.clearFrame();
```

Description

Movie method; clears all sprite channels in a frame during Score recording.

Parameters

None.

Example

The following handler clears the content of each frame before it edits that frame during Score generation:

```
-- Lingo syntax
on newScore
    _movie.beginRecording()
    repeat with counter = 1 to 50
        _movie.clearFrame()
        _movie.frameScript = 25
        _movie.updateFrame()
    end repeat
    _movie.endRecording()
end

// JavaScript syntax
function newScore() {
    _movie.beginRecording();
    for (var i = 1; i <= 50; i++) {
        _movie.clearFrame();
        _movie.frameScript = 25;
        _movie.updateFrame();
    }
    _movie.endRecording();
```

```
}
```

See also

beginRecording(), endRecording(), Movie, updateFrame()

clearGlobals()

Usage

```
-- Lingo syntax
_global.clearGlobals()

// JavaScript syntax
_global.clearGlobals();
```

Description

Global method; sets all global variables to VOID (Lingo) or null (JavaScript syntax).

This method is useful when initializing global variables or when opening a new movie that requires a new set of global variables.

Parameters

None.

Example

The following handlers set all global variables to VOID (Lingo) or null (JavaScript):

```
-- Lingo syntax
on mouseDown
    _global.clearGlobals()
end

// JavaScript syntax
function mouseDown() {
    _global.clearGlobals();
}
```

See also

Global

clone

Usage

```
member(whichCastmember).model(whichModel).clone(cloneName)
member(whichCastmember).group(whichGroup).clone(cloneName)
member(whichCastmember).light(whichLight).clone(cloneName)
member(whichCastmember).camera(whichCamera).clone(cloneName)
```

Description

3D command; creates a copy of the model, group, light, or camera and all of its children. The clone shares the parent of the model, group, light, or camera from which it was cloned.

A clone of a model uses the same model resource and is assigned the same shaderList as the original model.

If you do not specify the *cloneName*, or if you specify "", the clone will not be counted by the count method, but it will appear in the scene.

Parameters

cloneName Required. Specifies the name of the new clone.

Example

This statement creates a clone named Teapot2 from the model named Teapot, and returns a reference to the new model.

```
-- Lingo
teapotCopy = member("3D World").model("Teapot").clone("Teapot2")

// Javascript
teapotCopy = member("3D World").getProp("model","Teapot").clone("Teapot2")
```

See also

cloneDeep, cloneModelFromCastmember, cloneMotionFromCastmember, loadFile()

cloneDeep

Usage

```
member(whichCastmember).model(whichModel).cloneDeep(cloneName)
member(whichCastmember).group(whichGroup).cloneDeep(cloneName)
member(whichCastmember).light(whichLight).cloneDeep(cloneName)
member(whichCastmember).camera(whichCamera).cloneDeep(cloneName)
```

Description

3D command; creates a copy of the model, group, light, or camera plus all of the following:

- The model resources, shaders, and textures used by the original model or group
- The children of the model, group, light, or camera
- The model resources, shaders, and textures used by the children

This method uses more memory and takes more time than the clone command.

Parameters

cloneName Required. Specifies the name of the new clone.

Example

This statement creates a copy of the model named Teapot, its children, and the model resources, shaders, and textures used by Teapot and its children. The variable teapotCopy is a reference to the cloned model.

```
-- Lingo
teapotCopy = member("3D World").model("Teapot").cloneDeep("Teapot2")

// Javascript
teapotCopy =member("3DWorld").getProp("model","Teapot").cloneDeep("Teapot2")
```

See also

clone, cloneModelFromCastmember, cloneMotionFromCastmember, loadFile()

cloneModelFromCastmember

Usage

```
member(whichCastmember).cloneModelFromCastmember(newModelName, sourceModelName,
sourceCastmember)
```

Description

3D command; copies a model from a cast member, renames it, and inserts it into a cast member as a child of its 3D world.

This command also copies the children of *sourceModelName*, as well as the model resources, shaders, and textures used by the model and its children.

The source cast member must be finished loading for this command to work correctly.

Parameters

newModelName Required. Specifies the name of the newly cloned model.

sourceModelName Required. Specifies the model to clone.

sourceCastMember Required. Specifies the cast member that contains the model to clone.

Example

This statement makes a copy of the model named Pluto of the cast member named Scene and inserts it into the cast member named Scene2 with the new name Planet. The children of Pluto are also imported, as are the model resources, shaders, and textures used by Pluto and its children.

```
--Lingo
member("Scene2").cloneModelFromCastmember("Planet", "Pluto", member("Scene"))

// Javascript
member("Scene2").cloneModelFromCastmember("Planet", "Pluto", member("Scene"));
```

See also

cloneMotionFromCastmember, clone, cloneDeep, loadFile()

cloneMotionFromCastmember

Usage

```
member(whichCastmember).cloneMotionFromCastmember(newMotionName, sourceMotionName,
sourceCastmember)
```

Description

3D command; copies a motion from a cast member, renames it, and inserts it into a cast member.

The source cast member must be finished loading for this command to work correctly.

Parameters

newMotionName Required. Specifies the name of the newly cloned motion.

sourceMotionName Required. Specifies the motion to clone.

sourceCastMember Required. Specifies the cast member that contains the motion to clone.

Example

This statement copies the motion named Walk from the cast member named ParkScene, names the copy FunnyWalk, and puts the copy in the cast member gbMember.

```
--Lingo
member("gbMember").cloneMotionFromCastmember("FunnyWalk", "Walk", member("ParkScene"))

// Javascript
member("gbMember").cloneMotionFromCastmember("FunnyWalk", "Walk", member("ParkScene"));
```

See also

map (3D), cloneModelFromCastmember, clone, cloneDeep, loadFile()

close()

Usage

```
-- Lingo syntax
windowObjRef.close()

// JavaScript syntax
windowObjRef.close();
```

Description

Window method; closes a window.

Closing a window that is already closed has no effect.

Be aware that closing a window does not stop the movie in the window nor clear it from memory. This method simply closes the window in which the movie is playing. You can reopen it quickly by using the `open() (Window)` method. This allows rapid access to windows that you want to keep available.

If you want to completely dispose of a window and clear it from memory, use the `forget()` method. Make sure that nothing refers to the movie in that window if you use the `forget()` method, or you will generate errors when scripts try to communicate or interact with the forgotten window.

Parameters

None.

Example

This statement closes the window named Panel, which is in the subfolder MIAW Sources within the current movie's folder:

```
-- Lingo syntax
window(_movie.path & "MIAW Sources\Panel").close()

// JavaScript syntax
window(_movie.path + "MIAW Sources\\Panel").close();
```

This statement closes the window that is number 5 in `windowList`:

```
-- Lingo syntax
window(5).close()

// JavaScript syntax
window(5).close();
```

See also
forget() (Window), open() (Window), Window

closeFile()

Usage

```
-- Lingo syntax
fileioObjRef.closeFile()

// JavaScript syntax
fileioObjRef.closeFile();
```

Description

Fileio method; Closes a file.

Parameters

None.

See also

Fileio

closeXlib

Usage

```
closeXlib whichFile
```

Description

Command; closes an Xlibrary file.

Xtra extensions are stored in Xlibrary files. Xlibrary files are resource files that contain Xtra extensions. HyperCard XCMDs and XFCNs can also be stored in Xlibrary files.

The `closeXlib` command doesn't work for URLs.

In Windows, using the DLL extension for Xtra extensions is optional.

It is good practice to close any file you have opened as soon as you have finished using it.

Note: *This command is not supported in Shockwave Player.*

Parameters

`whichFile` Optional. Specifies the Xlibrary file to close. If *`whichFile`* is in a folder other than that for the current movie, *`whichFile`* must specify a pathname. If *`whichFile`* is omitted, all open Xlibraries are closed.

Example

This statement closes all open Xlibrary files:

```
closeXlib
```

This statement closes the Xlibrary Video Disc Xlibrary when it is in the same folder as the movie:

```
closeXlib "Video Disc Xlibrary"
```

The following statement closes the Xlibrary Transporter Xtra extensions in the folder New Xtras, which is in the same folder as the movie. The disk is identified by the variable currentDrive:

```
closeXlib "@:New Xtras:Transporter Xtras"
```

See also
Interface(), openXlib

color()

Usage

```
-- Lingo syntax
color(intPaletteIndex)
color(intRed, intGreen, intBlue)

// JavaScript syntax
color(intPaletteIndex);
color(intRed, intGreen, intBlue);
```

Description

Top level function and data type. Returns a Color data object using either RGB or 8-bit palette index values.

The resulting color object can be applied to cast members, sprites, and the Stage where appropriate.

Parameters

intPaletteIndex Required if using 8-bit palette values. An integer that specifies the 8-bit palette value to use. Valid values range from 0 to 255. All other values are truncated.

intRed Required if using RGB values. An integer that specifies the red color component in the current palette. Valid values range from 0 to 255. All other values are truncated.

intGreen Required if using RGB values. An integer that specifies the green color component in the current palette. Valid values range from 0 to 255. All other values are truncated.

intBlue Required if using RGB values. An integer that specifies the blue color component in the current palette. Valid values range from 0 to 255. All other values are truncated.

Example

These statements display the color of sprite 6 in the Message window, and then set the color of sprite 6 to a new value:

```
-- Lingo syntax
put(sprite(6).color) -- paletteIndex(255)
sprite(6).color = color(137)
put(sprite(6).color) -- paletteIndex(137)

// JavaScript syntax
put(sprite(6).color) // paletteIndex(255);
sprite(6).color = color(137);
put(sprite(6).color) // paletteIndex(137);
```

constrainH()

Usage

```
-- Lingo syntax
_movie.constrainH(intSpriteNum, intPosn)

// JavaScript syntax
_movie.constrainH(intSpriteNum, intPosn);
```

Description

Movie method; returns an integer whose value depends on the horizontal coordinates of the left and right sides of a sprite.

The returned integer can be one of three possible values.

- If the *intPosn* parameter is between the values of the sprite's left and right coordinates, the returned integer equals *intPosn*.
- If the *intPosn* parameter is less than the value of the sprite's left coordinate, the returned integer changes to the value of the sprite's left coordinate.
- If the *intPosn* parameter is greater than the value of the sprite's right coordinate, the returned integer changes to the value of the sprite's right coordinate.

This method does not change the sprite's properties.

Both the `constrainH()` and `constrainV()` methods constrain only one axis each.

Parameters

intSpriteNum Required. An integer that specifies the sprite whose horizontal coordinates are evaluated against *intPosn*.

intPosn Required. An integer to be evaluated against by the horizontal coordinates of the left and right sides of the sprite identified by *intSpriteNum*.

Example

These statements check the `constrainH` function for sprite 1 when it has left and right coordinates of 40 and 60:

```
-- Lingo syntax
put(constrainH(1, 20)) -- 40
put(constrainH(1, 55)) -- 55
put(constrainH(1, 100)) -- 60

// JavaScript syntax
put(constrainH(1, 20)); // 40
put(constrainH(1, 55)); // 55
put(constrainH(1, 100)); // 60
```

This statement constrains a moveable slider (sprite 1) to the edges of a gauge (sprite 2) when the mouse pointer goes past the edge of the gauge:

```
-- Lingo syntax
sprite(1).locH = _movie.constrainH(2, _mouse.mouseH)

// JavaScript syntax
sprite(1).locH = _movie.constrainH(2, _mouse.mouseH);
```

See also
constrainV(), Movie

constrainV()

Usage

```
-- Lingo syntax
_movie.constrainV(intSpriteNum, intPosn)

// JavaScript syntax
_movie.constrainV(intSpriteNum, intPosn);
```

Description

Movie method; returns an integer whose value depends on the vertical coordinates of the top and bottom sides of a sprite.

The returned integer can be one of three possible values.

- If the *intPosn* parameter is between the values of the sprite's top and bottom coordinates, the returned integer equals *intPosn*.
- If the *intPosn* parameter iis less than the value of the sprite's top coordinate, the returned integer changes to the value of the sprite's top coordinate.
- If the *intPosn* parameter iis greater than the value of the sprite's bottom coordinate, the returned integer changes to the value of the sprite's bottom coordinate.

This method does not change the sprite's properties.

Both the `constrainV()` and `constrainH()`s constrain only one axis each.

Parameters

intSpriteNum Required. An integer that identifies the sprite whose vertical coordinates are evaluated against *intPosn*.

intPosn Required. An integer to be evaluated against by the vertical coordinates of the left and right sides of the sprite identified by *intSpriteNum*.

Example

These statements check the constrainV function for sprite 1 when it has top and bottom coordinates of 40 and 60:

```
-- Lingo syntax
put(constrainV(1, 20)) -- 40
put(constrainV(1, 55)) -- 55
put(constrainV(1, 100)) -- 60

// JavaScript syntax
put(constrainV(1, 20)); // 40
put(constrainV(1, 55)); // 55
put(constrainV(1, 100)); // 60
```

This statement constrains a moveable slider (sprite 1) to the edges of a gauge (sprite 2) when the mouse pointer moves past the edge of the gauge:

```
-- Lingo syntax
sprite(1).locV = _movie.constrainV(2, _mouse.mouseH)
```

```
// JavaScript syntax
sprite(1).locV = _movie.constrainV(2, _mouse.mouseH);
```

See also
constrainH(), Movie

copyPixels()

Usage

```
-- Lingo syntax
imageObjRef.copyPixels(sourceImgObj, destRectOrQuad, sourceRect {, paramList})

// JavaScript syntax
imageObjRef.copyPixels(sourceImgObj, destRectOrQuad, sourceRect {, paramList});
```

Description

Image method. Copies the contents of a rectangle in an existing image object into a new image object.

When copying pixels from one area of a cast member to another area of the same member, it is best to copy the pixels first into a duplicate image object before copying them back into the original member. Copying directly from one area to another in the same image is not recommended.

To simulate matte ink with `copyPixels()`, create a matte object with `createMatte()` and then pass that object as the `#maskImage` parameter of `copyPixels()`.

To see an example of quad used in a completed movie, see the Quad movie in the Learning/Lingo folder inside the Director application folder.

During copyPixel, if the useAlpha property is false for either the source or the destination image, the destination does not have any alpha information. If you want the destination to have alpha content, the useAlpha property should be true for both source and destination.

Parameters

`sourceImgObj` Required. A reference to the source image object from which pixels are copied.

`destRectOrQuad` Required if copying pixels into a screen coordinate rectangle or a floating point quad. The rectangle or quad into which pixels are copied.

`sourceRect` Required. The source rectangle from which pixels are copied.

`paramList` Optional. A parameter list that can be used to manipulate the copied pixels before they are placed into *`destRect`* or *`destQuad`*. The property list may contain any or all of the following parameters.

Property	Use and Effect
#color	The foreground color to apply for colorization effects. The default color is black.
#bgColor	The background color to apply for colorization effects or background transparency. The default color is white.
#ink	The type of ink to apply to the copied pixels. This can be an ink symbol or the corresponding numeric ink value. The default ink is `#copy`.

Property	Use and Effect
#blendLevel	The degree of blend (transparency) to apply to the copied pixels. The range of values is from 0 to 255. The default value is 255 (opaque). Using a value less than 255 forces the `#ink` setting to be `#blend`, or `#blendTransparent` if it was originally `#backgroundTransparent`. #blendLevel could also be replaced with `#blend`; if so, use use a value range of 0 to 100.
#dither	A `TRUE` or `FALSE` value that determines whether the copied pixels will be dithered when placed into the *`destRect`* in 8- and 16-bit images. The default value is `FALSE`, which maps the copied pixels directly into the *`imageObjRef`*'s color palette.
#useFastQuads	A `TRUE` or `FALSE` value that determines whether quad calculations are made using the faster but less precise method available in Director when copying pixels into *`destQuad`*. Set to `TRUE` to use quads for simple rotation and skew operations. Set to `FALSE` for arbitrary quads, such as those used for perspective transformations. The default value is `FALSE`.
#maskImage	Specifies a mask or matte object, created with the `creatMask()` or `createMatte()`s, that will be used as a mask for the pixels being copied. This enables the effects of mask and matte sprite inks to be duplicated. If the source image has an alpha channel and its `useAlpha` property is `TRUE`, the alpha channel is used and the specified mask or matte is ignored. The default is no mask.
#maskOffset	A point indicating the amount of x and y offset to apply to the mask specified by `#maskImage`. The offset is relative to the upper left corner of the source image. The default offset is (0, 0).

Example

This statement copies the entire image of member Happy into the rectangle of member flower. If the members are different sizes, the image of member Happy will be resized to fit the rectangle of member flower.

```
-- Lingo
member("flower").image.copyPixels(member("Happy").image,member("flower").image.rect,
member("Happy").image.rect)

// JavaScript syntax
member("flower").image.copyPixels(member("Happy").image,member("flower").image.rect,
member("Happy").image.rect);
```

The following statement copies part of the image of member Happy into part of member flower. The part of the image copied from Happy is within rectangle(0, 0, 200, 90). It is pasted into rectangle(20, 20, 100, 40) within the image of member flower. The copied portion of Happy is resized to fit the rectangle into which it is pasted.

```
-- Lingo
member("flower").image.copyPixels(member("Happy").image,
rect(20,20,100,40),rect(0,0,200,90))

// JavaScript syntax
member("flower").image.copyPixels(member("Happy").image,
rect(20,20,100,40),rect(0,0,200,90))
```

The following statement copies the entire image of member Happy into a rectangle within the image of member flower. The rectangle into which the copied image of member Happy is pasted is the same size as the rectangle of member Happy, so the copied image is not resized. The blend level of the copied image is 50, so it is semi-transparent, revealing the part of member flower it is pasted over.

```
-- Lingo
member("flower").image.copyPixels(member("Happy").image, rect(90,110,290,310),
member("Happy").image.rect, [#blendLevel: 50])

// JavaScript syntax
member("flower").image.copyPixels(member("Happy").image, rect(90,110,290,310),
member("Happy").image.rect, \propList(symbol("blendLevel"),50))
```

See also
color(), image()

copyToClipBoard()

Usage

```
-- Lingo syntax
memberObjRef.copyToClipBoard()

// JavaScript syntax
memberObjRef.copyToClipBoard();
```

Description

Member method; copies a specified cast member to the Clipboard.

Calling this method does not require the Cast window to be active.

This method is useful when copying cast members between movies or applications.

Parameters

None.

Example

This statement copies the cast member named `chair` to the Clipboard:

```
-- Lingo syntax
member("chair").copyToClipBoard()

// JavaScript syntax
member("chair").copyToClipBoard();
```

This statement copies cast member number 5 to the Clipboard:

```
--- Lingo syntax
member(5).copyToClipBoard()

// JavaScript syntax
member(5).copyToClipBoard();
```

See also
Member, pasteClipBoardInto()

cos()

Usage

```
(angle).cos
cos (angle)
```

Description

Function (Lingo only); calculates the cosine of the specified angle, which must be expressed in radians.

In JavaScript syntax, use the Math object's `cos()` function.

Parameters

angle Required. An integer that specifies the angle to test.

Example

The following statement calculates the cosine of PI divided by 2 and displays it in the Message window:

```
put (PI/2).cos
```

See also

atan(), PI, sin()

count()

Usage

```
-- Lingo syntax
list.count
object.count

// JavaScript syntax
list.count;
object.count;
```

Description

Function; returns the number of entries in a linear or property list, the number of properties in a parent script without counting the properties in an ancestor script, or the chunks of a text expression such as characters, lines, or words.

The `count` command works with linear and property lists, objects created with parent scripts, and `the globals` property.

To see an example of `count()` used in a completed movie, see the Text movie in the Learning/Lingo folder inside the Director application folder.

Parameters

None.

Example

This statement displays the number 3, the number of entries:

```
--Lingo syntax
put([10,20,30].count) -- 3

// JavaScript syntax
put(list(10,20,30).count); // 3
```

See also

globals

createFile()

Usage

```
-- Lingo syntax
fileioObjRef.createFile(stringFileName)

// JavaScript syntax
fileioObjRef.createFile(stringFileName);
```

Description

Fileio method; Creates a specified file.

Parameters

stringFileName Required. A string that specifies the path and name of the file to create.

Example

The below sample creates a file called “xtra.txt” in c:\.

```
-- Lingo syntax
objFileio = new xtra("fileio")
objFileio.createFile("c:\xtra.txt")

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.createFile("c:\xtra.txt");
```

See also

Fileio

createMask()

Usage

```
imageObject.createMask()
```

Description

This function creates and returns a mask object for use with the `copyPixels()` function.

Mask objects aren't image objects; they're useful only with the `copyPixels()` function for duplicating the effect of mask sprite ink. To save time, if you plan to use the same image as a mask more than once, it's best to create the mask object and save it in a variable for reuse.

Example

This statement copies the entire image of member Happy into a rectangle within the image of member brown square. Member gradient2 is used as a mask with the copied image. The mask is offset by 10 pixels up and to the left of the rectangle into which the image of member Happy is pasted.

```
member("brown square").image.copyPixels(member("Happy").image, rect(20, 20, 150, 108),
member("Happy").rect,[#maskImage:member("gradient2").image.createMask(),
maskOffset:point(-10, -10)])
```

See also3

copyPixels(), createMatte(), ink

createMatte()

Syntax

```
imageObject.createMatte({alphaThreshold})
```

Description

This function creates and returns a matte object that you can use with `copyPixels()` to duplicate the effect of the matte sprite ink. The matte object is created from the specified image object's alpha layer. The optional parameter *`alphaThreshold`* excludes from the matte all pixels whose alpha channel value is below that threshold. It is used only with 32-bit images that have an alpha channel. The *`alphaThreshold`* must be a value between 0 and 255.

Matte objects aren't image objects; they are useful only with the `copyPixels()` function. To save time, if you plan to use the same image as a matte more than once, it's best to create the matte and save it in a variable for reuse.

Example

This statement creates a new matte object from the alpha layer of the image object testImage and ignores pixels with alpha values below 50%:

```
newMatte = testImage.createMatte(128)
```

See also

`copyPixels()`, `createMask()`

crop() (Image)

Usage

```
-- Lingo syntax
imageObjRef.crop(rectToCropTo)

// JavaScript syntax
imageObjRef.crop(rectToCropTo);
```

Description

Image method. Returns a new image object that contains a copy of a source image object, cropped to a given rectangle.

Calling `crop()` does not alter the source image object.

The new image object does not belong to any cast member and has no association with the Stage. To assign the new image to a cast member, set the `image` property of that cast member.

Parameters

`rectToCropTo` Required. The rectangle to which the new image is cropped.

Example

This statement instructs Lingo to crop any sprite that refers to the digital video cast member Interview.

```
-- Lingo
Dot syntax:
member("Interview").crop = TRUE
Verbose syntax:
set the crop of member "Interview" to TRUE
```

```
// Javascript
member("Interview").crop=true
```

See also
image (Image), image(), rect (Image)

crop() (Bitmap)

Usage

```
-- Lingo syntax
memberObjRef.crop()

// JavaScript syntax
memberObjRef.crop();
```

Description

Bitmap command; allows a bitmap cast member to be cropped to a specific size.

You can use `crop` to trim existing cast members, or in conjunction with the picture of the Stage to grab a snapshot and then crop it to size for display.

The registration point is kept in the same location so the bitmap does not move in relation to the original position.

Parameters

rectToCropTo Required. Specifes the rectangle to which a cast member is cropped.

Example

This statement sets an existing bitmap member to a snapshot of the Stage, then crops the resulting image to a rectangle equal to sprite 10:

```
-- Lingo syntax
stageImage = (_movie.stage).image
spriteImage = stageImage.crop(sprite(10).rect)
member("sprite snapshot").image = spriteImage

// JavaScript syntax
var stageImage = (_movie.stage).image;
var spriteImage = stageImage.crop(sprite(10).rect);
member("sprite snapshot").image = spriteImage;
```

See also
picture (Member)

cross

Usage

```
vector1.cross(vector2)
```

Description

3D vector method; returns a vector which is perpendicular to both *vector1* and *vector2*.

Example

In this example, pos1 is a vector on the x axis and pos2 is a vector on the y axis. The value returned by `pos1.cross(pos2)` is `vector( 0.0000, 0.0000, 1.00000e4 )`, which is perpendicular to both pos1 and pos2.

```
ppos1 = vector(100, 0, 0)
pos2 = vector(0, 100, 0)

-- Lingo
put pos1.cross(pos2)
-- vector( 0.0000, 0.0000, 1.00000e4 )

// Javascript
trace(pos1.cross(pos2))
// vector( 0.0000, 0.0000, 1.00000e4 )
```

See also

crossProduct(), perpendicularTo

crossProduct()

Usage

```
vector1.crossProduct(vector2)
```

Description

3D vector method; returns a vector which is perpendicular to both *vector1* and *vector2*.

Example

In this example, pos1 is a vector on the x axis and pos2 is a vector on the y axis. The value returned by `pos1.crossProduct(pos2)` is `vector( 0.0000, 0.0000, 1.00000e4 )`, which is perpendicular to both pos1 and pos2.

```
pos1 = vector(100, 0, 0)
pos2 = vector(0, 100, 0)

-- Lingo
put pos1.crossProduct(pos2)
-- vector( 0.0000, 0.0000, 1.00000e4 )

// Javascript
trace(pos1.crossProduct(pos2))
// vector( 0.0000, 0.0000, 1.00000e4 )
```

See also

perpendicularTo, cross

cursor()

Usage

```
-- Lingo syntax
_player.cursor(intCursorNum)
_player.cursor(cursorMemNum, maskMemNum)
_player.cursor(cursorMemRef)
```

```
// JavaScript syntax
_player.cursor(intCursorNum);
_player.cursor(cursorMemNum, maskMemNum);
_player.cursor(cursorMemRef);
```

Description

Player method; changes the cast member or built-in cursor that is used for a cursor and stays in effect until you turn it off by setting the cursor to 0.

- Use the syntax `_player.cursor(cursorMemNum, maskMemNum)` to specify the number of a cast member to use as a cursor and its optional mask. The cursor's hot spot is the registration point of the cast member.

  The cast member that you specify must be a 1-bit cast member. If the cast member is larger than 16 by 16 pixels, Director crops it to a 16-by-16-pixel square, starting in the upper left corner of the image. The cursor's hot spot is still the registration point of the cast member.

- Use the syntax `_player.cursor(cursorMemRef)` for the custom cursors available through the Cursor Xtra.

  Note: *Although the Cursor Xtra allows cursors of different cast library types, text cast members cannot be used as cursors.*

- Use the syntax `_player.cursor(intCursorNum)` to specify default system cursors. The term `intCursorNum` must be one of the following integer values:

Value	Description
-1, 0	Arrow
1	I-Beam
2	Cross
3	Crossbar
4	Watch (Mac) or Hour glass (Windows)
5	North South East West (NSEW)
6	North South (NS)
200	Blank (hides cursor)
254	Help
256	Pencil
257	Eraser
258	Select
259	Bucket
260	Hand
261	Rectangle tool
262	Rounded rectangle tool
263	Circle tool
264	Line tool
265	Rich text tool

Value	Description
266	Text field tool
267	Button tool
268	Check box tool
269	Radio button tool
270	Placement tool
271	Registration point tool
272	Lasso
280	Finger
281	Dropper
282	Wait mouse down 1
283	Wait mouse down 2
284	Vertical size
285	Horizontal size
286	Diagonal size
290	Closed hand
291	No-drop hand
292	Copy (closed hand)
293	Inverse arrow
294	Rotate
295	Skew
296	Horizontal double arrow
297	Vertical double arrow
298	Southwest Northeast double arrow
299	Northwest Southeast double arrow
300	Smear/smooth brush
301	Air brush
302	Zoom in
303	Zoom out
304	Zoom cancel
305	Start shape
306	Add point
307	Close shape
308	Zoom camera

Value	Description
309	Move camera
310	Rotate camera
457	Custom

During system events such as file loading, the operating system may display the watch cursor and then change to the pointer cursor when returning control to the application, overriding the `cursor` command settings from the previous movie. To use `cursor()` at the beginning of any new movie that is loaded in a presentation using a custom cursor for multiple movies, store any special cursor resource number as a global variable that remains in memory between movies.

Cursor commands can be interrupted by an Xtra or other external agent. If the cursor is set to a value in Director and an Xtra or external agent takes control of the cursor, resetting the cursor to the original value has no effect because Director doesn't perceive that the cursor has changed. To work around this, explicitly set the cursor to a third value and then reset it to the original value.

Parameters

intCursorNum Required when using an integer to identify a cursor. An integer that specifies the built-in cursor to use as a cursor.

cursorMemNum Required when using a cast member number and its optional mask to identify the cursor. An integer that specifies the cast member number to use as a cursor.

maskMemNum Required when using a cast member number and its optional mask to identify the cursor. An integer that specifies the mask number of *cursorMemNum*.

cursorMemRef Required when using a cast member reference to identify the cursor. A reference to the cast member to use as a cursor.

Example

This statement changes the cursor to a watch cursor on the Mac, and hourglass in Windows, whenever the value in the variable named `status` equals 1:

```
-- Lingo syntax syntax
if (status = 1) then
    _player.cursor(4)
end if

// JavaScript syntax
if (status == 1) {
    _player.cursor(4);
}
```

This handler checks whether the cast member assigned to the variable is a 1-bit cast member and then uses it as the cursor if it is:

```
-- Lingo syntax syntax
on myCursor(someMember)
    if (member(someMember).depth = 1) then
        _player.cursor(someMember)
    else
        _sound.beep()
    end if
end
```

```
// JavaScript syntax
function myCursor(someMember) {
    if (member(someMember).depth == 1) {
        _player.cursor(someMember);
    }
    else {
        _sound.beep();
    }
}
```

See also

Player

date() (formats)

Usage

```
-- Lingo syntax syntax
date({stringFormat})
date({intFormat})
date({intYearFormat, intMonthFormat, intDayFormat})

// JavaScript syntax
Date({"month dd, yyyy hh:mm:ss"});
Date({"month dd, yyyy"});
Date({yy,mm,dd,hh,mm,ss});
Date({yy,mm,dd});
Date({milliseconds});
```

Description

Top level function and data type. Creates a standard, formatted date object instance for use with other date object instances in arithmetic operations and for use in manipulating dates across platforms and in international formats.

Lingo date objects and JavaScript syntax date objects are different; therefore, Lingo date objects cannot be created using JavaScript syntax, and JavaScript syntax date objects cannot be created using Lingo syntax.

Create a new JavaScript syntax Date object using the `new Date()` syntax. Case is important in JavaScript syntax. For example, using `new date()` results in a runtime error.

When creating a date using Lingo, use four digits for the year, two digits for the month, and two digits for the day. The following expressions all return a date object equivalent to October 21, 2004.

Date Format	Usage
string	date("20041021")
integer	date(20041021)
comma separated	date(2004, 10, 21)

The individual properties of the returned date object are as follows.

Property	Description
#year	An integer representing the year
#month	An integer representing the month of the year
#day	An integer representing the day of the month

Addition and subtraction operations on the date are interpreted as the addition and subtraction of days.

Parameters

stringFormat Optional when creating a Lingo date object. A string that specifies the new date object.

intFormat Optional when creating a Lingo date object. An integer that specifies the new date object.

intYearFormat Optional when creating a Lingo date object. An integer that specifies the four-digit year of the new date object.

intMonthFormat Optional when creating a Lingo date object. An integer that specifies the two-digit month of the new date object.

intDayFormat Optional when creating a Lingo date object. An integer that specifies the two-digit day of the new date object.

month Optional when creating an JavaScript syntax Date object. A string that specifies the month of the new Date object. Valid values range from 0 (January) to 11 (December).

dd Optional when creating an JavaScript syntax Date object. A two-digit integer that specifies the day of the new Date object. Valid values range from 0 (Sunday) to 6 (Saturday).

yyyy Optional when creating an JavaScript syntax Date object. A four-digit integer that specifies the year of the new Date object.

hh Optional when creating an JavaScript syntax Date object. A two-digit integer that specifies the hour of the new Date object. Valid values range from 0 (12:00am) to 23 (11:00pm).

mm Optional when creating an JavaScript syntax Date object. A two-digit integer that specifies the minute of the new Date object. Valid values range from 0 to 59.

ss Optional when creating an JavaScript syntax Date object. A two-digit integer that specifies the seconds of the new Date object. Valid values range from 0 to 59.

yy Optional when creating an JavaScript syntax Date object. A two-digit integer that specifies the year of the new Date object. Valid values range from 0 to 99.

milliseconds Optional when creating an JavaScript syntax Date object. An integer that specifies the milliseconds of the new Date object. Valid values range from 0 to 999.

Example

These statements create and determine the number of days between two dates:

```
-- Lingo syntax syntax
myBirthday = date(19650712)
yourBirthday = date(19450529)
put("There are" && abs(yourBirthday - myBirthday) && "days between our birthdays.")

// JavaScript syntax
var myBirthday = new Date(1965, 07, 12);
var yourBirthday = new Date(1945, 05, 29);
```

```
put("There are " + Math.abs(((yourBirthday - myBirthday)/1000/60/60/24)) + " days between
our birthdays.");
```

These statements access an individual property of a date:

```
-- Lingo syntax syntax
myBirthday = date(19650712)
put("I was born in month number" && myBirthday.month)

// JavaScript syntax
var myBirthday = new Date(1965, 07, 12);
put("I was born in month number " + myBirthday.getMonth());
```

date() (System)

Usage

```
-- Lingo syntax
_system.date({yyyymmdd})

// JavaScript syntax
_system.date({yyyymmdd});
```

Description

System method; returns the current date in the system clock.

The format Director uses for the date varies, depending on how the date is formatted on the computer.

- In Windows, you can customize the date display by using the International control panel. (Windows stores the current short date format in the System.ini file. Use this value to determine what the parts of the short date indicate.)
- On the Mac, you can customize the date display by using the Date and Time control panel.

Parameters

yyyymmdd Optional. A number that specifies the four-digit year (*yyyy*), two-digit month (*mm*), and two-digit day (*dd*) of the returned date.

Example

This statement tests whether the current date is January 1 by checking whether the first four characters of the date are 1/1. If it is January 1, the alert "Happy New Year!" appears:

```
-- Lingo syntax
if (_system.date().char[1..4] = "1/1/") then
    _player.alert("Happy New Year!")
end if

// JavaScript syntax
if (_system.date().toString().substr(0, 4) == "1/1/") {
    _player.alert("Happy New Year!");
}
```

See also

System

delay()

Usage

```
-- Lingo syntax
_movie.delay(intTicks)

// JavaScript syntax
_movie.delay(intTicks);
```

Description

Movie method; pauses the playhead for a given amount of time.

The only mouse and keyboard activity possible during this time is stopping the movie by pressing Control+Alt+period (Windows) or Command+period (Mac). Because it increases the time of individual frames, `delay()` is useful for controlling the playback rate of a sequence of frames.

The `delay()` method can be applied only when the playhead is moving. However, when `delay()` is in effect, handlers still run; only the playhead halts, not script execution. Place scripts that use `delay()` in either an `enterFrame` or `exitFrame` handler.

To mimic the behavior of a halt in a handler when the playhead is not moving, use the `milliseconds` property of the System object and wait for the specified amount of time to pass before exiting the frame.

Parameters

`intTicks` Required. An integer that specifies the number of ticks to pause the playhead. Each tick is 1/60 of a second.

Example

This handler delays the movie for 2 seconds when a key is pressed:

```
-- Lingo syntax
on keyDown
    _movie.delay(2*60)
end

// JavaScript syntax
function keyDown() {
    _movie.delay(2*60)
}
```

This handler, which can be placed in a frame script, delays the movie a random number of ticks:

```
-- Lingo syntax
on keyDown
    if (_key.key = "x") then
        _movie.delay(random(180))
    end if
end

// JavaScript syntax
function keyDown() {
    if (_key.key == "x") {
        _movie.delay(random(180));
    }
}
```

See also

endFrame, milliseconds, Movie

delete()

Usage

```
delete chunkExpression
```

Description

Chunk expressions are any character, word, item, or line in a container of characters.

Parameters

None.

See also

Fileio

Example

```
delete member(1).line[1]
```
This statement deletes the line 1 in the text member.

```
delete member(1).word[1]
```
This statement deletes the word 1 in the text member.

```
delete member(1).char[1]
```
This statement deletes the character 1 in the text member.

deleteFile()

Usage

```
-- Lingo syntax
fileioObjRef.deleteFile()

// JavaScript syntax
fileioObjRef.deleteFile();
```

Description

Fileio method; Deletes a file.

Parameters

None.

See also

Fileio

deleteAt

Usage

```
list.deleteAt(number)
deleteAt list, number
```

Description

List command; deletes an from a linear or property list.

The `deleteAt` command checks whether an item is in a list; if you try to delete an object that isn't in the list, Director displays an alert.

Parameters

`number` Required. Specifies the position of the item in the list to delete.

Example

This statement deletes the second item from the list named designers, which contains [gee, kayne, ohashi]:

```
--Lingo
designers = ["gee", "kayne", "ohashi"]
designers.deleteAt(2)

// Javascript
Designers = list("gee","kayne","ohashi");
Designers.deleteAt(2);
```

The result is the list [gee, ohashi].

This handler checks whether an object is in a list before attempting to delete it:

```
on myDeleteAt theList, theIndex
    if theList.count < theIndex then
        beep
    else
        theList.deleteAt(theIndex)
    end if
end
```

See also

`addAt`

deleteCamera

Usage

```
member(whichCastmember).deleteCamera(cameraName)
member(whichCastmember).deleteCamera(index)
sprite(whichSprite).deleteCamera(cameraOrIndex)
```

Description

3D command; in a cast member, this command removes the camera from the cast member and the 3D world. Children of the camera are removed from the 3D world but not deleted.

It is not possible to delete the default camera of the cast member.

In a sprite, this command removes the camera from the sprite's list of cameras. The camera is not deleted from the cast member.

Parameters

`cameraNameOrNum` Required. A string or an integer that specifies the name or index position of the camera to delete.

Example

This statement deletes two cameras from the cast member named Room: first the camera named Camera06, and then camera 1.

```
member("Room").deleteCamera("Camera06")
member("Room").deleteCamera(1)
```

This statement removes two cameras from the list of cameras for sprite 5: first the second camera in the list, then the camera named Camera06

```
sprite(5).deleteCamera(2)
sprite(5).deleteCamera(member("Room").camera("Camera06"))
```

See also

newCamera, addCamera, cameraCount()

deleteFrame()

Usage

```
-- Lingo syntax
_movie.deleteFrame()

// JavaScript syntax
_movie.deleteFrame();
```

Description

Movie method; deletes the current frame and makes the next frame the new current frame during a Score generation session only.

Parameters

None.

Example

The following handler checks whether the sprite in channel 10 of the current frame has gone past the right edge of a 640-by-480-pixel Stage and deletes the frame if it has:

```
-- Lingo syntax
on testSprite
    _movie.beginRecording()
    if (sprite(10).locH > 640) then
        _movie.deleteFrame()
    end if
    _movie.endRecording()
end

// JavaScript syntax
function testSprite() {
    _movie.beginRecording();
    if (sprite(10).locH > 640) {
        _movie.deleteFrame();
    }
    _movie.endRecording();
}
```

See also

beginRecording(), endRecording(), Movie, updateFrame()

deleteGroup

Usage

```
member(whichCastmember).deleteGroup(whichGroup)
member(whichCastmember).deleteGroup(index)
```

Description

3D command; removes the group from the cast member and the 3D world. Children of the group are removed from the 3D world but not deleted.

It is not possible to delete the group named World, which is the default group.

Parameters

groupNameOrNum Required. A string or integer that specifies the name or index position of the group to delete.

Example

The first line of this example deletes the group Dummy16 from the cast member Scene. The second line deletes the third group of Scene.

```
member("Scene").deleteGroup("Dummy16")
member("Scene").deleteGroup(3)
```

See also

newGroup, child (3D), parent

deleteLight

Usage

```
member(whichCastmember).deleteLight(whichLight)
member(whichCastmember).deleteLight(index)
```

Description

3D command; removes the light from the cast member and the 3D world. Children of the light are removed from the 3D world but not deleted.

Parameters

lightNameOrNum Required. A string or integer that specifies the name or index position of the light to delete.

Example

These examples delete lights from the cast member named Room.

```
member("Room").deleteLight("ambientRoomLight")
member("Room").deleteLight(6)
```

See also

newLight

deleteModel

Usage

```
member(whichCastmember).deleteModel(whichModel)
member(whichCastmember).deleteModel(index)
```

Description

3D command; removes the model from the cast member and the 3D world. Children of the model are removed from the 3D world but not deleted.

Parameters

modelNameOrNum Required. A string or integer that specifies the name or index position of the model to delete.

Example

The first line of this example deletes the model named Player3 from the cast member named gbWorld. The second line deletes the ninth model of gbWorld.

```
member("gbWorld").deleteModel("Player3")
member("gbWorld").deleteModel(9)
```

See also

newModel

deleteModelResource

Usage

```
member(whichCastmember).deleteModelResource(whichModelResource)
member(whichCastmember).deleteModelResource(index)
```

Description

3D command; removes the model resource from the cast member and the 3D world.

Models using the deleted model resource become invisible, because they lose their geometry, but they are not deleted or removed from the world.

Parameters

resourceNameOrNum Required. A string or integer that specifies the name or index position of the model resource to delete.

Example

These examples delete two model resources from the cast member named StreetScene.

```
member("StreetScene").deleteModelResource("HouseB")
member("StreetScene").deleteModelResource(3)
```

See also

newModelResource, newMesh

deleteMotion

Usage

```
member(whichCastmember).deleteMotion(whichMotion)
member(whichCastmember).deleteMotion(index)
```

Description

3D command; removes the motion from the cast member.

Parameters

motionNameOrNum Required. A string or integer that specifies the name or index position of the motion to delete.

Example

The first line of this example deletes the motion named BackFlip from the cast member named PicnicScene. The second line deletes the fifth motion in PicnicScene.

```
member("PicnicScene").deleteMotion("BackFlip")
member("PicnicScene").deleteMotion(5)
```

See also

newMotion(), removeLast()

deleteOne

Usage

```
list.deleteOne(value)
deleteOne list, value
```

Description

List command; deletes a value from a linear or property list. For a property list, `deleteOne` also deletes the property associated with the deleted value. If the value appears in the list more than once, `deleteOne` deletes only the first occurrence.

Attempting to delete a property has no effect.

When you add a filter using the add or append method of the filterlist, a duplicate is created and added to the list. Methods such as deleteOne, getPos, findPos, and getOne use the exact value in the list and not the duplicate value.

In such cases, you can use the deleteOne command, as follows:

```
f = filter(#glowfilter)
sprite(1).filterlist.append(f)
f = sprite(1).filterlist[1]-- here we get the actual value added to the list.
sprite(1).filterlist.deleteOne(f)
```

The third line in the script adds the reference of the filter value to the list.

Parameters

value Required. The value to delete from the list.

Example

The first statement creates a list consisting of the days Tuesday, Wednesday, and Friday. The second statement deletes the name Wednesday from the list.

```
--Lingo
days = ["Tuesday", "Wednesday", "Friday"]
days.deleteOne("Wednesday")
put days

// Javascript
days = list("Tuesday", "Wednesday", "Friday");
days.deleteOne("Wednesday");
trace(days);
```

The `put days` statement causes the Message window to display the result:

```
-- ["Tuesday", "Friday"].
```

deleteProp

Usage

```
list.deleteProp(item)
deleteProp list, item
```

Description

List command; deletes the specified item from the specified list.

- For linear lists, replace *item* with the number identifying the list position of the item to be deleted. The `deleteProp` command for linear lists is the same as the `deleteAt` command. If the number is greater than the number of items in the list, a script error occurs.
- For property lists, replace *item* with the name of the property to be deleted. Deleting a property also deletes its associated value. If the list has more than one of the same property, only the first property in the list is deleted.

Parameters

item Required. The item to delete from the list.

Example

This statement deletes the `color` property from the list [#height:100, #width: 200, #color: 34, #ink: 15], which is called spriteAttributes:

```
spriteAttributes.deleteProp(#color)
```

The result is the list [#height:100, #width: 200, #ink: 15].

See also

deleteAt

deleteShader

Usage

```
member(whichCastmember).deleteShader(whichShader)
```

```
member(whichCastmember).deleteShader(index)
```

Description

3D command; removes the shader from the cast member.

Parameters

shaderNameOrNum Required. A string or integer that specifies the name or index position of the shader to delete.

Example

The first line of this example deletes the shader Road from the cast member named StreetScene. The second line deletes the third shader of StreetScene.

```
-- Lingo
member("StreetScene").deleteShader("Road")
member("StreetScene").deleteShader(3)

// Javascript
member("StreetScene").deleteShader("Road");
member("StreetScene").deleteShader(3);
```

See also

newShader, shaderList

deleteTexture

Usage

```
member(whichCastmember).deleteTexture(whichTexture)
member(whichCastmember).deleteTexture(index)
```

Description

3D command; removes the texture from the cast member.

Parameters

textureNameOrNum Required. A string or integer that specifies the name or index position of the texture to delete.

Example

The first line of this example deletes the texture named Sky from the cast member named PicnicScene. The second line deletes the fifth texture of PicnicScene.

```
-- Lingo
member("PicnicScene).deleteTexture("Sky")
member("PicnicScene").deleteTexture(5)

// Javascript
member("PicnicScene).deleteTexture("Sky");
member("PicnicScene").deleteTexture(5)
```

See also

newTexture

deleteVertex()

Usage

```
-- Lingo syntax
memberObjRef.deleteVertex(indexToRemove)

// JavaScript syntax
memberObjRef.deleteVertex(indexToRemove);
```

Description

Vector shape command; removes an existing vertex of a vector shape cast member in the index position specified.

Parameters

indexToRemove Required. An integer that specifies the index position of the vertex to delete.

Example

This line removes the second vertex point in the vector shape Archie:

```
-- Lingo syntax
member("Archie").deleteVertex(2)

// JavaScript syntax
member("Archie").deleteVertex(2);
```

See also

addVertex(), moveVertex(), originMode, vertexList

displayOpen()

Usage

```
-- Lingo syntax
fileioObjRef.displayOpen()

// JavaScript syntax
fileioObjRef.displayOpen();
```

Description

Fileio method; Displays an Open dialog box.

This method returns to script the full path and name of the selected file.

Parameters

None.

See also

Fileio

displaySave()

Usage

```
-- Lingo syntax
fileioObjRef.displaySave(stringTitle, stringFileName)

// JavaScript syntax
fileioObjRef.displaySave(stringTitle, stringFileName);
```

Description

Fileio method; Displays a Save dialog box.

This method returns to script the full path and name of the saved file.

Parameters

`stringTitle` Required. A string that specifies the title displayed in the Save dialog box.

`stringFileName` Required. A string that specifies the full path and name of the file to save.

See also

`Fileio`

do

Usage

```
do stringExpression
```

Description

Command; evaluates a string and executes the result as a script statement. This command is useful for evaluating expressions that the user has typed and for executing commands stored in string variables, fields, arrays, and files.

Using uninitialized local variables within a `do` command creates a compile error. Initialize any local variables in advance.

Note: *This command does not allow global variables to be declared; these variables must be declared in advance.*

The `do` command works with multiple-line strings as well as single lines.

Parameters

`stringExpression` Required. The string to be evaluated.

Example

This statement performs the statement contained within quotation marks:

```
do "beep 2"
do commandList[3]
```

doneParsing()

Usage

```
parserObject.doneParsing()
```

Description

Function; returns `1 (TRUE)` when the parser has completed parsing a document using `parseURL()`. The return value is `0 (FALSE)` until the parsing is complete.

Parameters

None.

See also

`parseURL()`

dot()

Usage

```
vector1.dot(vector2)
```

Description

3D vector method; returns the sum of the products of the x, y, and z components of two vectors. If both vectors are normalized, the `dot` is the cosine of the angle between the two vectors.

To manually arrive at the dot of two vectors, multiply the x component of `vector1` by the x component of `vector2`, then multiply the y component of `vector1` by the y component of `vector2`, then multiply the z component of `vector1` by the z component of `vector2`, and finally add the three products together.

This method is identical to `dotProduct()` function.

Parameters

`vector2` Required. The second vector from which a sum is returned.

Example

In this example, the angle between the vectors `pos5` and `pos6` is 45 degrees. The getNormalized function returns the normalized values of pos5 and pos6, and stores them in the variables `norm1` and `norm2`. The dot of `norm1` and `norm2` is 0.7071, which is the cosine of 45 degrees.

```
pos5 = vector(100, 100, 0)
pos6 = vector(0, 100, 0)
put pos5.angleBetween(pos6)
-- 45.0000
norm1 = pos5.getNormalized()
put norm1
-- vector( 0.7071, 0.7071, 0.0000 )
norm2 = pos6.getNormalized()
put norm2
-- vector( 0.0000, 1.0000, 0.0000 )
put norm1.dot(norm2)
-- 0.7071
```

See also

dotProduct(), getNormalized, normalize

dotProduct()

Usage

```
vector1.dotProduct(vector2)
```

Description

3D vector method; returns the sum of the products of the x, y, and z components of two vectors. If both vectors are normalized, the dotproduct is the cosine of the angle between the two vectors.

To manually arrive at the dot of two vectors, multiply the x component of vector1 by the x component of vector2, then multiply the y component of vector1 by the y component of vector2, then multiply the z component of vector1 by the z component of vector2, and finally add the three products together.

This method is identical to dot() function.

Parameters

vector2 Required. The second vector from which a sum is returned.

Example

In this example, the angle between the vectors pos5 and pos6 is 45°. The getNormalized function returns the normalized values of pos5 and pos6, and stores them in the variables norm1 and norm2. The dotProduct of norm1 and norm2 is 0.7071, which is the cosine of 45°.

```
pos5 = vector(100, 100, 0)
pos6 = vector(0, 100, 0)
put pos5.angleBetween(pos6)
-- 45.0000
norm1 = pos5.getNormalized()
put norm1
-- vector( 0.7071, 0.7071, 0.0000 )
norm2 = pos6.getNormalized()
put norm2
-- vector( 0.0000, 1.0000, 0.0000 )
put norm1.dotProduct(norm2)
-- 0.7071
```

See also

dot(), getNormalized, normalize

downloadNetThing

Usage

```
downloadNetThing URL, localFile
```

Description

Command; copies a file from the Internet to a file on the local disk, while the current movie continues playing. Use netDone to find out whether downloading is finished.

Director movies in authoring mode and projectors support the `downLoadNetThing` command, but the Shockwave Player does not. This protects users from unintentionally copying files from the Internet.

Although many network operations can be active at one time, running more than four concurrent operations usually slows down performance unacceptably.

Neither the Director movie's cache size nor the setting for the Check Documents option affects the behavior of the `downloadNetThing` command.

Parameters

`URL` Required. The URL of any object that can be downloaded: for example, an FTP or HTTP server, an HTML page, an external cast member, a Director movie, or a graphic.

`localFile` Required. The pathname and filename for the file on the local disk.

Example

These statements download an external cast member from a URL to the Director application folder and then make that file the external cast member named Cast of Thousands:

```
downLoadNetThing("http://www.cbDeMille.com/Thousands.cst",
the\applicationPath&"Thousands.cst")
castLib("Cast of Thousands").fileName = the applicationPath&"Thousands.cst"
```

See also

`importFileInto()`, `netDone()`, `preloadNetThing()`

draw()

Usage

```
-- Lingo syntax
imageObjRef.draw(x1, y1, x2, y2, colorObjOrParamList)
imageObjRef.draw(point(x, y), point(x, y), colorObjOrParamList)
imageObjRef.draw(rect, colorObjOrParamList)

// JavaScript syntax
imageObjRef.draw(x1, y1, x2, y2, colorObjOrParamList);
imageObjRef.draw(point(x, y), point(x, y), colorObjOrParamList);
imageObjRef.draw(rect, colorObjOrParamList);
```

Description

Image method. Draws a line or an unfilled shape with a specified color in a rectangular region of a given image object.

This method returns a value of 1 if there is no error.

If the optional parameter list is not provided, `draw()` draws a 1-pixel line between the first and second points given or between the upper left and lower right corners of the given rectangle.

For best performance, with 8-bit or lower images the color object should contain an indexed color value. For 16- or 32-bit images, use an RGB color value.

To fill a solid region, use the `fill()` method.

Parameters

x1 Required if drawing a line using x and y coordinates. An integer that specifies the x coordinate of the start of the line.

y1 Required if drawing a line using x and y coordinates. An integer that specifies the y coordinate of the start of the line.

x2 Required if drawing a line using x and y coordinates. An integer that specifies the x coordinate of the end of the line.

y2 Required if drawing a line using x and y coordinates. An integer that specifies the y coordinate of the end of the line.

colorObjOrParamList Required. A color object or parameter list that specifies the color of the line or shape's border. The parameter list can be used instead of a simple color object to specify the following properties.

Property	Description
#shapeType	A symbol value of `#oval`, `#rect`, `#roundRect`, or `#line`. The default is #line.
#lineSize	The width of the line to use in drawing the shape.
#color	A color object, which determines the color of the shape's border.

`point(x, y), point(x, y)` Required if drawing a line using points. Two points that specify the start and end points of the line.

rect Required if drawing a shape. A rectangle that specifies the rectangular region in which a shape is drawn.

Example

This statement draws a 1-pixel, dark red, line from point (20, 20) to point (20, 60) within the image of the stage.

```
-- Lingo
objImage = _movie.stage.image
objImage.draw(point(20, 20), point(20, 60), rgb(255, 0 ,0))

// Javascript
var objImage = _movie.stage.image ;
objImage.draw(point(20, 20), point(20, 60), color(255, 0 ,0)) ;
```

See also

`color()`, `copyPixels()`, `fill()`, `image()`, `setPixel()`

duplicate() (Image)

Usage

```
-- Lingo syntax
imageObjRef.duplicate()

// JavaScript syntax
imageObjRef.duplicate();
```

Description

Image method. Creates and returns a copy of a given image.

The new image is completely independent of the original, and is not linked to any cast member. If planning to make a lot of changes to an image, it is better to make a copy that is independent of a cast member.

Parameters

None.

Example

This statement creates a new image object from the image of cast member Lunar Surface and places the new image object into the variable `workingImage`:

```
workingImage = member("Lunar Surface").image.duplicate()
```

See also

`image()`

duplicate() (list function)

Usage

```
(oldList).duplicate()
duplicate(oldList)
```

Description

List function; returns a copy of a list and copies nested lists (list items that also are lists) and their contents. The function is useful for saving a list's current content.

Note: *filterlist() cannot be duplicated using this function.*

When you assign a list to a variable, the variable contains a reference to the list, not the list itself. This means any changes to the copy also affect the original list.

To see an example of `duplicate() (list function)` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo folder inside the Director application folder.

Parameters

`oldList` Required. Specifies the list to duplicate.

Example

This statement makes a copy of the list CustomersToday and assigns it to the variable `CustomerRecord`:

```
-- Lingo
CustomersToday = [1, 3, 5]
CustomerRecord = CustomersToday.duplicate()

// Javascript

CustomersToday = list(1, 3, 5);
CustomerRecord = CustomersToday.duplicate();
```

See also

`image()`

duplicate() (Member)

Usage

```
-- Lingo syntax
memberObjRef.duplicate({intPosn})

// JavaScript syntax
memberObjRef.duplicate({intPosn});
```

Description

Member method; makes a copy of a specified cast member.

This method is best used during authoring rather than during runtime; it creates another cast member in memory, which could result in memory problems.

Use this method to permanently save cast member changes with the file.

Parameters

intPosn Optional. An integer that specifies the Cast window for the duplicate cast member. If omitted, the duplicate cast member is placed in the first open Cast window position.

Example

This statement makes a copy of cast member Desk and places it in the first empty Cast window position:

```
-- Lingo syntax
member("Desk").duplicate()

// JavaScript syntax
member("Desk").duplicate();
```

This statement makes a copy of cast member Desk and places it in the Cast window at position 125:

```
-- Lingo syntax
member("Desk").duplicate(125)

// JavaScript syntax
member("Desk").duplicate(125);
```

See also

Member

duplicateFrame()

Usage

```
-- Lingo syntax
_movie.duplicateFrame()

// JavaScript syntax
_movie.duplicateFrame();
```

Description

Movie method; duplicates the current frame and its content, inserts the duplicate frame after the current frame, and then makes the duplicate frame the current frame. This method can be used during Score generation only.

This method performs the same function as the `insertFrame()` method.

Parameters

None.

Example

When used in the following handler, the `duplicateFrame` command creates a series of frames that have cast member Ball in the external cast Toys assigned to sprite channel 20. The number of frames is determined by the argument `numberOfFrames`.

```
-- Lingo syntax
on animBall(numberOfFrames)
    _movie.beginRecording()
    sprite(20).member = member("Ball", "Toys")
    repeat with i = 0 to numberOfFrames
        _movie.duplicateFrame()
    end repeat
    _movie.endRecording()
end animBall

// JavaScript syntax
function animBall(numberOfFrames) {
    _movie.beginRecording();
    sprite(20).member = member("Ball", "Toys");
    for (var i = 0; i <= numberOfFrames; i++) {
        _movie.duplicateFrame();
    }
    _movie.endRecording();
}
```

See also

`insertFrame()`, `Movie`

enableHotSpot()

Usage

```
-- Lingo syntax
spriteObjRef.enableHotSpot(hotSpotID, trueOrFalse)

// JavaScript syntax
spriteObjRef.enableHotSpot(hotSpotID, trueOrFalse);
```

Description

QTVR (QuickTime® VR) command; determines whether a hot spot in a QTVR sprite is enabled (`TRUE`), or disabled (`FALSE`).

Parameters

`hotSpotID` Required. Specifies the hot spot in the QTVR sprite to test.

`trueOrFalse` Required. A `TRUE` or `FALSE` value that specifies whether the QTVR sprite is enabled.

endRecording()

Usage

```
-- Lingo syntax
_movie.endRecording()

// JavaScript syntax
_movie.endRecording();
```

Description

Movie method; ends a Score update session.

You can resume control of Score channels through scripting after calling `endRecording()`.

Parameters

None.

Example

When used in the following handler, the `endRecording` keyword ends the Score generation session:

```
-- Lingo syntax
on animBall(numberOfFrames)
    _movie.beginRecording()
    horizontal = 0
    vertical = 100
    repeat with i = 1 to numberOfFrames
        _movie.go(i)
        sprite(20).member = member("Ball")
        sprite(20).locH = horizontal
        sprite(20).locV = vertical
        sprite(20).foreColor = 255
        horizontal = horizontal + 3
        vertical = vertical + 2
        _movie.updateFrame()
    end repeat
    _movie.endRecording()
end animBall

// JavaScript syntax
function animBall(numberOfFrames) {
    _movie.beginRecording();
    var horizontal = 0;
    var vertical = 100;
    for (var i = 1; i <= numberOfFrames; i++) {
        _movie.go(1);
        sprite(20).member = member("Ball");
        sprite(20).locH = horizontal;
        sprite(20).locV = vertical;
        sprite(20).foreColor = 255;
        horizontal = horizontal + 3;
        vertical = vertical + 2;
        _movie.updateFrame();
    }
    _movie.endRecording();
}
```

See also

beginRecording(), Movie, updateFrame()

erase()

Usage

```
-- Lingo syntax
memberObjRef.erase()

// JavaScript syntax
memberObjRef.erase();
```

Description

Member method; deletes a specified cast member and leaves its slot in the Cast window empty.

For best results, use this method during authoring and not in projectors. Using this method in projectors may cause memory problems.

Parameters

None.

Example

This statement deletes the cast member named Gear in the Hardware cast:

```
-- Lingo syntax
member("Gear", "Hardware").erase()

// JavaScript syntax
member("Gear", "Hardware").erase();
```

This handler deletes cast members numbered from `start` through `finish`:

```
-- Lingo syntax
on deleteMember start, finish
    repeat with i = start to finish
        member(i).erase()
    end repeat
end deleteMember

// JavaScript syntax
function deleteMember(start, finish) {
    for (var i=start; i<=finish; i++) {
        member(i).erase();
    }
}
```

See also

Member, new()

error()

Usage

```
-- Lingo syntax
fileioObjRef.error(intError)

// JavaScript syntax
fileioObjRef.error(intError);
```

Description

Fileio method; Returns a specified error message.

Parameters

intError Required. An integer that specifies the error. Valid values include 0 ("OK") or 1 ("Memory allocation failure"). All other values return "Unknown error".

Example

This following example creates a file called c:\check.txt. If there is any error in creating the file it would give that error.

```
-- Lingo syntax
objFileio = new xtra("fileio")
isok = objFileio.createFile("c:\check.txt")
if ( isok = 0 ) then
    alert( " file creation successful")
else
    alert( " file creation failed " & objFileio.error(isok))
end if

// JavaScript syntax
var objFileio = new xtra("fileio");
isok = objFileio.createFile("c:\check.txt");
if ( isok == 0 )
{
    _player.alert( " file creation successful");
}
else
{
    _player.alert( " file creation failed " + objFileio.error(isok));
}
```

See also

Fileio

externalEvent()

Usage

```
externalEvent "string"
```

Description

Command; sends a string to the browser that the browser can interpret as a scripting language instruction, allowing a movie playing or a browser to communicate with the HTML page in which it is embedded.

This command works only for movies in browsers.

Note: *The `externalEvent` command does not produce a return value. There is no immediate way to determine whether the browser handled the event or ignored it. Use `on EvalScript` within the browser to return a message to the movie.*

Parameters

string Required. The string to send to the browser. This string must be in a scripting language supported by the browser.

Example

The following statements use `externalEvent` in the LiveConnect scripting environment, which is supported by Netscape 3.x and later.

LiveConnect evaluates the string passed by `externalEvent` as a function call. JavaScript authors must define and name this function in the HTML header. In the movie, the function name and parameters are defined as a string in `externalEvent`. Because the parameters must be interpreted by the browser as separate strings, each parameter is surrounded by single quotation marks.

Statements within HTML:

```
function MyFunction(parm1, parm2) {
    //script here
}
```

Statements within a script in the movie:

```
externalEvent ("MyFunction('parm1','parm2')")
```

The following statements use `externalEvent` in the ActiveX scripting environment used by Internet Explorer® in Windows®. ActiveX treats `externalEvent` as an event and processes this event and its string parameter the same as an `onClick` event in a button object.

- Statements within HTML:

```
Sub
NameOfShockwaveInstance_externalEvent(aParam)
    'script here
End Sub
```

Alternatively, define a script for the event:

```
<SCRIPT FOR="NameOfShockwaveInstance"
EVENT="externalEvent(aParam)"
LANGUAGE="VBScript">
 'script here
</SCRIPT>
```

Within the movie, include the function and any parameters as part of the string for `externalEvent`:

```
externalEvent ("MyFunction ('parm1','parm2')")
```

See also

`on EvalScript`

extrude3D

Usage

```
member(whichTextCastmember).extrude3D(member(which3dCastmember))
```

Description

3D command; creates a new `#extruder` model resource in a 3D cast member from the text in a text cast member.

This is not the same as using the 3D `displayMode` property of a text cast member.

To create a model using extrude3D:

1 Create a new `#extruder` model resource in a 3D cast member:

```
textResource = member("textMember").extrude3D(member("3DMember"))
```

2 Create a new model using the model resource created in step 1:

```
member("3DMember").newModel("myText", textResource)
```

Parameters

which3dCastmember Required. The cast member within which a new `#extruder` model resource is created.

Example

In this example, Logo is a text cast member and Scene is a 3D cast member. The first line creates a model resource in Scene which is a 3D version of the text in Logo. The second line uses this model resource to create a model named 3dLogo.

```
-- Lingo
myTextModelResource =member("Logo").extrude3d(member("Scene"))
member("Scene").newModel("3dLogo", myTextModelResource)

// Javascript
myTextModelResource =member("Logo").extrude3d(member("Scene"));
member("Scene").newModel("3dLogo", myTextModelResource);
```

See also

bevelDepth, bevelType, displayFace, smoothness, tunnelDepth, displayMode

externalParamName()

Usage

```
-- Lingo syntax
_player.externalParamName(paramNameOrNum)

// JavaScript syntax
_player.externalParamName(paramNameOrNum);
```

Description

Player method; returns the name of a specified parameter in the list of external parameters from an HTML <EMBED> or <OBJECT> tag.

If specifying a parameter by name, this method returns any parameter names that matches *paramNameOrNum*. The match is not case sensitive. If no matching parameter name is found, this method returns `VOID` (Lingo) or `null` (JavaScript syntax).

If specifying a parameter by number, this method returns the parameter name at the *paramNameOrNum* position in the parameter list. If no matching parameter position is found, this method returns `VOID` or `null`.

This method is valid only for movies with Shockwave content that are running in a browser. It cannot be used with Director movies or projectors.

The following list describes the pre-defined external parameters that can be used.

Parameter	Definition
swAudio	A string that specifies the location of a Shockwave Audio file to be played with the movie. The value is a fully qualified URL.
swBackColor	A color value intended to modify the movie's Stage color property. The value is any integer value from 0 to 255. Use 0 to 255 for movies in 8-bit color, and 0 to 15 for movies in 4-bit color.
swBanner	A string that specifies the text to be used as a banner in the movie.
swColor	A color value for use in modifying the color of a specific object. The value is any integer from 0 to 255. Use 0 to 255 for movies in 8-bit color, and 0 to 15 for movies in 4-bit color.
swForeColor	A new foreground color value. Text written into field cast members is rendered in the currently active foreground color. The value is any integer value from 0 to 255. Use 0 to 255 for movies in 8-bit color, and 0 to 15 for movies in 4-bit color.
swFrame	A string value that is the name assigned to a given frame in the movie.
swList	A comma-delimited list of items that can be parsed with script. List values may be key/value pairs, Boolean items, integers, or strings.
swName	A name, such as a user name, to be displayed or used within the movie.
swPassword	A password, perhaps for use it conjunction with the `swName` property, to be used within the movie.
swPreloadTime	An integer value which specifies the number of seconds of an audio file sound that should be preloaded before the sound begins to play. Used with Shockwave Audio to improve playback performance by increasing the amount of audio already downloaded before playback begins.
swSound	A string value which may specify the name of a sound in the Director movie to be played, or whether or not a sound should be played at all.
swText	A string value that specifies text to be used in the movie.
swURL	A string URL that may specify the location of another movie with Shockwave content or Shockwave Audio file.
swVolume	An integer value (0 to 10 is recommended) that is used to control the volume level of the sound output from the movie. 0 is off (no sound), 10 is maximum volume.
`sw1` through `sw9`	Nine additional properties for author-defined parameters.

Parameters

`paramNameOrNum` Required. A string that specifies the name of the parameter name to return, or an integer that specifies the index location of the parameter name to return.

Example

This statement places the value of a given external parameter in the variable `myVariable`:

```
-- Lingo syntax
if (_player.externalParamName("swURL") = "swURL") then
    myVariable = _player.externalParamValue("swURL")
end if

// JavaScript syntax
if (_player.externalParamName("swURL") == "swURL") {
    var myVariable = _player.externalParamName("swURL");
}
```

See also

`externalParamValue()`, `Movie`

externalParamValue()

Usage

```
-- Lingo syntax
_player.externalParamValue(paramNameOrNum)

// JavaScript syntax
_player.externalParamValue(paramNameOrNum);
```

Description

Returns the value of a specified parameter in the list of external parameters from an HTML <EMBED> or <OBJECT> tag.

If specifying a parameter value by name, this method returns the value of the first parameter whose name matches *paramNameOrNum*. The match is not case sensitive. If no matching parameter value is found, this method returns `VOID` (Lingo) or `null` (JavaScript syntax).

If specifying a parameter value by index, this method returns the value of the parameter at the *paramNameOrNum* position in the parameter list. If no matching parameter position is found, this method returns `VOID` or `null`.

This method is valid only for movies with Shockwave content that are running in a browser. It cannot be used with Director movies or projectors.

The following list describes the pre-defined external parameters that can be used.

Parameter	Definition
swAudio	A string that specifies the location of a Shockwave Audio file to be played with the movie. The value is a fully qualified URL.
swBackColor	A color value intended to modify the movie's Stage color property. The value is any integer value from 0 to 255. Use 0 to 255 for movies in 8-bit color, and 0 to 15 for movies in 4-bit color.
swBanner	A string that specifies the text to be used as a banner in the movie.
swColor	A color value for use in modifying the color of a specific object. The value is any integer from 0 to 255. Use 0 to 255 for movies in 8-bit color, and 0 to 15 for movies in 4-bit color.
swForeColor	A new foreground color value. Text written into field cast members is rendered in the currently active foreground color. The value is any integer value from 0 to 255. Use 0 to 255 for movies in 8-bit color, and 0 to 15 for movies in 4-bit color.
swFrame	A string value that is the name assigned to a given frame in the movie.
swList	A comma-delimited list of items that can be parsed with script. List values may be key/value pairs, Boolean items, integers, or strings.
swName	A name, such as a user name, to be displayed or used within the movie.
swPassword	A password, perhaps for use it conjunction with the `swName` property, to be used within the movie.
swPreloadTime	An integer value which specifies the number of seconds of an audio file sound that should be preloaded before the sound begins to play. Used with Shockwave Audio to improve playback performance by increasing the amount of audio already downloaded before playback begins.
swSound	A string value which may specify the name of a sound in the Director movie to be played, or whether or not a sound should be played at all.
swText	A string value that specifies text to be used in the movie.

Parameter	Definition
swURL	A string URL that may specify the location of another Shockwave movie or Shockwave Audio file.
swVolume	An integer value (0 to 10 is recommended) that is used to control the volume level of the sound output from the movie. 0 is off (no sound), 10 is maximum volume.
`sw1` through `sw9`	Nine additional properties for author-defined parameters.

Parameters

`paramNameOrNum` Required. A string that specifies the name of the parameter value to return, or an integer that specifies the index location of the parameter value to return.

Example

This statement places the value of a given external parameter in the variable `myVariable`:

```
-- Lingo syntax
if (_player.externalParamName("swURL") = "swURL") then
    myVariable = _player.externalParamValue("swURL")
end if

// JavaScript syntax
if (_player.externalParamName("swURL") == "swURL") {
    var myVariable = _player.externalParamName("swURL");
}
```

See also

`externalParamName()`, `Movie`

extractAlpha()

Usage

```
imageObject.extractAlpha()
```

Description

This function copies the alpha channel from the given 32-bit image and returns it as a new image object. The result is an 8-bit grayscale image representing the alpha channel.

This function is useful for down-sampling 32-bit images with alpha channels.

Example

This statement places the alpha channel of the image of member 1 into the variable `mainAlpha`:

```
-- Lingo
mainAlpha = member(1).image.extractAlpha()

// Javascript
mainAlpha = member(1).image.extractAlpha();
```

See also

`setAlpha()`

fadeIn()

Usage

```
-- Lingo syntax
soundChannelObjRef.fadeIn({intMilliseconds})

// JavaScript syntax
soundChannelObjRef.fadeIn({intMilliseconds});
```

Description

Sound Channel method; immediately sets the `volume` of a sound channel to zero and then brings it back to the current volume over a given number of milliseconds.

The current pan setting is retained for the entire fade.

Parameters

`intMilliseconds` Optional. An integer that specifies the number of milliseconds over which the volume is increased back to its original value. The default is 1000 milliseconds (1 second) if no value is given.

Example

This Lingo fades in sound channel 3 over a period of 3 seconds from the beginning of cast member `introMusic2`:

```
-- Lingo syntax
sound(3).play(member("introMusic2"))
sound(3).fadeIn(3000)

// JavaScript syntax
sound(3).play(member("introMusic2"));
sound(3).fadeIn(3000);
```

See also

`fadeOut()`, `fadeTo()`, `pan`, `Sound Channel`, `volume (Windows Media)`

fadeOut()

Usage

```
-- Lingo syntax
soundChannelObjRef.fadeOut({intMilliseconds})

// JavaScript syntax
soundChannelObjRef.fadeOut({intMilliseconds});
```

Description

Sound Channel method; gradually reduces the `volume` of a sound channel to zero over a given number of milliseconds.

The current pan setting is retained for the entire fade.

Parameters

`intMilliseconds` Optional. An integer that specifies the number of milliseconds over which the volume is reduced to zero. The default is 1000 milliseconds (1 second) if no value is given.

Example

This statement fades out sound channel 3 over a period of 5 seconds:

```
-- Lingo syntax
sound(3).fadeOut(5000)

// JavaScript syntax
sound(3).fadeOut(5000);
```

See also

fadeIn(), fadeTo(), pan, Sound Channel, volume (Windows Media)

fadeTo()

Usage

```
-- Lingo syntax
soundChannelObjRef.fadeTo(intVolume {, intMilliseconds})

// JavaScript syntax
soundChannelObjRef.fadeTo(intVolume {, intMilliseconds});
```

Description

Sound Channel method; gradually changes the `volume` of a sound channel to a specified volume over a given number of milliseconds.

The current pan setting is retained for the entire fade.

To see an example of `fadeTo()` used in a completed movie, see the Sound Control movie in the Learning/Lingo folder inside the Director application folder.

Parameters

`intVolume` Required. An integer that specifies the volume level to change to. The range of values for *`intVolume`* volume is 0 to 255.

`intMilliseconds` Optional. An integer that specifies the number of milliseconds over which the volume is changed to *`intVolume`*. The default value is 1000 milliseconds (1 second) if no value is given.

Example

The following statement changes the volume of sound channel 4 to 150 over a period of 2 seconds. It can be a fade up or a fade down, depending on the original volume of sound channel 4 when the fade begins.

```
-- Lingo syntax
sound(4).fadeTo(150, 2000)

// JavaScript syntax
sound(4).fadeTo(150, 2000);
```

See also

fadeIn(), fadeOut(), pan, Sound Channel, volume (Windows Media)

fileName()

Usage

```
-- Lingo syntax
fileioObjRef.fileName()

// JavaScript syntax
fileioObjRef.fileName();
```

Description

Fileio method; Returns the full path and name of an open file.

You must first open a file by calling `openFile()` before using `fileName()` to return the file's name.

Parameters

None.

Example

This statement creates a file and prints the location of the file.

```
-- Lingo syntax
objFileio = new xtra("fileio")
objFileio.createFile(_player.ApplicationPath)
put objFileio.fileName()

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.createFile(_player.ApplicationPath);
trace(objFileio.fileName());
```

See also

Fileio , openFile()

fileOpen()

Usage

```
FileOpen(MUIObject, string)
```

Description

This function displays a standard file open dialog box provided by an instance of the MUI Xtra.

The second parameter specifies a string that appears in the editable field when the dialog box opens. The user can specify which file to open by entering the file name in the editable field. When the user clicks a button, the text is returned.

- If the user clicks Cancel, the returned text is the same as the value that was passed in.
- If the user clicks OK, the returned text is a platform-specific path.

Parameters

None.

Example

These statements create and display a standard file open dialog box.

- The first statement creates an instance of the MUI Xtra, which is the object used as the dialog box.
- The second statement assigns a string to the variable fileString, which is used later as the second parameter of the FileOpen command.
- The third statement uses the FileOpen command to generate the open file dialog box.
- The final statements check whether the original string sent with the FileOpen command is the same as the string that was returned when the user clicked a button. If the values are different, the user selected a file to open.

```
-- Lingo syntax
set aMuiObj = new (xtra "MUI")

set fileString = "Open this file"

set result = fileOpen(aMuiObj, fileString)

-- Check to see if the dialog was canceled

if (result <> fileString) then
   -- The dialog wasn't canceled, do something with the new path data.
else
   put "ERROR - fileOpen requires a valid aMuiObj"
end if
```

See also

Fileio , openFile()

fileSave()

Usage

```
FileSave( MUIObject, string, message )
```

Description

This function displays a standard file saving dialog box that saves the current file.The dialog box is created from an instance of the MUI Xtra.

- The string parameter specifies the string that appears in the dialog box's file name field. The user can use this field to enter a new file name for the file. When the user clicks a button, Lingo returns a value for string that contains the field's content. If the user clicks Cancel, the returned string is the same as the original string.
- The message parameter is the string that appears above the dialog box's editable field.

Parameters

None.

Example

These statements create and display a file save dialog box.

- The first statement creates an instance of the MUI Xtra, which is the object used as the dialog box.
- The second statement assigns a string to the variable fileString, which is used later as the second parameter of the FileSave command.
- The third statement uses the FileSave command to generate the save file dialog box.

• The final statements check whether the result after the user clicks a button is the same as the string sent when the dialog box opened. If it differs, the user clicked something other than Cancel.

```
-- Lingo syntax
set aMuiObj = new (xtra "MUI")

set fileString = "Save this file"

set result = fileSave(aMuiObj, fileString, "with this prompt")

-- Check to see if the dialog was canceled

if (result <> fileString) then
   -- The dialog wasn't canceled, do something with the new path data.
end if
```

fill()

Usage

```
-- Lingo syntax
imageObjRef.fill(left, top, right, bottom, colorObjOrParamList)
imageObjRef.fill(point(x, y), point(x, y), colorObjOrParamList)
imageObjRef.fill(rect, colorObjOrParamList)

// JavaScript syntax
imageObjRef.fill(left, top, right, bottom, colorObjOrParamList);
imageObjRef.fill(point(x, y), point(x, y), colorObjOrParamList);
imageObjRef.fill(rect, colorObjOrParamList);
```

Description

Image method. Fills a rectangular region with a specified color in a given image object.

This method returns a value of 1 if there is no error, zero if there is an error.

For best performance, with 8-bit or lower images the color object should contain an indexed color value. For 16- 32-bit images, use an RGB color value.

Parameters

left Required if filling a region specified by coordinates. An integer that specifies the left side of the region to fill.

top Required if filling a region specified by coordinates. An integer that specifies the top side of the region to fill.

right Required if filling a region specified by coordinates. An integer that specifies the right side of the region to fill.

bottom Required if filling a region specified by coordinates. An integer that specifies the bottom side of the region to fill.

colorObjOrParamList Required. A color object or parameter list that specifies the color used to fill the region. The parameter list can be used instead of a simple color object to specify the following properties.

Property	Description
#shapeType	A symbol value of `#oval`, `#rect`, `#roundRect`, or `#line`. The default is #line.
#lineSize	The width of the line to use in drawing the shape.
#color	A color object, which determines the fill color of the region.
#bgColor	A color object, which determines the color of the region's border.

`point(x, y), point(x, y)` Required if filling a region using points. Two points that specify the upper-left and lower-right corners of region to fill, relative to the upper-left corner of the given image object.

`rect` Required if filling a region using a rectangle. A rectangle that specifies the rectangular region to fill.

Example

This statement fills a region specified by the two point which are the upper-left and lower-right corners of region to fill, relative to the upper-left corner of the given image object - Stage.

```
-- Lingo
objImage = _movie.stage.image
objImage.fill(point(20, 20), point(30, 60), rgb(255, 0 ,0))

// Javascript
var objImage = _movie.stage.image ;
objImage.fill(point(20, 20), point(30, 60), color(255, 0 ,0)) ;
```

See also

`color()`, `draw()`, `image()`

filter()

Usage

```
filter(filter symbol)
```

Description

Bitmap Filters method; used to create bitmap filters and use it on a sprite.

Example

The first statement sets the variable named myFilter to the Blur filter. The next line sets the blur filter to the sprite(1).

```
--Lingo syntax
MyFilter=filter(#BlurFilter)
sprite(1).filterlist.append(MyFilter)

// JavaScript syntax
var MyFilter = filter(symbol("BlurFilter"));
sprite(1).filterlist.append(MyFilter);
```

See also

Bitmap filters in Using Director.

findLabel()

Usage

```
-- Lingo syntax
spriteObjRef.findLabel(whichLabelName)

// JavaScript syntax
spriteObjRef.findLabel(whichLabelName);
```

Description

Function: this function returns the frame number (within the Flash movie) that is associated with the label name requested.

A 0 is returned if the label doesn't exist, or if that portion of the Flash movie has not yet been streamed in.

Parameters

whichLabelName Required. Specifies the frame label to find.

Example

This returns the frame number (within the Flash movie- in member(1)) that is associated with the label name "GetMe"

```
-- Lingo syntax
sprite(1).findLabel("GetMe")

// JavaScript syntax
sprite(1).findLabel("GetMe");
```

findEmpty()

Usage

```
-- Lingo syntax
castObjRef.findEmpty({memberObjRef})

// JavaScript syntax
castObjRef.findEmpty({memberObjRef});
```

Description

Cast library method; displays the next empty cast member position or the position after a specified cast member.

This method is available only on the current cast library.

Parameters

memberObjRef Optional. A reference to the cast member after which the next empty cast member position is displayed. If omitted, the next empty cast member position is displayed.

Example

This statement finds the first empty cast member on or after cast member 100:

```
-- Lingo syntax
trace(castLib(1).findEmpty(member(100)))

// JavaScript syntax
trace(castLib(1).findEmpty(member(100)));
```

See also

Cast Library, Member

findPos

Usage

```
list.findPos(property)
findPos(list, property)
```

Description

List command; identifies the position of a property in a property list.

Using `findPos` with linear lists returns a bogus number if the value of *`property`* is a number and a script error if the value of *`property`* is a string.

The `findPos` command performs the same function as the `findPosNear` command, except that `findPos` is `VOID` when the specified property is not in the list.

When you add a filter using the add or append method of the filterlist, a duplicate is created and added to the list. Methods such as deleteOne, getPos, findPos, and getOne use the exact value in the list and not the duplicate value.

In such cases, you can use the findPos method, as follows:

```
f = filter(#glowfilter)
sprite(1).filterlist.append(f)
f = sprite(1).filterlist[1]-- here we get the actual value added to the list.
sprite(1).filterlist.findPos(f)
```

The third line in the above script addds the reference of the filter value to the list.

Parameters

`property` Required. The property whose position is identified.

Example

This statement identifies the position of the property `c` in the list Answers, which consists of `[#a:10, #b:12, #c:15, #d:22]`:

```
-- Lingo
Answers.findPos(#c)

// Javascript
Answers.findPos("c");
```

The result is 3, because `c` is the third property in the list.

See also

`findPosNear`, `sort`

findPosNear

Usage

```
sortedList.findPosNear(valueOrProperty)
findPosNear(sortedList, valueOrProperty)
```

Description

List command; for sorted lists only, identifies the position of an item in a specified sorted list.

The `findPosNear` command works only with sorted lists. Replace *`valueOrProperty`* with a value for sorted linear lists, and with a property for sorted property lists.

The `findPosNear` command is similar to the `findPos` command, except that when the specified property is not in the list, the `findPosNear` command identifies the position of the value with the most similar alphanumeric name. This command is useful in finding the name that is the closest match in a sorted directory of names.

Parameters

`valueOrProperty` Required. The value or property whose position is identified.

Example

This statement identifies the position of a property in the sorted list Answers, which consists of `[#Nile:2, #Pharaoh:4, #Raja:0]`:

```
Answers.findPosNear(#Ni)
```

The result is 1, because Ni most closely matches `Nile`, the first property in the list.

See also

`findPos`

finishIdleLoad()

Usage

```
-- Lingo syntax
_movie.finishIdleLoad(intLoadTag)

// JavaScript syntax
_movie.finishIdleLoad(intLoadTag);
```

Description

Movie method; forces completion of loading for all the cast members that have the specified load tag.

Parameters

`intLoadTag` Required. An integer that specifies the load tag of the cast members to be loaded.

Example

This statement completes the loading of all cast members that have the load tag 20:

```
-- Lingo syntax
_movie.finishIdleLoad(20)

// JavaScript syntax
_movie.finishIdleLoad(20);
```

See also

`idleHandlerPeriod`, `idleLoadDone()`, `idleLoadMode`, `idleLoadPeriod`, `idleLoadTag`, `idleReadChunkSize`, `Movie`

flashToStage()

Usage

```
-- Lingo syntax
spriteObjRef.flashToStage(pointInFlashMovie)

// JavaScript syntax
spriteObjRef.flashToStage(pointInFlashMovie);
```

Description

Function; returns the coordinate on the Director Stage that corresponds to a specified coordinate in a Flash movie sprite. The function accepts both the Flash channel and movie coordinate and returns the Director Stage coordinate as Director point values: for example, point(300,300).

Flash movie coordinates are measured in Flash movie pixels, which are determined by a movie's original size when it was created in Flash. For the purpose of calculating Flash movie coordinates, point(0,0) of a Flash movie is always at its upper left corner. (The cast member's `originPoint` property is used only for rotation and scaling, not to calculate movie coordinates.)

The `flashToStage` and the corresponding `stageToFlash` functions are helpful for determining which Flash movie coordinate is directly over a Director Stage coordinate. For both Flash and Director, point(0,0) is the upper left corner of the Flash Stage or Director Stage. These coordinates may not match on the Director Stage if a Flash sprite is stretched, scaled, or rotated.

Parameters

`pointInFlashMovie` Required. The point in the Flash movie sprite whose coordinates are returned.

Example

This handler accepts a point value and a sprite reference as a parameter, and it then sets the upper left coordinate of the specified sprite to the specified point within a Flash movie sprite in channel 10:

```
-- Lingo syntax
on snapSprite(whichFlashPoint, whichSprite)
    sprite(whichSprite).loc = sprite(1).FlashToStage(whichFlashPoint)
    _movie.updatestage()
end

// JavaScript syntax
function snapSprite(whichFlashPoint, whichSprite) {
    sprite(whichSprite).loc = sprite(1).FlashToStage(whichFlashPoint);
    _movie.updateStage();
}
```

See also

`stageToFlash()`

float()

Usage

```
(expression).float
float (expression)
```

Description

Function (Lingo only); converts an expression to a floating-point number. The number of digits that follow the decimal point (for display purposes only, calculations are not affected) is set using the `floatPrecision` property.

In JavaScript syntax, use the `parseFloat()` function.

Parameters

`expression` Required. The expression to convert to a floating-point number.

Example

This statement converts the integer 1 to the floating-point number 1:

```
put (1).float
-- 1.0
```

Math operations can be performed using float; if any of the terms is a float value, the entire operation is performed with `float`:

```
put 2 + 2
-- 4
put (2).float + 2
-- 4.0
the floatPrecision = 4
put 22/7
-- 3
put (22).float / 7
-- 3.1429"
```

See also

`floatPrecision`, `ilk()`

floatP()

Usage

```
(expression).floatP
floatP(expression)
```

Description

Function (Lingo only); indicates whether an expression is a floating-point number (1 or `TRUE`) or not (0 or `FALSE`).

The *P* in `floatP` stands for *predicate*.

Parameters

`expression` Required. The expression to test.

Example

This statement tests whether 3.0 is a floating-point number. The Message window displays the number 1, indicating that the statement is `TRUE`.

```
put (3.0).floatP
-- 1
```

This statement tests whether 3 is a floating-point number. The Message window displays the number 0, indicating that the statement is `FALSE`.

```
put (3).floatP
-- 0
```

See also

float(), ilk(), integerP(), objectP(), stringP(), symbolP()

flushInputEvents()

Usage

```
-- Lingo syntax
_player.flushInputEvents()

// JavaScript syntax
_player.flushInputEvents();
```

Description

Player method; flushes any waiting mouse or keyboard events from the Director message queue.

Generally this is useful when script is in a tight loop and the author wants to make sure any mouse clicks or keyboard presses don't get through.

This method operates at runtime only and has no effect during authoring.

Parameters

None.

Example

This statement disables mouse and keyboard events while a repeat loop executes:

```
-- Lingo syntax
repeat with i = 1 to 10000
    _player.flushInputEvents()
    sprite(1).loc = sprite(1).loc + point(1, 1)
end repeat

// JavaScript syntax
for (var i = 1; i <= 10000; i++) {
    _player.flushInputEvents();
    sprite(1).loc = sprite(1).loc + point(1, 1);
}
```

See also

on keyDown, on keyUp, on mouseDown (event handler), on mouseUp (event handler), Player

forget() (Window)

Usage

```
-- Lingo syntax
windowObjRef.forget()

// JavaScript syntax
windowObjRef.forget();
```

Description

Window method; instructs script to close a window and stop its playback when it's no longer in use and no other variables refer to it.

Calling `forget()` on a window also removes that window's reference from the `windowList`.

When the `forget()` method is called, the window and the movie in a window (MIAW) disappear without calling the `stopMovie`, `closeWindow`, or `deactivateWindow` handlers.

If there are many global references to the movie in a window, the window doesn't respond to the `forget()` method.

Parameters

None.

Example

This statement instructs Lingo to delete the window Control Panel when the movie no longer uses the window:

```
-- Lingo syntax
window("Control Panel").forget()

// JavaScript syntax
window("Control Panel").forget();
```

See also

`close()`, `open() (Window)`, `Window`, `windowList`

forget() (Timeout)

Usage

```
timeout("timeoutName").forget()
forget(timeout("timeoutName"))
```

Description

This timeout object function removes a timeout object from the `timeoutList`, and prevents it from sending further timeout events.

Parameters

None.

Example

This statement deletes the timeout object named AlarmClock from the `timeoutList`:

```
-- Lingo
timeout("AlarmClock").forget()

// Javascript
timeout("AlarmClock").forget();
```

See also

`timeout()`, `timeoutHandler`, `timeoutList`, `new()`

framesToHMS()

Usage

```
framesToHMS(frames, tempo, dropFrame, fractionalSeconds)
```

Description

Function; converts the specified number of frames to their equivalent length in hours, minutes, and seconds. This function is useful for predicting the actual playtime of a movie or controlling a video playback device.

The resulting string uses the form `sHH:MM:SS.FFD`, where:

s	A character is used if the time is less than zero, or a space if the time is greater than or equal to zero.
HH	Hours.
MM	Minutes.
SS	Seconds.
FF	Indicates a fraction of a second if *fractionalSeconds* is `TRUE` or `frames` if *fractionalSeconds* is `FALSE`.
D	A `"d"` is used if *dropFrame* is `TRUE`, or a space if *dropFrame* is `FALSE`.

Parameters

frames Required. An integer expression that specifies the number of frames.

tempo Required. An integer expression that specifies the tempo in frames per second.

dropFrame Required. Compensates for the color NTSC frame rate, which is not exactly 30 frames per second and is meaningful only if FPS is set to 30 frames per second. Normally, this parameter is set to `FALSE`.

fractionalSeconds Required. Determines whether the residual frames are converted to the nearest hundredth of a second (`TRUE`) or returned as an integer number of frames (`FALSE`).

Example

The following statement converts a 2710-frame, 30 frame-per-second movie. The `dropFrame` and `fractionalSeconds` arguments are both turned off:

```
put framesToHMS(2710, 30, FALSE, FALSE)
-- " 00:01:30.10 "
```

See also

HMStoFrames()

frameReady() (Movie)

Usage

```
-- Lingo syntax
_movie.frameReady({intFrameNum})
_movie.frameReady(frameNumA, frameNumB)

// JavaScript syntax
_movie.frameReady({intFrameNum});
_movie.frameReady(frameNumA, frameNumB);
```

Description

Movie method; for Director movies, projectors, and movies with Shockwave content, determines whether the cast members of a frame or range of frames have been downloaded.

This method returns `TRUE` if the specified cast members have been downloaded, and `FALSE` if not.

For a demonstration of the `frameReady()` method used in a Director movie, see the sample movie "Streaming Shockwave" in Director Help.

Parameters

`intFrameNum` Optional if testing whether a single frame's cast members have been downloaded. An integer that specifies the individual frame to test. If omitted, `frameReady()` determines whether the cast members used in any frame of a Score have been downloaded.

`frameNumA` Required if testing whether the cast members in a range of frames have been downloaded. An integer that specifies the first frame in the range.

`frameNumB` Required if testing whether the cast members in a range of frames have been downloaded. An integer that specifies the last frame in the range.

Example

This statement determines whether the cast members for frame 20 are downloaded and ready to be viewed:

```
-- Lingo syntax
on exitFrame
    if (_movie.frameReady(20)) then
        _movie.go(20)
    else
        _movie.go(1)
    end if
end

// JavaScript syntax
function exitFrame() {
    if (_movie.frameReady(20)) {
        _movie.go(20);
    }
    else {
        _movie.go(1);
    }
}
```

The following frame script checks to see if frame 25 of a Flash movie sprite in channel 5 can be rendered. If it can't, the script keeps the playhead looping in the current frame of the Director movie. When frame 25 can be rendered, the script starts the movie and lets the playhead proceed to the next frame of the Director movie.

See also

`mediaReady`, `Movie`

frameStep()

Usage

```
-- Lingo syntax
dvdObjRef.frameStep(intFrames)
```

```
// JavaScript syntax
dvdObjRef.frameStep(intFrames);
```

Description

DVD method; steps forward from the current location a specified number of frames when playback is paused.

Stepping backward is not supported by either Windows or Mac system software for DVD playback.

Parameters

intFrames Required. An integer that specifies the number of frames to step forward.

Example

This statements jumps 100 frames forward:

```
-- Lingo syntax
member("drama").frameStep(100)

// JavaScript syntax
member("drama").frameStep(100);
```

See also

DVD

freeBlock()

Usage

```
the freeBlock
```

Description

Function; indicates the size of the largest free contiguous block of memory, in bytes. A kilobyte (K) is 1024 bytes. A megabyte (MB) is 1024 kilobytes. Loading a cast member requires a free block at least as large as the cast member.

Parameters

None.

Example

This statement determines whether the largest contiguous free block is smaller than 10K and displays an alert if it is:

```
-- Lingo syntax
if (the freeBlock < (10 * 1024)) then alert "Not enough memory!"

// JavaScript syntax
if (freeBlock < (10 * 1024)) {
    alert("Not enough memory!")
}
```

See also

freeBytes(), memorySize, ramNeeded(), size

freeBytes()

Usage

```
the freeBytes
```

Description

Function; indicates the total number of bytes of free memory, which may not be contiguous. A kilobyte (K) is 1024 bytes. A megabyte (MB) is 1024 kilobytes.

This function differs from `freeBlock` in that it reports all free memory, not just contiguous memory.

On the Mac, selecting Use System Temporary Memory in the Director General Preferences or in a projector's Options dialog box tells the `freeBytes` function to return all the free memory that is available to the application. This amount equals the application's allocation shown in its Get Info dialog box and the Largest Unused Block value in the About This Mac dialog box.

Parameters

None.

Example

This statement checks whether more than 200K of memory is available and plays a color movie if it is:

```
if (the freeBytes > (200 * 1024)) then play movie "colorMovie"
```

See also

`freeBlock()`, `memorySize`, `objectP()`, `ramNeeded()`, `size`

generateNormals()

Usage

```
member(whichCastmember).modelResource(whichModelResource).generateNormals(style)
```

Description

3D `#mesh` model resource command; calculates the `normal` vectors for each vertex of the mesh.

If the *`style`* parameter is set to `#flat`, each vertex receives a normal for each face to which it belongs. Furthermore, all three of the vertices of a face will have the same normal. For example, if the vertices of `face[1]` all receive `normal[1]` and the vertices of `face[2]` all receive `normal[2]`, and the two faces share `vertex[8]`, then the normal of `vertex[8]` is `normal[1]` in `face[1]` and `normal[2]` in `face[2]`. Use of the `#flat` parameter results in very clear delineation of the faces of the mesh.

If the *`style`* parameter is set to `#smooth`, each vertex receives only one normal, regardless of the number of faces to which it belongs, and the three vertices of a face can have different normals. Each vertex normal is the average of the face normals of all of the faces that share the vertex. Use of the `#smooth` parameter results in a more rounded appearance of the faces of the mesh, except at the outer edges of the faces at the silhouette of the mesh, which are still sharp.

A vertex normal is a direction vector which indicates the "forward" direction of a vertex. If the vertex normal points toward the camera, the colors displayed in the area of the mesh controlled by that normal are determined by the shader. If the vertex normal points away from the camera, the area of the mesh controlled by that normal will be non-visible.

After using the `generateNormals()` command, you must use the `build()` command to rebuild the mesh.

Parameters

`style` Required. A symbol that specifies the style of the vertex.

Example

The following statement calculates vertex normals for the model resource named FloorMesh. The *`style`* parameter is set to `#smooth`, so each vertex in the mesh will receive only one normal.

```
-- Lingo
member("Room").modelResource("FloorMesh").generateNormals(#smooth)

// Javascript
member("Room").getProp("modelResource", "index (of the
modelresource)").generateNormals("smooth");
```

See also

`build()`, `face[ ]`, `normalList`, `normals`, `flat`

getaProp

Usage

```
propertyList.propertyName
getaProp(list, item)
list[listPosition]
propertyList [ #propertyName ]
propertyList [ "propertyName" ]
```

Description

List command; for linear and property lists, identifies the value associated with the item specified by *`item`*, *`listPosition`*, or *`propertyName`* in the list specified by *`list`*.

- When the list is a linear list, replace *`item`* with the number for an item's position in a list as shown by *`listPosition`*. The result is the value at that position.
- When the list is a property list, replace *`item`* with a property in the list as in *`propertyName`*. The result is the value associated with the property.

The `getaProp` command returns `VOID` when the specified value is not in the list.

When used with linear lists, the `getaProp` command has the same function as the `getAt` command.

Parameters

`itemNameOrNum` Required. For linear lists, an integer that specifies the index position of the value in the list to return; for property lists, a symbol (Lingo) or a string (JavaScript syntax) that specifies the property whose value is returned.

Example

This statement identifies the value associated with the property `#joe` in the property list ages, which consists of [#john:10, #joe:12, #cheryl:15, #barbara:22]:

```
put getaProp(ages, #joe)
```

The result is 12, because this is the value associated with the property `#joe`.

The same result can be achieved using bracket access on the same list:

```
put ages[#joe]
```

The result is again 12.

If you want the value at a certain position in the list, you can also use bracket access. To get the third value in the list, associated with the third property, use this syntax:

```
put ages[3]
-- 15
```

Note: *Unlike the getAProp command where VOID is returned when a property doesn't exist, a script error will occur if the property doesn't exist when using bracket access.*

See also

`getAt`, `getOne()`, `getProp()`, `setaProp`, `setAt`

getAt

Usage

```
getAt(list, position)
list [position]
```

Description

List command; identifies the item in a specified position of a specified list. If the list contains fewer elements than the specified position, a script error occurs.

The `getAt` command works with linear and property lists. This command has the same function as the `getaProp` command for linear lists.

This command is useful for extracting a list from within another list, such as the deskTopRectList.

Parameters

`list` Required. Specifies the list from in which the item exists.

`position` Required. Specifies the index position of the item in the list.

Example

This statement causes the Message window to display the third item in the answers list, which consists of [10, 12, 15, 22]:

```
put getAt(answers, 3)
-- 15
```

The same result can be returned using bracket access:

```
put answers[3]
-- 15
```

The following example extracts the first entry in a list containing two entries that specify name, department, and employee number information. Then the second element of the newly extracted list is returned, identifying the department in which the first person in the list is employed. The format of the list is [["Dennis", "consulting", 510], ["Sherry", "Distribution", 973]], and the list is called employeeInfoList.

```
firstPerson = getAt(employeeInfoList, 1)
put firstPerson
-- ["Dennis", "consulting", 510]
```

```
firstPersonDept = getAt(firstPerson, 2)
put firstPersonDept
-- "consulting"
```

It's also possible to nest `getAt` commands without assigning values to variables in intermediate steps. This format can be more difficult to read and write, but less verbose.

```
firstPersonDept = getAt(getAt(employeeInfoList, 1), 2)
put firstPersonDept
 -- "consulting"
```

You can also use the bracket list access:

```
firstPerson = employeeInfoList[1]
put firstPerson
-- ["Dennis", "consulting", 510]
firstPersonDept = firstPerson[2]
put firstPersonDept
-- "consulting"
```

As with getAt, brackets can be nested:

```
firstPersonDept = employeeInfoList[1][2]
```

See also

getaProp, setaProp, setAt

getError() (Flash, SWA)

Usage

```
-- Lingo syntax
memberObjRef.getError()

// JavaScript syntax
memberObjRef.getError();
```

Description

Function; for Shockwave Audio (SWA) or Flash cast members, indicates whether an error occurred as the cast member streamed into memory and returns a value.

Shockwave Audio cast members have the following possible `getError()` integer values and corresponding `getErrorString()` messages:

getError() value	getErrorString() message
0	OK
1	memory
2	network
3	playback device
99	other

Flash movie cast members have the following possible `getError` values:

- `FALSE`—No error occurred.

- `#memory`—There is not enough memory to load the cast member.
- `#fileNotFound`—The file containing the cast member's assets could not be found.
- `#network`—A network error prevented the cast member from loading.
- `#fileFormat`—The file was found, but it appears to be of the wrong type, or an error occurred while reading the file.
- `#other`—Some other error occurred.

When an error occurs as a cast member streams into memory, Director sets the cast member's state property to -1. Use the `getError` function to determine what type of error occurred.

Parameters

None.

Example

This handler uses `getError` to determine whether an error involving the Shockwave Audio cast member Norma Desmond Speaks occurred and displays the appropriate error string in a field if it did:

```
-- Lingo syntax
on exitFrame
    if member("Norma Desmond Speaks").getError() <> 0 then
        member("Display Error Name").text = member("Norma Desmond \
Speaks").getErrorString()
    end if
end

// JavaScript syntax
function exitFrame() {
var memNor = member("Norma Desmond Speaks").getError();
    if (memNor != 0) {
        member("Display Error Name").text = member("Norma Desmond Speaks").getErrorString();
    }
}
```

The following handler checks to see whether an error occurred for a Flash cast member named Dali, which was streaming into memory. If an error occurred, and it was a memory error, the script uses the `unloadCast` command to try to free some memory; it then branches the playhead to a frame in the Director movie named Artists, where the Flash movie sprite first appears, so Director can again try to load and play the Flash movie. If something other than an out-of-memory error occurred, the script goes to a frame named Sorry, which explains that the requested Flash movie can't be played.

```
-- Lingo syntax
on CheckFlashStatus
errorCheck = member("Dali").getError()
    if errorCheck <> 0 then
        if errorCheck = #memory then
            member("Dali").clearError()
            unloadCast()
            _movie.go("Artists")
        else
            _movie.go("Sorry")
        end if
    end if
end

// JavaScript syntax
function CheckFlashStatus() {
```

```
var errorCheck = member("Dali").getError();
    if (errorCheck != 0) {
        if (errorCheck == "memory") {
            member("Dali").clearError();
            unloadCast();
            _movie.go("Artists");
        } else {
            _movie.go("Sorry");
        }
    }
}
```

See also

clearError(), getErrorString(), state (Flash, SWA)

getError() (XML)

Usage

```
parserObject.getError()
```

Description

Function; returns the descriptive error string associated with a given error number (including the line and column number of the XML where the error occurred). When there is no error, this function returns <VOID>.

Parameters

None.

Example

These statements check an error after parsing a string containing XML data:

```
-- Lingo
errCode = parserObject.parseString(member("XMLtext").text)
errorString = parserObject.getError()
if voidP(errorString) then
    -- Go ahead and use the XML in some way
else
    alert "Sorry, there was an error " & errorString
    -- Exit from the handler
    exit
end if

// Javascript
errCode = parserObject.parseString(member("XMLtext").text);
errorString = parserObject.getError();
if (errorString != null)
{
    // Go ahead and use the XML in some way
}else{
    _player.alert("Sorry, there was an error " + errorString);
    // Exit from the handler
}
```

getErrorString()

Usage

```
-- Lingo syntax
memberObjRef.getErrorString()

// JavaScript syntax
memberObjRef.getErrorString();
```

Description

Function; for Shockwave Audio (SWA) cast members, returns the error message string that corresponds to the error value returned by the `getError()` function.

Possible `getError()` integer values and corresponding `getErrorString()` messages are:

getError() value	getErrorString() message
0	OK
1	memory
2	network
3	playback device
99	other

Parameters

None.

Example

This handler uses `getError()` to determine whether an error occurred for Shockwave Audio cast member Norma Desmond Speaks, and if so, uses `getErrorString` to obtain the error message and assign it to a field cast member:

```
-- Lingo syntax
on exitFrame
    if member("Norma Desmond Speaks").getError() <> 0 then
        member("Display Error Name").text = member("Norma Desmond Speaks").getErrorString()
    end if
end

// JavaScript syntax
function exitFrame() {
    var memNor = member("Norma Desmond Speaks").getError();
    if (memNor != 0) {
        member("Display Error Name").text = member("Norma Desmond Speaks").getErrorString();
    }
}
```

See also

`getError() (Flash, SWA)`

getFinderInfo()

Usage

```
-- Lingo syntax
fileioObjRef.getFinderInfo()

// JavaScript syntax
fileioObjRef.getFinderInfo();
```

Description

Fileio method (Mac only); Returns the finder information for an open file.

You must first open a file by calling `openFile()` before using `getFinderInfo()` to return the file's finder information.

Parameters

None.

Example

```
-- Lingo syntax
objFileio = new xtra("fileio")
objFileio.openFile(stringFileName, intMode)
objFileio.getFinderInfo()

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile(stringFileName, intMode) ;
objFileio.getFinderInfo() ;
```

See also

`Fileio`, `openFile()`

getFlashProperty()

Usage

```
-- Lingo syntax
spriteObjRef.getFlashProperty(targetName, symProp)

// JavaScript syntax
spriteObjRef.getFlashProperty(targetName, symProp);
```

Description

This function allows Lingo to invoke the Flash action script function `getProperty()` on the given Flash sprite. This Flash action script function is used to get the value of properties of movie clips or levels within a Flash movie. This is similar to testing sprite properties within Director.

To get a global property of the Flash sprite, pass an empty string as the *`targetName`*. These global Flash properties can be tested: `#focusRect` and `#spriteSoundBufferTime`.

See the Flash documentation for descriptions of these properties.

Parameters

targetName Required. A string that specifies the name of the movie clip or level whose property you want to get within the given Flash sprite.

symProp Required. A symbol that specifies the name of the property to get. Valid values include: #posX, #posY, #scaleX, #scaleY, #visible, #rotate, #alpha, #name, #width, #height, #target, #url, #dropTarget, #totalFrames, #currentFrame, #cursor, and #lastframeLoaded.

Example

This statement gets the value of the #rotate property of the movie clip Star in the Flash member in sprite 3:

```
-- Lingo syntax
sprite(3).setFlashProperty("Star", #rotate)
sprite(3).getFlashProperty("Star")

// JavaScript syntax
sprite(3).setFlashProperty("Star", symbol("rotate"));
sprite(3).getFlashProperty("Star");
```

getFrameLabel()

Usage

```
sprite(whichFlashSprite).getFrameLabel(whichFlashFrameNumber)
getFrameLabel(sprite whichFlashSprite, whichFlashFrameNumber)
```

Description

Function; returns the frame label within a Flash movie that is associated with the frame number requested. If the label doesn't exist, or that portion of the Flash movie has not yet been streamed in, this function returns an empty string.

Parameters

whichFlashFrameNumber Required. Specifies the frame number that is associated with the frame label.

Example

The following handler looks to see if the marker on frame 15 of the Flash movie playing in sprite 1 is called "Lions". If it is, the Director movie navigates to frame "Lions". If it isn't, the Director movie stays in the current frame and the Flash movie continues to play.

```
-- Lingo syntax
on exitFrame
    if sprite(1).getFrameLabel(15) = "Lions" then
        go "Lions"
    else
        go the frame
    end if
end

// JavaScript syntax
function exitFrame() {
    if (sprite(1).getFrameLabel(15) == "Lions") {
        _movie.go("Lions");
    } else {
        _movie.go(_movie.frame);
    }
```

```
}
```

getHardwareInfo()

Usage

```
getRendererServices().getHardwareInfo()
```

Description

3D `rendererServices` method; returns a property list with information about the user's video card. The list contains the following properties:

`#present` is a Boolean value indicating whether the computer has hardware video acceleration.

`#vendor` indicates the name of the manufacturer of the video card.

`#model` indicates the model name of the video card.

`#version` indicates the version of the video driver.

`#maxTextureSize` is a linear list containing the maximum width and height of a texture, in pixels. Textures that exceed this size are downsampled until they do not. To avoid texture sampling artifacts, author textures of various sizes and choose the ones that do not exceed the `#maxTextureSize` value at run time.

`#supportedTextureRenderFormats` is a linear list of texture pixel formats supported by the video card. For details, see `textureRenderFormat`.

`#textureUnits` indicates the number of texture units available to the card.

`#depthBufferRange` is a linear list of bit-depth resolutions to which the `depthBufferDepth` property can be set.

`#colorBufferRange` is a linear list of bit-depth resolutions to which the `colorBufferDepth` property can be set.

Example

This statement displays a detailed property list of information about the user's hardware:

```
-- Lingo
put getRendererServices().getHardwareInfo()
-- [#present: 1, #vendor: "NVIDIA Corporation", #model: "32MB DDR NVIDIA GeForce2 GTS
(Dell)", #version: "4.12.01.0532", #maxTextureSize: [2048, 2048],
#supportedTextureRenderFormats: [#rgba8888, #rgba8880, #rgba5650, #rgba5551, #rgba5550,
#rgba4444], #textureUnits: 2, #depthBufferRange: [16, 24], #colorBufferRange: [16, 32]]

// Javascript
trace(getRendererServices().getHardwareInfo())
//<[#present: 1, #vendor: "NVIDIA Corporation", #model: "32MB DDR NVIDIA GeForce2 GTS
(Dell)", #version: "4.12.01.0532", #maxTextureSize: [2048, 2048],
#supportedTextureRenderFormats: [#rgba8888, #rgba8880, #rgba5650, #rgba5551, #rgba5550,
#rgba4444], #textureUnits: 2, #depthBufferRange: [16, 24], #colorBufferRange: [16, 32]]>
```

See also

`getRendererServices()`

getHotSpotRect()

Usage

```
-- Lingo syntax
spriteObjRef.getHotSpotRect(hotSpotID)

// JavaScript syntax
spriteObjRef.getHotSpotRect(hotSpotID);
```

Description

QuickTime VR function; returns an approximate bounding rectangle for a hot spot. If the hot spot doesn't exist or isn't visible on the Stage, this function returns rect(0, 0, 0, 0). If the hot spot is partially visible, this function returns the bounding rectangle for the visible portion.

Parameters

hotSpotID Required. Specified the hot spot from which a bounding rectangle is returned.

GetItemPropList

Usage

```
GetItemPropList(MUIObject)
```

Description

This function returns a list of the MUI Xtra's predefined properties for components in a general purpose dialog box. It is useful for defining new components in a general purpose dialog box. Use GetItemPropList to obtain a comprehensive list of properties and values and then edit individual properties as necessary.

The list of properties and values are the following:

Property	Default value
#value	0
#type	#checkBox
#attributes	[],
#title	"title"
#tip	"tip" (This is reserved for possible use in later versions of the MUI Xtra.)
#locH	20
#locV	24
#width	200
#height	210
#enabled	1

Example

These statements define the beginning of a dialog box window.

- The first statement creates an instance of the MUI Xtra, which is the object used as the dialog box.

- The second statement assigns a list of default dialog component settings to the variable tempItemProps.
- The third statement makes the component the dialog box's beginning by changing its type to #windowBegin.

```
--Lingo Syntax
set aMuiObj = new (Xtra "MUI")
set tempItemProps = GetItemPropList(aMuiObj)
set the type of tempItemProps = #windowBegin
```

getLast()

Usage

```
list.getLast()
getLast(list)
```

Description

List function; identifies the last value in a linear or property list specified by *list*.

Parameters

None.

Example

This statement identifies the last item, 22, in the list Answers, which consists of [10, 12, 15, 22]:

```
put Answers.getLast()
```

This statement identifies the last item, 850, in the list Bids, which consists of [#Gee:750, #Kayne:600, #Ohashi:850]:

```
put Bids.getLast()
```

getLatestNetID

Usage

```
getLatestNetID
```

Description

This function returns an identifier for the last network operation that started.

The identifier returned by `getLatestNetID` can be used as a parameter in the `netDone, netError,` and `netAbort` functions to identify the last network operation.

Note: *This function is included for backward compatibility. It is recommended that you use the network ID returned from a net lingo function rather than `getLatestNetID`. However, if you use `getLatestNetID`, use it immediately after issuing the netLingo command.*

Parameters

None.

Example

This script assigns the network ID of a `getNetText` operation to the field cast member Result so results of that operation can be accessed later:

```
on startOperation
    global gNetID
    getNetText("url")
    set gNetID = getLatestNetID()
end
on checkOperation
    global gNetID
    if netDone(gNetID) then
        put netTextResult into member "Result"
    end if
end
```

See also

netAbort, netDone(), netError()

getLength()

Usage

```
-- Lingo syntax
fileioObjRef.getLength()

// JavaScript syntax
fileioObjRef.getLength();
```

Description

Fileio method; Returns the length of an open file.

You must first open a file by calling `openFile()` before using `getLength()` to return the length of the file.

Parameters

None.

Example

The below mentioned sample opens a file called c:\check.txt and gets the length of the file.

```
-- Lingo syntax
objFileio = new xtra("fileio")
objFileio.openFile("c:\check.txt",2)
put objFileio.getLength()

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile("c:\check.txt",2);
trace(objFileio.getLength())
```

See also

Fileio, openFile()

getNetText()

Usage

```
getNetText(URL {, serverOSString} {, characterSet})
```

```
getNetText(URL, propertyList {, serverOSString} {, characterSet})
```

Description

Function; starts the retrieval of text from a file usually on an HTTP or FTP server, or initiates a CGI query.

The first syntax shown starts the text retrieval. You can submit HTTP CGI queries this way and must properly encode them in the URL. The second syntax includes a property list and submits a CGI query, providing the proper URL encoding.

Use the optional parameter *propertyList* to take a property list for CGI queries. The property list is URL encoded and the URL sent is `(urlstring & "?" & encodedproplist)`.

Use the optional parameter *serverOSString* to encode any return characters in *propertylist*. The value defaults to `UNIX` but may be set to `Win` or `Mac` and translates any carriage returns in the *propertylist* argument into those used on the server. For most applications, this setting is unnecessary because line breaks are usually not used in form responses.

The optional parameter *characterSet* applies only if the user is running Director on a `shift-JIS` (Japanese) system. Possible character set settings are `JIS`, `EUC`, `ASCII`, and `AUTO`. Lingo converts the retrieved data from `shift-JIS` to the named character set. Using the AUTO setting, character set tries to determine what character set the retrieved text is in and translate it to the character set on the local machine. The default setting is ASCII.

Use `netDone` to find out when the `getNetText` operation is complete, and `netError` to find out if the operation was successful. Use `netTextResult` to return the text retrieved by `getNetText`.

The function works with relative URLs.

To see an example of `getNetText()` used in a completed movie, see the Forms and Post movie in the Learning/Lingo folder inside the Director application folder.

Parameters

URL Required. The URL to the file that contains the text to get.

propertyList Optional. Specifies a property list used for CGI queries.

serverOSString Optional. Specifies the encoding of return characters in *propertyList*.

characterSet Optional. Specifies character settings.

Example

This script retrieves text from the URL http://BigServer.com/sample.txt and updates the field cast member the mouse pointer is on when the mouse button is clicked:

```
property spriteNum

property theNetID

on mouseUp me
    theNetID = getNetText ("http://BigServer.com/sample.txt")
end

on exitFrame me
    if netDone(theNetID) then
        sprite(spriteNum).member.text = netTextResult(theNetID)
    end if
end
```

This example retrieves the results of a CGI query:

```
getNetText("http://www.yourserver.com/cgi-bin/query.cgi?name=Bill")
```

This is the same as the previous example, but it uses a property list to submit a CGI query, and does the URL encoding for you:

```
getNetText("http://www.yourserver.com/cgi-bin/query.cgi", [#name:"Bill"])
```

See also

netDone(), netError(), netTextResult()

getNormalized

Usage

```
getNormalized(vector)
vector.getNormalized()
```

Description

3D vector method; copies the vector and divides the x, y, and z components of the copy by the length of the original vector. The resulting vector has a length of 1 world unit.

This method returns the copy and leaves the original vector unchanged. To normalize the original vector, use the `normalize` command.

Example

The following statement stores the normalized value of the vector MyVec in the variable `Norm`. The value of Norm is vector (-0.1199, 0.9928, 0.0000) and the magnitude of Norm is 1.

```
-- Lingo
MyVec = vector(-209.9019, 1737.5126, 0.0000)
Norm = MyVec.getNormalized()
put Norm
-- vector( -0.1199, 0.9928, 0.0000 )
put Norm.magnitude
-- 1.0000

// Javascript
MyVec = vector(-209.9019, 1737.5126, 0.0000);
Norm = MyVec.getNormalized();
trace(Norm);
// vector( -0.1199, 0.9928, 0.0000 )
trace(Norm.magnitude);
// 1.0000
```

See also

normalize

getNthFileNameInFolder()

Usage

```
getNthFileNameInFolder(folderPath, fileNumber)
```

Description

Movie method; returns a filename from the directory folder based on the specified path and number within the folder. To be found by the `getNthFileNameInFolder` function, Director movies must be set to visible in the folder structure. (On the Mac, other types of files are found whether they are visible or invisible.) If this function returns an empty string, you have specified a number greater than the number of files in the folder.

The `getNthFileNameInFolder` function doesn't work with URLs.

To specify other folder names, use the `@ pathname` operator or the full path defined in the format for the specific platform on which the movie is running. For example:

- In Windows, use a directory path such as C:/Director/Movies.
- On the Mac, use a pathname such as HardDisk:Director:Movies. To look for files on the Mac desktop, use the path HardDisk:Desktop Folder
- This function is not available in Shockwave Player.

Parameters

`folderPath` Required. Specifies the path to the folder that contains the file.

`fileNumber` Required. Specifies the index position of the file in the folder.

Example

The following handler returns a list of filenames in the folder on the current path. To call the function, use parentheses, as in `put currentFolder()`.

```
-- Lingo
on currentFolder
    fileList = [ ]
    repeat with i = 1 to 100
        n = getNthFileNameInFolder(the moviePath, i)
        if n = EMPTY then exit repeat
        fileList.append(n)
    end repeat
    return fileList
end currentFolder

// Javascript
Function currentFolder()
{
    fileList = list();
    var i=0;
    while(i<100)
    {
        n= getNthFileNameInFolder(_movie.path,i);
        if ( n==null)
        {
            I=101;
        }
        Else
        {
            fileList.append(n);
        }
        I++;
    }
    Return fileList;
}
```

See also

@ (pathname), Movie

getOne()

Usage

```
list.getOne(value)
getOne(list, value)
```

Description

List function; identifies the position (linear list) or property (property list) associated with a value in a list.

For values contained in the list more than once, only the first occurrence is displayed. The `getOne` command returns the result 0 when the specified value is not in the list.

When used with linear lists, the `getOne` command performs the same functions as the `getPos` command.

When you add a filter using the add or append method of the filterlist, a duplicate is created and added to the list. Methods such as deleteOne, getPos, findPos, and getOne use the exact value in the list and not the duplicate value.

In such cases, you can use the getOne method, as follows:

```
f = filter(#glowfilter)
sprite(1).filterlist.append(f)
f = sprite(1).filterlist[1]-- here we get the actual value added to the list.
sprite(1).filterlist.getOne(f)
```

The third line in the above script adds the reference of the filter value to the list.

Parameters

value Required. Specifies the value associated with the position or property.

Example

This statement identifies the position of the value 12 in the linear list Answers, which consists of [10, 12, 15, 22]:

```
-- Lingo
put Answers.getOne(12)

// Javascript
trace(Answers.getOne(12));
```

The result is 2, because 12 is the second value in the list.

This statement identifies the property associated with the value 12 in the property list Answers, which consists of `[#a:10, #b:12, #c:15, #d:22]`:

```
-- Lingo
put Answers.getOne(12)

// Javascript
trace(Answers.getOne(12));
```

The result is #b, which is the property associated with the value 12.

See also

getPos()

getOSDirectory()

Usage

```
-- Lingo syntax
getOSDirectory()

// JavaScript syntax
getOSDirectory();
```

Description

Function; Returns the full path to the System folder (Mac) or Windows directory (Windows).

Parameters

None.

Example

This statement puts the OS directory of the machine. In this case its c:\windows.

```
-- Lingo
Put getOSDirectory()

// Javascript
trace(getOSDirectory());
```

See also

Fileio

getPixel()

Usage

```
-- Lingo syntax
imageObjRef.getPixel(x, y {, #integer})
imageObjRef.getPixel(point(x, y) {, #integer})

// JavaScript syntax
imageObjRef.getPixel(x, y {, #integer});
imageObjRef.getPixel(point(x, y) {, #integer});
```

Description

Image method. Returns an indexed or RGB color of the pixel at a specified point in a given image.

The index of the rows and columns of the returned image starts with 0. Therefore, in order to access the top left pixel of an image, specify the location as `(0,0)`, and not `(1,1)`. If a given image is *h* pixels high and *w* pixels wide, to access the bottom right pixel of the image, specify the location as `(w,1)`, `(h,1)`.

This method returns a value of 0 if the specified pixel is outside the given image.

To set a lot of pixels to the color of another pixel, it is faster to set them as raw numbers (by using the optional `#integer` parameter). Raw integer color values are also useful because they contain alpha layer information as well as color when the image is 32-bit. The alpha channel information can be extracted from the raw integer by dividing the integer by 2^8+8+8.

Parameters

x Required if specifying a pixel using x and y coordinates. An integer that specifies the x coordinate of the pixel.

y Required if specifying a pixel using x and y coordinates. An integer that specifies the y coordinate of the pixel.

#integer Optional. A symbol that specifies the raw number of the returned color value.

point*(x, y)* Required if specifying a pixel using a point. A point that specifies the point of the pixel.

Example

These statements get the color of the pixel at point (20, 20) in member Image stage.

```
-- Lingo
objImage = _movie.stage.image
objImage.getPixel(20, 20)
put (objImage)

-- Javascript
var objImage = _movie.stage.image ;
objImage.getPixel(20, 20) ;
put (objImage) ;
```

See also

color(), image(), power(), setPixel()

getPlayList()

Usage

```
-- Lingo syntax
soundChannelObjRef.getPlayList()

// JavaScript syntax
soundChannelObjRef.getPlayList();
```

Description

Sound Channel method; returns a copy of the list of queued sounds for a sound channel.

The returned list does not include the currently playing sound, nor may it be edited directly. You must use setPlayList().

The playlist is a linear list of property lists. Each property list corresponds to one queued sound cast member. Each queued sound may specify these properties:

Property	Description
#member	The sound cast member to queue. This property must be provided; all others are optional.
#startTime	The time within the sound at which playback begins, in milliseconds. The default is the beginning of the sound. See startTime.
#endTime	The time within the sound at which playback ends, in milliseconds. The default is the end of the sound. See endTime.
#loopCount	The number of times to play a loop defined with #loopStartTime and #loopEndTime. The default is 1. See loopCount.

Property	Description
#loopStartTime	The time within the sound to begin a loop, in milliseconds. See `loopStartTime`.
#loopEndTime	The time within the sound to end a loop, in milliseconds. See `loopEndTime`.
#preloadTime	The amount of the sound to buffer before playback, in milliseconds. See `preloadTime`.

Parameters

None.

Example

The following handler queues two sounds in sound channel 2, starts playing them, and then displays the playList in the message window. The playlist includes only the second sound queued, because the first sound is already playing.

```
-- Lingo syntax
on playMusic
    sound(2).queue(member("Chimes"))
    sound(2).queue([#member:member("introMusic"), #startTime:3000, #endTime:10000,
#loopCount:5, #loopStartTime:8000, #loopEndTime:8900])
    put(sound(2).getPlayList())
    sound(2).play()
end playMusic

// JavaScript syntax
function playMusic() {
    sound(2).queue(member("Chimes"));
    sound(2).queue(propList("member",member("introMusic"), "startTime",3000,
"endTime",10000, "loopCount",5, "loopStartTime",8000, "loopEndTime",8900));
    put(sound(2).getPlayList());
sound(2).play();
}
```

See also

endTime, loopCount, loopEndTime, loopStartTime, Member, member, preLoadTime, queue(), setPlayList(), Sound Channel, startTime

getPosition()

Usage

```
-- Lingo syntax
fileioObjRef.getPosition()

// JavaScript syntax
fileioObjRef.getPosition();
```

Description

Fileio method; Returns the position of a file.

Parameters

None.

Example

This statement opens the file c:\xtra.txt and gets the current position.

```
-- Lingo syntax
objFileio = new xtra("fileio")
objFileio.openFile("c:\xtra.txt",0)
put objFileio.getPosition()

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile("c:\xtra.txt",0);
trace(objFileio.getPosition());
```

See also

Fileio

getPref()

Usage

```
-- Lingo syntax
_player.getPref(stringPrefName)

// JavaScript syntax
_player.getPref(stringPrefName);
```

Description

Player method; retrieves the content of the specified file.

When you use this method, replace *stringPrefName* with the name of a file created by the `setPref()` method. If no such file exists, `getPref()` returns `VOID` (Lingo) or `null` (JavaScript syntax).

The filename used for *stringPrefName* must be a valid filename only, not a full path; Director supplies the path. The path to the file is handled by Director. The only valid file extensions for *stringPrefName1* are .txt and .htm; any other extension is rejected.

Do not use this method to access read-only or locked media.

Note: *In a browser, data written by `setPref()` is not private. Any movie with Shockwave content can read this information and upload it to a server. Confidential information should not be stored using `setPref()`.*

To see an example of `getPref()` used in a completed movie, see the Read and Write Text movie in the Learning/Lingo folder inside the Director application folder.

Parameters

stringPrefName Required. A string that specifies the file for which content is retrieved.

Example

This handler retrieves the content of the file Test and then assigns the file's text to the field Total Score:

```
-- Lingo syntax
on mouseUp
    theText = _player.getPref("Test")
    member("Total Score").text = theText
end

// JavaScript syntax
function mouseUp() {
    var theText = _player.getPref("Test");
```

```
    member("Total Score").text = theText;
}
```

See also

Player, setPref()

getPos()

Usage

```
list.getPos(value)
getPos(list, value)
```

Description

List function; identifies the position of a value in a list. When the specified value is not in the list, the `getPos` command returns the value 0.

For values contained in the list more than once, only the first occurrence is displayed. This command performs the same function as the `getOne` command when used for linear lists.

When you add a filter using the add or append method of the filterlist, a duplicate is created and added to the list. Methods like deleteOne, getPos, findPos, getOne use the exact value in the list, and not the duplicate value.

In such cases, you can use the getPos method to work as follows:

```
f = filter(#glowfilter)
sprite(1).filterlist.append(f)
f = sprite(1).filterlist[1]-- here we get the actual value added to the list.
sprite(1).filterlist.getPos(f)
```

The third line in the above script gets the reference of the filter value added to the list.

Parameters

value Required. Specifies the value associated with the position.

Example

This statement identifies the position of the value 12 in the list Answers, which consists of [#a:10, #b:12, #c:15, #d:22]:

```
-- Lingo
put Answers.getPos(12)

// Javascript
trace(Answers.getPos(12));
```

The result is 2, because 12 is the second value in the list.

See also

getOne()

getPref()

Usage

```
getPref(prefFileName)
```

Description

Function; retrieves the content of the specified file.

When you use this function, replace *prefFileName* with the name of a file created by the `setPref` function. If no such file exists, `getPref` returns `VOID`.

The filename used for *prefFileName* must be a valid filename only, not a full path; Director supplies the path. The path to the file is handled by Director. The only valid file extensions for *prefFileName* are .txt and .htm; any other extension is rejected.

Do not use this command to access read-only or locked media.

Note: *In a browser, data written by `setPref` is not private. Any movie with Shockwave content can read this information and upload it to a server. Confidential information should not be stored using `setPref`.*

To see an example of `getPref()` used in a completed movie, see the Read and Write Text movie in the Learning/Lingo folder inside the Director application folder.

Parameters

prefFileName Required. Specifies the file from which content is retrieved.

Example

This handler retrieves the content of the file Test and then assigns the file's text to the field Total Score:

```
on mouseUp
    theText = getPref("Test")
    member("Total Score").text = theText
end
```

See also

`setPref()`

getProp()

Usage

```
getProp(list, property)
list.property
```

Description

Property list function; identifies the value associated with a property in a property list.

Almost identical to the `getaProp` command, the `getProp` command displays an error message if the specified property is not in the list or if you specify a linear list.

Parameters

list Required. Specifies the property list from which *property* is retrieved.

property Required. Specifies the property with which the identified value is associated.

Example

This statement identifies the value associated with the property #c in the property list Answers, which consists of [#a:10, #b:12, #c:15, #d:22]:

```
-- Lingo
getProp(Answers, #c)

/ Javascript
Answers.getProp("c");
```

The result is 15, because 15 is the value associated with #c.

See also

getOne()

getPropAt()

Usage

```
list.getPropAt(index)
getPropAt(list, index)
```

Description

Property list function; for property lists only, identifies the property name associated with a specified position in a property list. If the specified item isn't in the list, or if you use `getPropAt()` with a linear list, a script error occurs.

Parameters

index Required. Specifies the index position of the property in the property list.

Example

This statement identifies the property 2nd in the property list Answers, which consists of `[#a:10, #b:12, #c:15, #d:22]`:

```
-- Lingo
put Answers.getPropAt(2)
-- #b

// Javascript
trace(Answers.getPropAt(2))
// b
```

getRendererServices()

Usage

```
getRendererServices()
getRendererServices().whichGetRendererServicesProperty
```

Description

3D command; returns the `rendererServices` object. This object contains hardware information and properties that affect all 3D sprites and cast members.

The `rendererServices` object has the following properties:

- `renderer` indicates the software rasterizer used to render all 3D sprites.
- `rendererDeviceList` returns a list of software rasterizers available on the user's system. Possible values include `#openGL`, `#directX5_2`, `#directX7_0`, `#directx9` and `#software`. The value of `renderer` must be one of these. This property can be tested but not set.
- `textureRenderFormat` indicates the pixel format used by the renderer. Possible values include `#rgba8888`, `#rgba8880`, `#rgba5650`, `#rgba5550`, `#rgba5551`, and `#rgba4444`. The four digits in each symbol indicate how many bits are used for each red, green, blue, and alpha component.
- `depthBufferDepth` indicates the bit depth of the hardware output buffer.
- `colorBufferDepth` indicates the bit depth of the color buffer. This property can be tested but not set.
- `modifiers` is a linear list of modifiers available for use by models in 3D cast members. Possible values include `#collision`, `#bonesPlayer`, `#keyframePlayer`, `#toon`, `#lod`, `#meshDeform`, `#sds`, `#inker`, and third-party Xtra-based modifiers. This property can be tested but not set.
- `primitives` is a linear list of primitive types available for use in the creation of new model resources. Possible values include `#sphere`, `#box`, `#cylinder`, `#plane`, `#particle`, and third-party Xtra-based primitive types. This property can be tested but not set.

Note: *For more detailed information about these properties, see the individual property entries.*

Parameters

None.

Example

This statement gets the rendererServices object and gets the renderer information.

```
-- Lingo
Ro = getRendererServices()
Put Ro.renderer
-- #directX7_0

// Javascript
Var ro = getRendererServices();
trace(ro.renderer);
// #directX7_0
```

See also

`renderer`, `preferred3dRenderer`, `active3dRenderer`, `rendererDeviceList`

getStreamStatus()

Usage

```
getStreamStatus(netID)
getStreamStatus(URLString)
```

Description

Function; returns a property list matching the format used for the globally available `tellStreamStatus` function that can be used with callbacks to sprites or objects. The list contains the following strings:

#URL	String containing the URL location used to start the network operation.
#state	String consisting of Connecting, Started, InProgress, Complete, "Error", or "NoInformation" (this last string is for the condition when either the net ID is so old that the status information has been dropped or the URL specified in `URLString` was not found in the cache).
#bytesSoFar	Number of bytes retrieved from the network so far.
#bytesTotal	Total number of bytes in the stream, if known. The value may be 0 if the HTTP server does not include the content length in the MIME header.
#error	String containing "" (`EMPTY`) if the download is not complete, OK if it completed successfully, or an error code if the download ended with an error.

For example, you can start a network operation with `getNetText()` and track its progress with `getStreamStatus()`.

Parameters

`netID` Required. A network operation that represents the stream of text to operate on.

Example

This statement displays in the message window the current status of a download begun with `getNetText()` and the resulting net ID placed in the variable `netID`:

```
-- Lingo
put getStreamStatus(netID)
-- [#URL: "www.adobe.com", #state: "InProgress", #bytesSoFar: 250, #bytesTotal: 50000,
#error: EMPTY]

// Javascript
trace(getStreamStatus(netID))
// <[#URL: "www.adobe.com", #state: "InProgress", #bytesSoFar: 250, #bytesTotal: 50000,
#error: EMPTY]>
```

See also

`on streamStatus`, `tellStreamStatus()`

getURL()

Usage

```
GetUrl(MUIObject, message, MovableOrNot)
```

Description

This function displays a dialog box for entering a URL and returns the URL that the user enters.

- *message* specifies the message that appears in the field for entering a URL. When the dialog box is first opened, this string is sent as a predefined value. When the user clicks a button, Lingo returns the string that the user entered. If the user clicks Cancel, the returned string is the same as the original value.
- On the Macintosh, *MovableOrNot* specifies whether the dialog box is movable. TRUE makes the dialog box movable. FALSE makes the dialog box not movable. The GetURL dialog box is always movable in Windows.

Example

These statements display a dialog box for entering a URL.

- The first statement creates an instance of the MUI Xtra, which is the object used as the dialog box.
- The second statement uses the GetUrl function to display a moveable dialog box for entering URLs and assigns the dialog box to the variable result. The message "Enter a URL here" appears in the dialog box's field for entering a URL.
- The final statements check whether the result after the user clicks a button is the same as the string sent when the dialog box opened. If it differs, the user entered a URL and clicked OK.

```
-- Lingo
set MUIObj = new (xtra "Mui")

set result = GetUrl(MUIObj, "Enter a URL", TRUE )

if objectP ( MUIObj) then
   set result = GetUrl( MUIObj, "Enter a URL", TRUE )
   if ( result <> "Enter a URL" ) then
      goToNetPage result
   end if
end if
```

getVal()

Usage

```
<float> Matrix.getVal(whichRow, whichColumn)
```

Description

Matrix method; gets the value of the specified element in the given matrix.

Parameters

whichRow Required. Row number of the element whose value is being read.

whichColumn Required. Column number of the element whose value is being read.

Example

The following function uses the getVal() method of a matrix to convert a matrix into a linear list.

```
--Lingo
on matrixToList(mat)
  rows = mat.numRows
  cols = mat.numColumns
  matrixList = []
  repeat with i = 1 to rows
    repeat with j = 1 to cols
      matrixList.append(mat.getVal(i,j))
    end repeat
  end repeat
  return matrixList
end

//Java Script
function matrixToList(mat)
{
  rows = mat.numRows;
  cols = mat.numColumns;
  matrixList = list();
  for( i = 1; i <= rows; i++)
  {
```

```
    for( j = 1; j <= cols; j++)
      matrixList.append(mat.getVal(i,j));
}
  return matrixList;
}
```

See also

setVal(), numRows(), numColumns(), matrixAddition(), matrixMultiply(), matrixMultiplyScalar(), matrixTranspose(), newMatrix()

getVariable()

Usage

```
-- Lingo syntax
spriteObjRef.getVariable(variableName {, returnValueOrReference})

// JavaScript syntax
spriteObjRef.getVariable(variableName {, returnValueOrReference});
```

Description

Function; returns the current value of the given variable from the specified Flash sprite. Flash variables were introduced in Flash version 4.

This function can be used in two ways.

Setting the optional *returnValueOrReference* parameter to TRUE (the default) returns the current value of the variable as a string. Setting the *returnValueOrReference* parameter to FALSE returns the current literal value of the Flash variable.

If the value of the Flash variable is an object reference, you must set the *returnValueOrReference* parameter to FALSE in order for the returned value to have meaning as an object reference. If it is returned as a string, the string will not be a valid object reference.

Parameters

variableName Required. Specifies the name of the variable whose value is returned.

returnValueOrReference Optional. Specifies whether the returned value is a string (TRUE) or as an object reference (FALSE).

Example

This statement sets the variable tValue to the string value of the Flash variable named gOtherVar in the Flash movie in sprite 3:

```
-- Lingo syntax
tValue = sprite(3).getVariable("gOtherVar", TRUE)
put(tValue) -- "5"
// JavaScript syntax
var tValue = sprite(3).getVariable("gOtherVar", true);
trace(tValue); // 5
```

This statement sets the variable tObject to refer to the same object that the variable named gVar refers to in the Flash movie in sprite 3:

```
-- Lingo syntax
tObject = sprite(3).getVariable("gVar",FALSE)
```

```
// JavaScript syntax
var tObject = sprite(3).getVariable("gVar",0);
```

This statement returns the value of the variable `currentURL` from the Flash cast member in sprite 3 and displays it in the Message window:

```
-- Lingo syntax
put(sprite(3).getVariable("currentURL"))

// JavaScript syntax
trace(sprite(3).getVariable("currentURL"));
```

See also

`setVariable()`

GetWidgetList()

Usage

```
GetWidgetList(MUIObject)
```

Description

This function returns a linear list of symbols for types of general purpose dialog box components supported for an instance of the MUI Xtra.

Example

This statement displays a list of widgets supported by MUIObject, which is an instance of the MUI Xtra:

```
-- Lingo
put GetWidgetList(MUIObject)

-- [#dividerV, #dividerH, #bitmap, #checkBox, #radioButton, #PopupList, #editText,
#WindowBegin, #WindowEnd, #GroupHBegin, #GroupHEnd, #GroupVBegin, #GroupVEnd, #label,
#IntegerSliderH, #FloatSliderH, #defaultPushButton, #cancelPushButton, #pushButton,
#toggleButton]
```

GetWindowPropList

Usage

```
GetWindowPropList(MUIObject)
```

Description

This function returns a list of the MUI Xtra's predefined settings for a general purpose dialog box's window.

When defining a new general purpose dialog box, use GetWindowPropList function to obtain a comprehensive list of dialog box properties and values and then edit individual properties as necessary. Besides being more convenient, this technique ensures compatibility with future versions of the MUI Xtra that may have additional properties.

These are the window properties and predefined values that GetWindowPropList returns:

Property	Predefined value
#type	#normal
#name	"window"
#callback	"nothing"
#mode	#data
#xPosition	100
#yPosition	120
#width	200
#height	210
#modal	1
#toolTips	0
#closeBox	1
#canZoom	0

Example

These statements define a new general purpose dialog box. The first statement assigns a list of predefined properties to the variable thePropList. Subsequent statements customize the dialog box by modifying these settings:

```
-- Lingo
set thePropList = GetWindowPropList(muiObject)

set the name of thePropList = "Picture Window"
set the callback of thePropList = "theWindowCallback"
set the mode of thePropList = #data
set the modal of thePropList = TRUE
set the closeBox of thePropList = FALSE
```

getWorldTransform()

Usage

```
member(whichCastmember).node(whichNode).getWorldTransform()
member(whichCastmember).node(whichNode).getWorldTransform().position
member(whichCastmember).node(whichNode).getWorldTransform().rotation
member(whichCastmember).node(whichNode).getWorldTransform().scale
```

Description

3D command; returns the world-relative transform of the model, group, camera, or light represented by `node`.

The `transform` property of a node is calculated relative to the transform of the node's parent, and is therefore parent-relative. The `getWorldTransform()` command calculates the node's transform relative to the origin of the 3D world, and is therefore world-relative.

Use `member(`*`whichCastmember`*`).node(`*`whichNode`*`).getWorldTransform().position` to find the position property of the node's world-relative transform. You can also use `worldPosition` as a shortcut for `getWorldTransform().position`.

Use `member(whichCastmember).node(whichNode).getWorldTransform().rotation` to find the rotation property of the node's world-relative transform.

Use `member(whichCastmember).node(whichNode).getWorldTransform().scale` to find the scale property of the node's world-relative transform.

These properties can be tested but not set.

Example

This statement shows the world-relative transform of the model named Box, followed by its position and rotation properties:

```
put member("3d world").model("Box").getworldTransform()
-- transform(1.000000,0.000000,0.000000,0.000000, 0.000000,1.000000,0.000000,0.000000,
0.000000,0.000000,1.000000,0.000000, - 94.144844,119.012825,0.000000,1.000000)
put member("3d world").model("Box"). getworldTransform().position
-- vector(-94.1448, 119.0128, 0.0000)
put member("3d world").model("Box"). getworldTransform().rotation
--vector(0.0000, 0.0000, 0.0000)

// Javascript
trace(member("3d world").getProp("model","Box").getworldTransform()
// <transform(1.000000,0.000000,0.000000,0.000000, 0.000000,1.000000,0.000000,0.000000,
0.000000,0.000000,1.000000,0.000000, -94.144844,119.012825,0.000000,1.000000)>
trace(member("3d world").getProp("model","Box"). getworldTransform().position
// <vector(-94.1448, 119.0128, 0.0000)>
trace(member("3d world").getProp("model","Box"). getworldTransform().rotation
//<vector(0.0000, 0.0000, 0.0000)>
```

See also

worldPosition, transform (property)

go()

Usage

```
-- Lingo syntax
_movie.go(frameNameOrNum {, movieName})

// JavaScript syntax
_movie.go(frameNameOrNum {, movieName});
```

Description

Movie method; causes the playhead to branch to a specified frame in a specified movie.

This method can be used to tell the playhead to loop to the previous marker, and is a convenient means of keeping the playhead in the same section of the movie while script remains active.

It is best to use marker labels for *frameNameOrNum* instead of frame numbers; editing a movie can cause frame numbers to change. Using marker labels also makes it easier to read scripts.

Calling `go()` with the *movieName* parameter loads frame 1 of the movie. If `go()` is called from within a handler, the handler in which it is placed continues executing.

When you specify a movie to play, specify its path if the movie is in a different folder, but to prevent a potential load failure, don't include the movie's .dir, .dxr, or .dcr file extension.

To more efficiently go to a movie at a URL, use the `downloadNetThing()` method to download the movie file to a local disk first, and then use the `go()` method with the *`movieName`* parameter to go to that movie on the local disk.

The `goLoop()` method sends the playhead to the previous marker in a movie, which is a convenient means of keeping the playhead in the same section of the movie while Lingo or JavaScript syntax remains active.

The following are reset when a movie is loaded: `beepOn` and `constraint` properties; `keyDownScript`, `mouseDownScript`, and `mouseUpScript`; `cursor` and `immediate` sprite properties; `cursor()` and `puppetSprite()` methods; and custom menus. However, the `timeoutScript` is not reset when loading a movie.

Parameters

`frameNameOrNum` Required. A string that specifies the marker label of the frame to which the playhead branches, or an integer that specifies the number of the frame to which the playhead branches.

`movieName` Optional. A string that specifies the movie that contains the frame specified by *`frameNameOrNum`*. This value must specify a movie file; if the movie is in another folder, *`movieName`* must also specify the path.

Example

This statement sends the playhead to the marker named start:

```
-- Lingo syntax
_movie.go("start")

// JavaScript syntax
_movie.go("start");
```

This statement sends the playhead to the marker named Memory in the movie named Noh Tale to Tell:

```
-- Lingo syntax
_movie.go("Memory", "Noh Tale to Tell")

// JavaScript syntax
_movie.go("Memory", "Noh Tale to Tell");
```

The following handler tells the movie to loop in the current frame. This handler is useful for making the movie wait in a frame while it plays so the movie can respond to events.

```
-- Lingo syntax
on exitFrame
    _movie.go(_movie.frame)
end

// JavaScript syntax
function exitFrame() {
    _movie.go(_movie.frame);
}
```

See also

`downloadNetThing`, `goLoop()`, `Movie`

goLoop()

Usage

```
-- Lingo syntax
_movie.goLoop()
```

```
// JavaScript syntax
_movie.goLoop();
```

Description

Movie method; sends the playhead to the previous marker in the movie, either one marker back from the current frame if the current frame does not have a marker, or to the current frame if the current frame has a marker.

If no markers are to the left of the playhead, the playhead branches to:

- The next marker to the right if the current frame does not have a marker.
- The current frame if the current frame has a marker.
- Frame 1 if the movie contains no markers.

Parameters

None.

Example

This statement causes the movie to loop between the current frame and the previous marker:

```
-- Lingo syntax
_movie.goLoop()

// JavaScript syntax
_movie.goLoop();
```

See also

go(), goNext(), goPrevious(), Movie

goNext()

Usage

```
-- Lingo syntax
_movie.goNext()

// JavaScript syntax
_movie.goNext();
```

Description

Movie method; sends the playhead to the next marker in the movie.

If no markers are to the right of the playhead, the playhead goes to the last marker in the movie or to frame 1 if there are no markers in the movie.

Parameters

None.

Example

This statement sends the playhead to the next marker in the movie:

```
-- Lingo syntax
_movie.goNext()

// JavaScript syntax
```

```
_movie.goNext();
```

See also

go(), goLoop(), goPrevious(), Movie

goPrevious()

Usage

```
-- Lingo syntax
_movie.goPrevious()

// JavaScript syntax
_movie.goPrevious();
```

Description

Movie method; sends the playhead to the previous marker in the movie.

This marker is two markers back from the current frame if the current frame does not have a marker or one marker back from the current frame if the current frame has a marker.

If no markers are to the left of the playhead, the playhead branches to one of the following:

- The next marker to the right if the current frame does not have a marker
- The current frame if the current frame has a marker
- Frame 1 if the movie contains no markers

Parameters

None.

Example

This statement sends the playhead to the previous marker in the movie:

```
-- Lingo syntax
_movie.goPrevious()

// JavaScript syntax
_movie.goPrevious();
```

See also

go(), goLoop(), goNext(), Movie

goToFrame()

Usage

```
-- Lingo syntax
spriteObjRef.goToFrame(frameNameOrNum)

// JavaScript syntax
spriteObjRef.goToFrame(frameNameOrNum);
```

Description

Command; plays a Flash movie sprite beginning at the frame identified by the *frameNumber* parameter. You can identify the frame by either an integer indicating a frame number or by a string indicating a label name. Using the goToFrame command has the same effect as setting a Flash movie sprite's frame property.

Example

The following handler branches to different points within a Flash movie in channel 5. It accepts a parameter that indicates which frame to go to.

```
-- Lingo syntax
on Navigate(whereTo)
    sprite(5).goToFrame(whereTo)
end

// JavaScript syntax
function Navigate(whereTo) {
    sprite(5).goToFrame(whereTo);
}
```

gotoNetMovie

Usage

```
gotoNetMovie URL
gotoNetMovie (URL)
```

Description

Command; retrieves and plays a new movie with Shockwave content from an HTTP or FTP server. The current movie continues to run until the new movie is available.

Only URLs are supported as valid parameters. The URL can specify either a filename or a marker within a movie. Relative URLs work if the movie is on an Internet server, but you must include the extension with the filename.

When performing testing on a local disk or network, media must be located in a directory named dswmedia.

If a gotoNetMovie operation is in progress and you issue a second gotoNetMovie command before the first is finished, the second command cancels the first.

Parameters

URL Required. Specifies the URL of the Shockwave content to play.

Example

In this statement, the URL indicates a Director filename:

```
gotoNetMovie "http://www.yourserver.com/movies/movie1.dcr"
```

In this statement, the URL indicates a marker within a filename:

```
gotoNetMovie "http://www.yourserver.com/movies/buttons.dcr#Contents"
```

In the following statement, gotoNetMovie is used as a function. The function returns the network ID for the operation.

```
myNetID = gotoNetMovie ("http://www.yourserver.com/movies/buttons.dcr#Contents")
```

gotoNetPage

Usage

```
gotoNetPage "URL", {"targetName"}
```

Description

Command; opens a movie with Shockwave content or another MIME file in the browser.

Only URLs are supported as valid parameters. Relative URLs work if the movie is on an HTTP or FTP server.

In the authoring environment, the `gotoNetPage` command launches the preferred browser if it is enabled. In projectors, this command tries to launch the preferred browser set with the Network Preferences dialog box or `browserName` command. If neither has been used to set the preferred browser, the `goToNetPage` command attempts to find a browser on the computer.

Parameters

`URL` Required. Specifies the URL of the movie with Shockwave content or MIME file to play.

`targetName` Optional. An HTML parameter that identifies the frame or window in which the page is loaded.

- If *`targetName`* is a window or frame in the browser, `gotoNetPage` replaces the contents of that window or frame.
- If *`targetName`* isn't a frame or window that is currently open, `goToNetPage` opens a new window. Using the string "_new" always opens a new window.
- If *`targetName`* is omitted, `gotoNetPage` replaces the current page, wherever it is located.

Example

The following script loads the file Newpage.html into the frame or window named frwin. If a window or frame in the current window called frwin exists, that window or frame is used. If the window frwin doesn't exist, a new window named frwin is created.

```
on keyDown
 gotoNetPage "Newpage.html", "frwin"
end
```

This handler opens a new window regardless of what window the browser currently has open:

```
on mouseUp
 gotoNetPage "Todays_News.html", "_new"
end
```

See also

`browserName()`, `netDone()`

group()

Usage

```
member(whichCastmember).group(whichGroup)
member(whichCastmember).group[index]
```

Description

3D element; a node in the 3D world that has a name, transform, parent, and children, but no other properties.

Every 3D cast member has a default group named World that cannot be deleted. The parent hierarchy of all models, lights, cameras, and groups that exist in the 3D world terminates in `group("world")`.

Example

This statement shows that the fourth group of the cast member newAlien is the group Direct01:

```
-- Lingo
put member("newAlien").group[4]

// Javascript
put member("newAlien").getPropRef("group",4) ;
-- group("Direct01")
```

See also

newGroup, deleteGroup, child (3D), parent

halt()

Usage

```
-- Lingo syntax
_movie.halt()

// JavaScript syntax
_movie.halt();
```

Description

Movie method; exits the current handler and any handler that called it and stops the movie during authoring or quits the projector during runtime from a projector.

Parameters

None.

Example

This statement checks whether the amount of free memory is less than 50K and, if it is, exits all handlers that called it and then stops the movie:

```
-- Lingo syntax
if (_system.freeBytes < (50*1024)) then
    _movie.halt()
end if

// JavaScript syntax
if (_system.freeBytes < (50*1024)) {
    _movie.halt();
}
```

See also

Movie

handler()

Usage

```
scriptObject.handler(#handlerSymbol)
```

Description

This function returns `TRUE` if the given *`scriptObject`* contains a specified handler, and `FALSE` if it does not. The script object must be a parent script, a child object, or a behavior.

Parameters

`symHandler` Required. Specifies the name of the handler.

Example

This code invokes a handler on an object only if that handler exists:

```
-- Lingo
if spiderObject.handler(#pounce) = TRUE then
    spiderObject.pounce()
end if

// Javascript
if (spiderObject.handler(symbol("pounce")) == true)
{
    spiderObject.pounce();
}
```

See also

`handlers()`, `new()`, `rawNew()`, `script()`

handlers()

Usage

```
scriptObject.handlers()
```

Description

This function returns a linear list of the handlers in the given *`scriptObject`*. Each handler name is presented as a symbol in the list. This function is useful for debugging movies.

You cannot get the handlers of a script cast member directly. You have to get them via the `script` property of the member.

Parameters

None.

Example

This statement displays the list of handlers in the child object RedCar in the Message window:

```
put RedCar.handlers()
-- [#accelerate, #turn, #stop]
```

This statement displays the list of handlers in the parent script member CarParentScript in the Message window:

```
put member("CarParentScript").script.handlers()
```

```
-- [#accelerate, #turn, #stop]
```

See also

handler(), script()

hilite (command)

Usage

```
fieldChunkExpression.hilite()
hilite fieldChunkExpression
```

Description

Command; highlights (selects) in the field sprite the specified chunk, which can be any chunk that Lingo lets you define, such as a character, word, or line. On the Mac, the highlight color is set in the Color control panel.

Parameters

None.

Example

This statement highlights the fourth word in the field cast member Comments, which contains the string Thought for the Day:

```
-- Lingo
member("Comments").word[4].hilite()

// Javascript
member("Comments").getPropRef("word",4).hilite() ;
```

See also

char...of, item...of, line...of, word...of, delete(), mouseChar, mouseLine, mouseWord, field, selection() (function), selEnd, selStart

hitTest()

Usage

```
-- Lingo syntax
spriteObjRef.hitTest(point)

// JavaScript syntax
spriteObjRef.hitTest(point);
```

Description

Function; indicates which part of a Flash movie is directly over a specific Director Stage location. The Director Stage location is expressed as a Director point value: for example, point(100,50). The `hitTest` function returns these values:

- `#background`—The specified Stage location falls within the background of the Flash movie sprite.
- `#normal`—The specified Stage location falls within a filled object.
- `#button`—The specified Stage location falls within the active area of a button.

- `#editText`—The specified Stage location falls within a Flash editable text field.

Parameters

`point` Required. Specifies the point to test.

Example

This frame script checks to see if the mouse is currently located over a button in a Flash movie sprite in channel 5 and, if it is, the script sets a text field used to display a status message:

```
-- Lingo syntax
on exitFrame
    if sprite(5).hitTest(_mouse.mouseLoc) = #button then
        member("Message Line").text = "Click here to play the movie."
        _movie.updatestage()
    else
        member("Message Line").text = ""
    end if
    _movie.go(_movie.frame)
end

// JavaScript syntax
function exitFrame() {
    var hT = sprite(5).hitTest(_mouse.mouseLoc);
    if (hT.toString() == "#button")
    {
        member("Message Line").text = "Click here to play the movie.";
        _movie.updatestage();
    } else {
        member("Message Line").text = "";
    }
    _movie.go(_movie.frame)
}
```

HMStoFrames()

Usage

```
HMStoFrames(hms, tempo, dropFrame, fractionalSeconds)
```

Description

Function; converts movies measured in hours, minutes, and seconds to the equivalent number of frames or converts a number of hours, minutes, and seconds into time if you set the *`tempo`* argument to 1 (1 frame = 1 second).

Parameters

`hms` Required. A string expression that specifies the time in the form `sHH:MM:SS.FFD`, where:

s	A character is used if the time is less than zero, or a space if the time is greater than or equal to zero.
HH	Hours.
MM	Minutes.
SS	Seconds.
FF	Indicates a fraction of a second if `fractionalSeconds` is `TRUE` or frames if `fractionalSeconds` is `FALSE`.
D	A d is used if *`dropFrame`* is `TRUE`, or a space if *`dropFrame`* is `FALSE`.

tempo Required. Specifies the tempo in frames per second.

dropFrame Required. Logical expression that determines whether the frame is a drop frame (TRUE) or not (FALSE). If the string *hms* ends in a *d*, the time is treated as a drop frame, regardless of the value of *dropFrame*.

fractionalSeconds Required. Logical expression that determines the meaning of the numbers after the seconds; they can be either fractional seconds rounded to the nearest hundredth of a second (TRUE) or the number of residual frames (FALSE).

Example

This statement determines the number of frames in a 1-minute, 30.1-second movie when the tempo is 30 frames per second. Neither the dropFrame nor fractionalSeconds arguments is used.

```
put HMStoFrames(" 00:01:30.10 ", 30, FALSE, FALSE)
-- 2710
```

This statement converts 600 seconds into minutes:

```
put framesToHMS(600, 1,0,0)
-- " 00:10:00.00 "
```

This statement converts an hour and a half into seconds:

```
put HMStoFrames("1:30:00", 1,0,0)
-- 5400
```

See also

framesToHMS()

hold()

Usage

```
-- Lingo syntax
spriteObjRef.hitTest(point)

// JavaScript syntax
spriteObjRef.hitTest(point);
```

Description

Flash command; stops a Flash movie sprite that is playing in the current frame, but any audio continues to play.

Parameters

None.

Example

This frame script holds the Flash movie sprites playing in channels 5 through 10 while allowing the audio for these channels to continue playing:

```
-- Lingo syntax
on enterFrame
    repeat with i = 5 to 10
        sprite(i).hold()
    end repeat
end
```

```
// JavaScript syntax
function enterFrame() {
    var i = 5;
    while (i < 11) {
        sprite(i).hold();
        i++;
    }
}
```

See also

playRate

identity()

Usage

```
member(whichCastmember).model(whichModel).transform.identity()
member(whichCastmember).group(whichGroup).transform.identity()
member(whichCastmember).camera(whichCamera).transform.identity()
sprite(whichSprite).camera{(index)}.transform.identity()
member(whichCastmember).light(whichLight).transform.identity()
transformReference.identity()
```

Description

3D command; sets the transform to the identity transform, which is `transform(1.0000,0.0000,0.0000,0.0000, 0.0000,1.0000,0.0000,0.0000, 0.0000,0.0000,1.0000,0.0000, 0.0000,0.0000,0.0000,1.0000)`.

The `position` property of the identity transform is `vector(0, 0, 0)`.

The `rotation` property of the identity transform is `vector(0, 0, 0)`.

The `scale` property of the identity transform is `vector(1, 1, 1)`.

The identity transform is parent-relative.

Parameters

None.

Example

This statement sets the transform of the model named Box to the identity transform:

```
-- Lingo
member("3d world").model("Box").transform.identity()

// Javascript
member("3d world").getProp("model",1).transform.identity()
```

This script assumes that Box is the first model available.

See also

transform (property), getWorldTransform()

idleLoadDone()

Usage

```
-- Lingo syntax
_movie.idleLoadDone(intLoadTag)

// JavaScript syntax
_movie.idleLoadDone(intLoadTag);
```

Description

Movie method; reports whether all cast members with the given tag have been loaded (`TRUE`) or are still waiting to be loaded (`FALSE`).

Parameters

`intLoadTag` Required. An integer that specifies the load tag for the cast members to test.

Example

This statement checks whether all cast members whose load tag is 20 have been loaded and then plays the movie Kiosk if they are:

```
-- Lingo syntax
if (_movie.idleLoadDone(20)) then
    _movie.play(1, "Kiosk")
end if

// JavaScript syntax
if (_movie.idleLoadDone(20)) {
    _movie.play(1, "Kiosk");
}
```

See also

`idleHandlerPeriod`, `idleLoadMode`, `idleLoadPeriod`, `idleLoadTag`, `idleReadChunkSize`, `Movie`

ignoreWhiteSpace()

Usage

```
XMLparserObject.ignoreWhiteSpace(trueOrFalse)
```

Description

XML Command; specifies whether the parser should ignore or retain white space when generating a Lingo list. When `ignoreWhiteSpace()` is set to `TRUE` (the default), the parser ignores white space.When set to `FALSE`, the parser will retain white space and treat it as actual data.

If an element has separate beginning and ending tags, such as `<sample> </sample>`, character data within the element will be ignored if, and only if, it is composed of white space only. If there is any non-white space, or if `ignoreWhiteSpace()` is set to `FALSE`, there will be a CDATA node with the exact text, including any white space.

Parameters

`trueOrFalse` Required. A value that specifies whether the parser should ignore white space (`TRUE`) or not (`FALSE`).

Example

These Lingo statements leave `ignoreWhiteSpace()` set to the default of `TRUE` and parse the given XML into a list. The element `<sample>` has no children in the list.

```
XMLtext = "<sample> </sample>"
parserObj.parseString(XMLtext)
theList = parserObj.makelist()
put theList
-- ["ROOT OF XML DOCUMENT": ["!ATTRIBUTES": [:], "sample": ["!ATTRIBUTES": [:]]]]
```

These Lingo statements set `ignoreWhiteSpace()` to `FALSE` and then parse the given XML into a list. The element `<sample>` now has a child containing one space character.

```
XMLtext = "<sample> </sample>"
parserObj.ignorewhitespace(FALSE)
parserObj.parseString(XMLtext)
theList = parserObj.makelist()
put theList
-- ["ROOT OF XML DOCUMENT": ["!ATTRIBUTES": [:], "sample": ["!ATTRIBUTES": [:], "!CHARDATA":
" "]]]
```

These Lingo statements leave `ignoreWhiteSpace()` set to the default of `TRUE` and parse the given XML. There is only one child node of the <sample> tag and only one child node of the `<sub>` tag.

```
XMLtext = "<sample> <sub> phrase 1 </sub></sample>"
parserObj.parseString(XMLtext)
theList = parserObj.makeList()
put theList
-- ["ROOT OF XML DOCUMENT": ["!ATTRIBUTES": [:], "sample": ["!ATTRIBUTES": [:], "sub":
["!ATTRIBUTES": [:], "!CHARDATA": " phrase 1 "]]]]
```

These Lingo statements set `ignoreWhiteSpace()` to `FALSE` and parse the given XML. There are now two child nodes of the `<sample>` tag, the first one being a single space character.

```
XMLtext = "<sample> <sub> phrase 1 </sub></sample>"
gparser.ignoreWhiteSpace(FALSE)
gparser.parseString(XMLtext)
theList = gparser.makeList()
put theList
-- ["ROOT OF XML DOCUMENT": ["!ATTRIBUTES": [:], "sample": ["!ATTRIBUTES": [:], "!CHARDATA":
" ", "sub": ["!ATTRIBUTES": [:], "!CHARDATA": " phrase 1 "]]]]
```

ilk()

Usage

```
ilk(object)
ilk(object, type)
```

Description

Function; indicates the type of an object.

The following table shows the return value for each type of object recognized by `ilk()`:

Type of Object	ilk(Object) returns	ilk(Object, Type) returns 1 only if Type =	Example
linear list	#list	#list or #linearlist	ilk ([1,2,3])
property list	#proplist	#list or #proplist	ilk ([#his: 1234, #hers: 7890])
integer	#integer	#integer or #number	ilk (333)
float	#float	#float or #number	ilk (123.456)
string	#string	#string	ilk ("asdf")
rect	#rect	#rect or #list	ilk (sprite(1).rect)
point	#point	#point or #list	ilk (sprite(1).loc)
color	#color	#color	ilk (sprite(1).color)
date	#date	#date	ilk (the systemdate)
symbol	#symbol	#symbol	ilk (#hello)
void	#void	#void	ilk (void)
picture	#picture	#picture	ilk (member (2).picture)
parent script instance	#instance	#object	ilk (new (script "blahblah"))
xtra instance	#instance	#object	ilk (new (xtra "fileio"))
member	#member	#object or #member	ilk (member 1)
xtra	#xtra	#object or #xtra	ilk (xtra "fileio")
script	#script	#object or #script	ilk (script "blahblah")
castlib	#castlib	#object or #castlib	ilk (castlib 1)
sprite	#sprite	#object or #sprite	ilk (sprite 1)
sound	#instance or #sound (when Sound Control Xtra is not present)	#instance or #sound	ilk (sound "yaddayadda")
window	#window	#object or #window	ilk (the stage)
media	#media	#object or #media	ilk (member (2).media)
timeout	#timeout	#object or #timeout	ilk (timeOut("intervalTimer"))
image	#image	#object or #image	ilk ((the stage).image)

Parameters

object Required. Specifies the object to test.

type Optional. Specifies the type to which *object* is compared. If the object is of the specified type, the `ilk()` function returns `TRUE`. If the object is not of the specified type, the `ilk()` function returns `FALSE`.

Example

The following `ilk` statement identifies the type of the object named Bids:

```
Bids = [:]
put ilk( Bids )
```

```
-- #proplist
```

The following ilk statement tests whether the variable `Total` is a list and displays the result in the Message window:

```
Total = 2+2
put ilk( Total, #list )
-- 0
```

In this case, since the variable `Total` is not a list, the Message window displays 0, which is the numeric equivalent of FALSE.

The following example tests a variable named `myVariable` and verifies that it is a date object before displaying it in the Message window:

```
myVariable = the systemDate
if ilk(myVariable, #date) then put myVariable
-- date( 1999, 2, 19 )
```

ilk (3D)

Usage

```
ilk(object)
ilk(object,type)
object.ilk
object.ilk(type)
```

Description

Lingo function; indicates the type of an object.

The following table shows the return value for each type of 3D object recognized by `ilk()`. See the main Lingo Dictionary for a list of return values of non-3D objects which are not discussed in this dictionary.

Type of object	ilk(object) returns	ilk(object, Type) if only Type =
render services	#renderer	#renderer
model resource	#modelresource, #plane, #box, #sphere, #cylinder, #particle, #mesh	Same as ilk(object), except for #modelresource which is the ilk of resources generated by an imported W3D file
model	#model	#model
motion	#motion	#motion or #list
shader	#shader	#shader or #list
texture	#texture	#texture or #list
group	#group	#group
camera	#camera	#camera
collision data	#collisiondata	#collisiondata
vector	#vector	#vector
transform	#transform	#transform

Parameters

object Required. Specifies the object to test.

type Optional. Specifies the type to which *object* is compared. If the object is of the specified type, the `ilk()` function returns `TRUE`. It the object is not of the specified type, the `ilk()` function returns `FALSE`.

Example

This statement shows that MyObject is a motion object:

```
put MyObject.ilk
-- #motion
```

The following statement tests whether MyObject is a motion object. The return value of 1 shows that it is.

```
put MyObject.ilk(#motion)
-- 1
```

See also

`tweenMode`

image()

Usage

```
-- Lingo syntax
image(intWidth, intHeight, intBitDepth)

// JavaScript syntax
image(intWidth, intHeight, intBitDepth);
```

Description

Top level function; creates and returns a new image with specified dimensions.

If you create a new image by using the top level `image()` function, the new image is a self-contained set of image data, and is independent of all other images. Therefore, changes made to any other images have no effect on the new image.

If you refer to an image by setting a variable equal to a source image, such as a cast member or the image of the Stage, the variable contains a reference to the source image. Therefore, a change made to the image in either the source object or the variable will be reflected in the other image.

To avoid this behavior and create a copy of an image that is independent of the source image, use the `duplicate()` method. The `duplicate()` method returns a copy of a source image that inherits all the values of the source image but is not tied to the source image. Therefore, a change made to either the source image or the new copy of the source image will have no effect on the other image.

If you create an image object by referring to a cast member, the new object contains a reference to the image of the member. Any changes made to the image are reflected in the cast member and in any sprites that are created from that member.

When you create a new image object, the background color defaults to white (color(255,255,255)), and the alpha channel is completely opaque (color(0,0,0)).

The alpha channel color for 100% transparency is white (color(255,255,255)); the alpha channel color for 100% opaque is black (color(0,0,0)).

To see an example of `image()` used in a completed movie, see the Imaging movie in the Learning/Lingo folder inside the Director application folder.

Parameters

`intWidth` Required. An integer that specifies the width of the new image.

`intHeight` Required. An integer that specifies the height of the new image.

`intBitDepth` Required. An integer that specifies the bit depth of the new image. Valid values are 1, 2, 4, 8, 16, or 32.

Example

The following example creates an 8-bit image that is 200 pixels wide by 200 pixels high.

```
-- Lingo syntax
objImage = image(200, 200, 8)

// JavaScript syntax
var objImage = image(200, 200, 8);
```

The following example creates an image by referring to the image of the Stage.

```
-- Lingo syntax
objImage = _movie.stage.image

// JavaScript syntax
var objImage = _movie.stage.image;
```

See also

`duplicate() (Image)`, `fill()`, `image (Image)`

importFileInto()

Usage

```
-- Lingo syntax
memberObjRef.importFileInto(fileOrUrlString, propertyList)

// JavaScript syntax
memberObjRef.importFileInto(fileOrUrlString, propertyList);
```

Description

Member method; replaces the content of a specified cast member with a specified file.

The `importFileInto()` method is useful in the following situations.

- When finishing or developing a movie, use it to embed external linked media so it can be edited during the project.
- When generating a Score from Lingo or JavaScript syntax during movie creation, use it to assign content to new cast members.
- When downloading files from the Internet, use it to download the file at a specific URL and set the filename of linked media.

> ***Note:*** *To import a file from a URL, it is usually more efficient to use the `preloadNetThing()` to download the file to a local disk first, and then import the file from the local disk. Using `preloadNetThing()` also minimizes any potential downloading issues.*

- Use it to import both RTF and HTML documents into text cast members with formatting and links intact.

Using `importFileInto()` in projectors can quickly consume available memory, so reuse the same members for imported data when possible.

In Director and projectors, `importFileInto()` automatically downloads the file. In Shockwave Player, call `preloadNetThing()` and wait for a successful completion of the download before using `importFileInto()` with the file.

Parameters

`fileOrUrlString` Required. A string that specifies the file that will replace the content of the cast member.

`propertyList` is optional and it supports the following properties:

#dither: 0; do not dither (default)

#dither: non-zero; dither

#trimWhiteSpace: 0; don't trim white space on outer edges of image

#trimWhiteSpace: non-zero; trim white space (default)

#linked: 0; import as an internal member (default)

#linked: non-zero; import as a linked member

#remapImageToStage: 0; use image's own depth

#remapImageToStage: non-zero; remap image to stage depth (default)

Example

This handler assigns a URL that contains a GIF file to the variable `tempURL` and then uses the `importFileInto` command to import the file at the URL into a new bitmap cast member:

```
-- Lingo syntax
on exitFrame
    tempURL = "http://www.dukeOfUrl.com/crown.gif"
    _movie.newMember(#bitmap).importFileInto(tempURL)
end

// JavaScript syntax
function exitFrame() {
    var tempURL = "http://www.dukeOfUrl.com/crown.gif";
    _movie.newMember("bitmap").importFileInto(tempURL);
}
```

This statement replaces the content of the sound cast member Memory with the sound file Wind:

```
-- Lingo syntax
member("Memory").importFileInto("Wind.wav")

// JavaScript syntax
member("Memory").importFileInto("Wind.wav");
```

These statements download an external file from a URL to the Director application folder and then import that file into the sound cast member Norma Desmond Speaks:

```
-- Lingo syntax

downLoadNetThing("http://www.cbDeMille.com/Talkies.AIF", _player.applicationPath &
"Talkies.AIF")
member("Norma Desmond Speaks").importFileInto(_player.applicationPath & "Talkies.AIF")
```

```
// JavaScript syntax
downLoadNetThing("http://www.cbDeMille.com/Talkies.AIF",_player.applicationPath +
"Talkies.AIF");
member("Norma Desmond Speaks").importFileInto(_player.applicationPath +"Talkies.AIF");
```

See also

downloadNetThing, fileName (Window), Member, preloadNetThing()

Initialize

Usage

```
Initialize (MUIObject, initialPropertyList)
```

Description

This command sets up a general purpose dialog box from an instance of the MUI Xtra. initialPropertyList is a property list that specifies where Director obtains definitions for the dialog box's attributes.

- The property list associated with the #windowPropList property is the list Director uses for definitions of the overall dialog box's attributes.
- The linear list associated with the #windowItemList property is the list Director uses for definitions of individual components. Each item in the list is a property list that defines one component.

Example

This statement initializes a general purpose dialog box created from MUIObject, which is an instance of the MUI Xtra. The list aWindowPropList contains definitions for the overall dialog box. The list aWindowItemList contains definitions for the dialog box's individual components:

```
-- Lingo
Initialize(MUIObj, [#windowPropList:aWindowPropList, \
#windowItemList:aWindowItemList])
```

insertBackdrop

Usage

```
sprite(whichSprite).camera{(index)}.insertBackdrop(index, texture, locWithinSprite,
rotation)
member(whichCastmember).camera(whichCamera).insertBackdrop(index, texture,
locWithinSprite, rotation)
```

Description

3D camera command; adds a backdrop to the camera's list of backdrops at a specified position in the list.

Parameters

index Required. Specifies the index position in the camera's list of backdrops to which the backdrop is added.

texture Required. Specifies the texture of the added backdrop.

locWithinSprite Required. A 2D loc at which the backdrop is displayed in the 3D sprite. This location is measured from the upper left corner of the sprite.

rotation Optional. Specifies the rotation of the added backdrop.

Example

The first line of this example creates a texture called Cedar. The second line inserts that texture at the first position in the list of backdrops of the camera of sprite 5. The backdrop is positioned at the point (300, 120), measured from the upper left corner of the sprite. It is rotated 45°.

```
-- Lingo
t1 = member("scene").texture("Cedar")
sprite(5).camera.insertBackdrop(1, t1, point(300, 120), 45)

// Javascript
Var t1= member(“scene”).getProp(“texture”,1);
Sprite(5).getPropRef(“camera”,1).insertBackDrop(1,t1,point(300,120),45);
```

See also

removeBackdrop, bevelDepth, overlay

insertFrame()

Usage

```
-- Lingo syntax
_movie.insertFrame()

// JavaScript syntax
_movie.insertFrame();
```

Description

Movie method; duplicates the current frame and its content.

The duplicate frame is inserted after the current frame and then becomes the current frame.

This method can be used only during a Score recording session and performs the same function as the `duplicateFrame()` method.

Parameters

None.

Example

The following handler generates a frame that has the transition cast member Fog assigned in the transition channel followed by a set of empty frames. The argument `numberOfFrames` sets the number of frames.

```
-- Lingo syntax
on animBall(numberOfFrames)
    _movie.beginRecording()
    _movie.frameTransition = member("Fog").number
    _movie.go(_movie.frame + 1)
    repeat with i = 0 to numberOfFrames
        _movie.insertFrame()
    end repeat
    _movie.endRecording()
end animBall
```

```
// JavaScript syntax
function animBall(numberOfFrames) {
    _movie.beginRecording();
    _movie.frameTransition = member("Fog").number;
    _movie.go(_movie.frame + 1);
    for (var i = 0; i <= numberOfFrames; i++) {
        _movie.insertFrame();
    }
    _movie.endRecording();
}
```

See also

duplicateFrame(), Movie

insertOverlay

Usage

```
sprite(whichSprite).camera{(index)}.insertOverlay(index, texture, locWithinSprite,
rotation)
member(whichCastmember).camera(whichCamera).insertOverlay(index, texture, locWithinSprite,
rotation)
```

Description

3D camera command; adds an overlay to the camera's list of overlays at a specified position in the list.

Parameters

index Required. Specifies the index position in the camera's list of overlays to which the overlay is added.

texture Required. Specifies the texture of the added overlay.

locWithinSprite Required. A 2D loc at which the overlay is displayed in the 3D sprite. This location is measured from the upper left corner of the sprite.

rotation Optional. Specifies the rotation of the added overlay.

Example

The first line of this example creates a texture named Cedar. The second line inserts that texture at the first position in the list of overlays of the camera of sprite 5. The overlay is positioned at the point (300, 120), measured from the upper left corner of the sprite. It is rotated 45°.

```
-- Lingo
t1 = member("scene").texture("Cedar")
sprite(5).camera.insertOverlay(1, t1, point(300, 120), 45)
```

```
// Javascript
Var t1=member("scene").getProp("texture",1);
Sprite(5).getPropRef("camera",1).insertOverlay(1,t1,point(300,120),45);
```

See also

removeOverlay, overlay, bevelDepth

inside()

Usage

```
point.inside(rectangle)
inside(point, rectangle)
```

Description

Function; indicates whether a specified point is within a specified rectangle (`TRUE`) or outside the rectangle (`FALSE`).

Parameters

`rectangle` Required. Specifies the rect that contains the point to test.

Example

This statement indicates whether the point Center is within the rectangle Zone and displays the result in the Message window:

```
-- Lingo
put Center.inside(Zone)

// Javascript
trace(Center. inside(Zone));
```

See also

`map()`, `mouseH`, `mouseV`, `point()`

installMenu

Usage

```
installMenu whichCastMember
```

Description

Command; installs the menu defined in the field cast member specified by *`whichCastMember`*. These custom menus appear only while the movie is playing. To remove the custom menus, use the `installMenu` command with no argument or with 0 as the argument. This command doesn't work with hierarchical menus.

For an explanation of how menu items are defined in a field cast member, see the `menu` keyword.

Avoid changing menus many times because doing so affects system resources.

In Windows, if the menu is longer than the screen, only part of the menu appears; on the Mac, menus longer than the screen can scroll.

Note: *Menus are not available in Shockwave Player.*

Parameters

`fieldMemberObjRef` Optional. Specifies the field cast member to which a menu is installed.

Example

This statement installs the menu defined in field cast member 37:

```
installMenu 37
```

This statement installs the menu defined in the field cast member named Menubar:

```
installMenu member "Menubar"
```

This statement disables menus that were installed by the installMenu command:

```
installMenu 0
```

See also

menu

integer()

Usage

```
(numericExpression).integer
integer(numericExpression)
```

Description

Function (Lingo only); rounds the value of an expression to the nearest whole integer.

You can force an integer to be a string by using the `string()` function.

In JavaScript syntax, use the `parseInt()` function.

Parameters

numericExpression Required. The number to round to an integer.

Example

This statement rounds off the number 3.75 to the nearest whole integer:

```
put integer(3.75)
-- 4
```

The following statement rounds off the value in parentheses. This provides a usable value for the `locH` sprite property, which requires an integer:

```
sprite(1).locH = integer(0.333 * stageWidth)
```

See also

float(), string()

integerP()

Usage

```
expression.integerP
(numericExpression).integerP
integerP(expression)
```

Description

Function (Lingo only); indicates whether a specified expression can be evaluated to an integer (`1 or TRUE`) or not (`0 or FALSE`). *P* in `integerP` stands for *predicate*.

Parameters

expression Required. The expression to test.

Example

This statement checks whether the number 3 can be evaluated to an integer and then displays 1 (TRUE) in the Message window:

```
put(3).integerP
-- 1
```

The following statement checks whether the number 3 can be evaluated to an integer. Because 3 is surrounded by quotation marks, it cannot be evaluated to an integer, so 0 (FALSE) is displayed in the Message window:

```
put("3").integerP
-- 0
```

This statement checks whether the numerical value of the string in field cast member Entry is an integer and if it isn't, displays an alert:

```
if field("Entry").value.integerP = FALSE then alert "Please enter an integer."
```

See also

floatP(), integer(), ilk(), objectP(), stringP(), symbolP()

Interface()

Usage

```
xtra("XtraName").Interface()
Interface(xtra "XtraName")
```

Description

Function; returns a Return-delimited string that describes the Xtra and lists its methods. This function replaces the now obsolete mMessageList function.

Parameters

None.

Example

This statement displays the output from the function used in the QuickTime Asset Xtra in the Message window:

```
-- Lingo
put Xtra("QuickTimeSupport").Interface()

// Javascript
trace(xtra("QuickTimeSupport").Interface());
```

interpolate()

Usage

```
transform1.interpolate(transform2,percentage)
```

Description

3D transform method; returns a copy of *transform1* created by interpolating from the position and rotation of *transform1* to the position and rotation of *transform2* by the specified percentage. The original *transform1* is not affected. To interpolate *transform1*, use interpolateTo().

To interpolate by hand, multiply the difference of two numbers by the percentage. For example, interpolation from 4 to 8 by 50 percent yields 6.

Example

In this example, tBox is the transform of the model named Box, and tSphere is the transform of the model named Sphere. The third line of the example interpolates a copy of the transform of Box halfway to the transform of Sphere.

```
-- Lingo
tBox = member("3d world").model("Box").transform
tSphere = member("3d world").model("Sphere").transform
tNew = tBox.interpolate(tSphere, 50)

// Javascript
var tBox = member("3d world").getPropRef("model",a).transform;
// where a is the number index for the model "Box"
var tSphere = member("3d world").getPropRef("model", i).transform;
// where i is the number index for the model "Sphere"
var tNew = tBox.interpolate(tSphere, 50);
```

See also

`interpolateTo()`

interpolateTo()

Usage

```
transform1.interpolateTo(transform2, percentage)
```

Description

3D `transform` method; modifies *`transform1`* by interpolating from the position and rotation of *`transform1`* to the position and rotation of a new transform by a specified percentage. The original *`transform1`* is changed. To interpolate a copy of *`transform1`*, use the `interpolate()` function.

To interpolate by hand, multiply the difference of two numbers by the percentage. For example, interpolation from 4 to 8 by 50 percent yields 6.

Parameters

`transform2` Required. Specifies the transform to which a given transform is interpolated.

`percentage` Required. Specifies the rotation percentage of *`transform2`*.

Example

In this example, tBox is the transform of the model named Box, and tSphere is the transform of the model named Sphere. The third line of the example interpolates the transform of Box halfway to the transform of Sphere.

```
-- Lingo
tBox = member("3d world").model("Box").transform
tSphere = member("3d world").model("Sphere").transform
tBox.interpolateTo(tSphere, 50)

// Javascript
var tBox = member("3d world").getPropRef("model", i).transform;
// where i the number index for the model "Box"
var tSphere = member("3d world").getPropRef("model",j).transform;
// where j is the number index for the model "Sphere"
```

```
tBox.interpolateTo(tSphere, 50);
```

See also

interpolate()

intersect()

Usage

```
rectangle1. Intersect(rectangle2)
intersect(rectangle1, rectangle2)
```

Description

Function; determines the rectangle formed where two rectangles intersect.

Parameters

rectangle2 Required. Specifies the second rectangle in the intersection test.

Example

This statement assigns the variable `newRectangle` to the rectangle formed where rectangle toolKit intersects rectangle Ramp:

```
-- Lingo
newRectangle = toolKit.intersect(Ramp)

// Javascript
newRectangle = toolKit.intersect(Ramp);
```

See also

map(), rect(), union()

inverse()

Usage

```
member(whichCastmember).model(whichModel).transform.inverse()
member(whichCastmember).group(whichGroup).transform.inverse()
member(whichCastmember).camera(whichCamera).transform.inverse()
sprite(whichSprite).camera{(index)}.transform.inverse()
member(whichCastmember).light(whichLight).transform.inverse()
transformReference.inverse()
```

Description

3D `transform` method; returns a copy of the transform with its position and rotation properties inverted.

This method does not change the original transform. To invert the original transform, use the `invert()` function.

Parameters

None.

Example

This statement inverts a copy of the transform of the model named Chair

```
-- Lingo
boxInv = member("3d world").model("Chair").transform.inverse()

// Javascript
var boxInv = member("3d world").getPropRef("model", a).transform.inverse();
// where a the number index for the model "Chair"
```

See also

invert()

invert()

Usage

```
member(whichCastmember).model(whichModel).transform.invert()
member(whichCastmember).group(whichGroup).transform.invert()
member(whichCastmember).camera(whichCamera).transform.invert()
sprite(whichSprite).camera{(index)}.transform.invert()
member(whichCastmember).light(whichLight).transform.invert()
transformReference.invert()
```

Description

3D `transform` method; inverts the position and rotation properties of the transform.

This method changes the original transform. To invert a copy of the original transform, use the `inverse()` function.

Parameters

None.

Example

```
-- Lingo
member("3d world").model("Box").transform.invert()

// Javascript
member("3d world").getPropRef("model", a).transform.invert();
// where a the number index for the model "Box"
```

See also

inverse()

isBusy()

Usage

```
-- Lingo syntax
soundChannelObjRef.isBusy()

// JavaScript syntax
soundChannelObjRef.isBusy();
```

Description

Sound Channel method; determines whether a sound is playing (`TRUE`) or not playing (`FALSE`) in a sound channel.

Make sure that the playhead has moved before using `isBusy()` to check the sound channel. If this function continues to return `FALSE` after a sound should be playing, add the `updateStage()` method to start playing the sound before the playhead moves again.

This method works for those sound channels occupied by actual audio cast members. QuickTime, Flash, and Shockwave Player audio handle sound differently, and this method will not work with those media types.

Consider using the `status` property of a sound channel instead of `isBusy()`. The `status` property can be more accurate under many circumstances.

Parameters

None.

Example

The following statement checks whether a sound is playing in sound channel 1 and loops in the frame if it is. This allows the sound to finish before the playhead goes to another frame.

```
-- Lingo syntax
if (sound(1).isBusy()) then
    _movie.go(_movie.frame)
end if

// JavaScript syntax
if (sound(1).isBusy()) {
    _movie.go(_movie.frame);
}
```

See also

`status`, `Sound Channel`

isInWorld()

Usage

```
member(whichCastmember).model(whichModel).isInWorld()
member(whichCastmember).camera(whichCamera).isInWorld()
member(whichCastmember).light(whichLight).isInWorld()
member(whichCastmember).group(whichGroup).isInWorld()
```

Description

3D command; returns a value of `TRUE` if the parent hierarchy of the model, camera, light, or group terminates in the world. If the value of `isInWorld` is `TRUE`, the model, camera, light, or group functions in the 3D world of the cast member.

Models, cameras, lights, and groups can be stored in a 3D cast member but not used in the 3D world of the cast member. Use the `addToWorld` and `removeFromWorld` commands to add and remove models, cameras, lights, and groups from the 3D world of the cast member.

Parameters

None.

Example

This statement shows that the model named Teapot exists in the 3D world of the cast member named TableScene:

```
--Lingo
put member("TableScene").model("Teapot").isInWorld()

// Javascript
put member("TableScene").getPropRef("model", a).isInWorld() ;
// where a is the member index for the Teapot model.
-- 1
```

See also

addToWorld, removeFromWorld, child (3D)

isPastCuePoint()

Usage

```
-- Lingo syntax
spriteObjRef.isPastCuePoint(cuePointID)

// JavaScript syntax
spriteObjRef.isPastCuePoint(cuePointID);
```

Description

Function; determines whether a sprite or sound channel has passed a specified cue point in its media. This function can be used with sound (WAV, AIFF, SND, SWA, AU), QuickTime, or Xtra files that support cue points.

Replace *spriteNum* or *channelNum* with a sprite channel or a sound channel. Shockwave Audio (SWA) sounds can appear as sprites in sprite channels, but they play sound in a sound channel. It is recommended that you refer to SWA sound sprites by their sprite channel number rather than their sound channel number.

Replace *cuePointID* with a reference for a cue point:

- If *cuePointID* is an integer, `isPastCuePoint` returns 1 if the cue point has been passed and 0 if it hasn't been passed.
- If *cuePointID* is a name, `isPastCuePoint` returns the number of cue points passed that have that name.

If the value specified for *cuePointID* doesn't exist in the sprite or sound, the function returns 0.

The number returned by `isPastCuePoint` is based on the absolute position of the sprite in its media. For example, if a sound passes cue point Main and then loops and passes Main again, `isPastCuePoint` returns 1 instead of 2.

When the result of `isPastCuePoint` is treated as a Boolean operator, the function returns `TRUE` if any cue points identified by *cuePointID* have passed and `FALSE` if no cue points are passed.

Parameters

cuePointID Required. A string or integer that specifies the name or number of the specified cue point.

Example

This statement plays a sound until the third time the cue point Chorus End is passed:

```
-- Lingo syntax
if (sound(1).isPastCuePoint("Chorus End")=3) then
 sound(1).stop()
end if

// JavaScript syntax
var ce = sound(1).isPastCuePoint("Chorus End");
```

```
if (ce = 3) {
 sound(1).stop();
}
```

The following example displays information in cast member "field 2" about the music playing in sound channel 1. If the music is not yet past cue point "climax", the text of "field 2" is "This is the beginning of the piece." Otherwise, the text reads "This is the end of the piece."

```
--- Lingo syntax
if not sound(1).isPastCuePoint("climax") then
    member("field 2").text = "This is the beginning of the piece."
else
    member("field 2").text = "This is the end of the piece."
end if

// JavaScript syntax
var cmx = sound(1).isPastCuePoint("climax");
if (cmx != 1) {
    member("field 2").text = "This is the beginning of the piece.";
} else {
    member("field 2").text = "This is the end of the piece.";
}
```

ItemUpdate()

Usage

```
ItemUpdate(MUIObject, itemNumber, itemInputPropList)
```

Description

This command updates a component in a general purpose dialog box. It is useful for updating a dialog box in response to user actions while the dialog box is displayed.

- itemNumber represents the number of the item being updated.
- itemInputPropList represents the list of new properties for the item.

The ItemUpdate command can be used for many things; possible uses include enabling or disabling buttons, changing the range of a pop-up, updating a sliders position, and updating editable text items if the user enters an invalid value.

You may want to update individual items in a dialog box depending on user input, user interaction, or to display underlying data. Although you would typically update an item's #value, you can also update everything else about an item, except for its type. Set the height, width, locH, and locV properties to -1 to keep their current values.

Example

These statements update the dialog box component that has the number itemNum.

- The first statement obtains the component's definitions from the overall list of item definitions.
- The second and third statements modify the component's type and attribute properties.
- The last statement uses the ItemUpdate command to update the component's settings.

```
--Lingo
set baseItemList = getAt ( theItemList, itemNum )
set the type of baseItemList = #IntegerSliderH
set the attributes of baseItemList = [#valueRange :[#min:1, #max:8, #increment:1,
#jump:1, #acceleration:1]
```

```
ItemUpdate(MUIObj, itemNum, baseItemList)

on smileyUpdate
   -- declare globals
   global smileyIndex, gMuiSmileDialObj, itemNumSmile, itemNumSlide, smileItemList

-- validate dialog object
if ( objectP ( gMuiSmileDialObj ) ) then
   -- get a list to put in new/updated values
   set baseItemList = duplicate ( getAt ( smileItemList,itemNumSmile ) )
   -- metrics can be set to -1, this "keeps them the same"
   -- instead of updating.
   -- could also be set to a new value if you
   -- wanted to resize the item or relocate it.
   set the width of baseItemList = -1 -- keep previous
   set the height of baseItemList = -1 -- keep previous
   set the locH of baseItemList = -1 -- keep previous
   set the locV of baseItemList = -1 -- keep previous

   -- in this particular case, the value is
   -- the only thing that's changing
   set the value of baseItemList = string(smileyIndex)
   -- member name

   -- tell the dialog to update the item number
   -- with the new item list
   ItemUpdate(gMuiSmileDialObj, itemNumSmile, baseItemList)
   end if
end
```

keyPressed()

Usage

```
-- Lingo syntax
_key.keyPressed({keyCodeOrCharacter})

// JavaScript syntax
_key.keyPressed({keyCodeOrCharacter});
```

Description

Key method; returns the character string assigned to the key that was last pressed, or optionally whether a specified key was pressed.

If the *keyCodeOrCharacter* parameter is omitted, this method returns the character string assigned to the last key that was pressed. If no key was pressed, this method returns an empty string.

If the *keyCodeOrCharacter* is used to specify the key being pressed, this method returns `TRUE` if that particular key is being pressed, or `FALSE` if not.

This method is updated when the user presses keys while in a `repeat` (Lingo) or `for` (JavaScript syntax) loop. This is an advantage over the `key` property, which doesn't update while in a `repeat` or `for` loop.

To test which characters correspond to different keys on different keyboards, use the Keyboard Lingo sample movie.

Parameters

keyCodeOrCharacter Optional. The key code or ASCII character string to test.

Example

The following statement checks whether the user pressed the Enter key in Windows or the Return key on a Mac and runs the handler `updateData` if the key was pressed:

```
-- Lingo syntax
if (_key.keyPressed(RETURN)) then
    updateData
end if

// JavaScript syntax
if (_key.keyPressed(36)) {
    updateData();
}
```

This statement uses the keyCode for the *a* key to test if it's down and displays the result in the Message window:

```
-- Lingo syntax
if (_key.keyPressed(0)) then
    put("The key is down")
end if

// JavaScript syntax
if (_key.keyPressed(0)) {
    put("The key is down");
}
```

This statement uses the ASCII strings to test if the *a* and *b* keys are down and displays the result in the Message window:

```
-- Lingo syntax
if (_key.keyPressed("a") and _key.keyPressed("b")) then
    put("Both keys are down")
end if

// JavaScript syntax
if (_key.keyPressed("a") && _key.keyPressed("b")) {
    put("Both keys are down");
}
```

See also

Key, key, keyCode

label()

Usage

```
-- Lingo syntax
_movie.label(stringMarkerName)

// JavaScript syntax
_movie.label(stringMarkerName);
```

Description

Movie method; indicates the frame associated with a marker label.

The parameter *stringMarkerName* should be a label in the current movie; if it's not, this method returns 0.

Parameters

stringMarkerName Required. A string that specifies the name of the marker label associated with a frame.

Example

This statement sends the playhead to the tenth frame after the frame labeled Start:

```
-- Lingo syntax
_movie.go(_movie.label("Start") + 10)

// JavaScript syntax
_movie.go(_movie.label("Start") + 10);
```

See also

frameLabel, go(), labelList, Movie

last()

Usage

```
the last chunk of ( chunkExpression )
the last chunk in (chunkExpression)
```

Description

Function; identifies the last chunk in a chunk expression.

Chunk expressions refer to any character, word, item, or line in a container of character. Supported containers are field cast members, variables that hold strings, and specified characters, words, items, lines, and ranges within containers.

Parameters

chunkExpression Required. Specifies the chunk expression that contains the last chunk.

Example

This statement identifies the last word of the string "Adobe, the multimedia company" and displays the result in the Message window:

```
-- Lingo
put the last word of "Adobe, the multimedia company"
```

The result is the word *company*.

This statement identifies the last character of the string "Adobe, the multimedia company" and displays the result in the Message window:

```
put last char("Adobe, the multimedia company")
```

The result is the letter *y*.

See also

char...of, word...of

lastClick()

Usage

```
the lastClick
```

Description

Function; returns the time in ticks (1 tick = 1/60 of a second) since the mouse button was last pressed.

This function can be tested but not set.

Parameters

None.

Example

This statement checks whether 10 seconds have passed since the last mouse click and, if so, sends the playhead to the marker No Click:

```
if the lastClick > 10 * 60 then go to "No Click"
```

See also

lastEvent(), lastKey, lastRoll, milliseconds

lastEvent()

Usage

```
the lastEvent
```

Description

Function; returns the time in ticks (1 tick = 1/60 of a second) since the last mouse click, rollover, or key press occurred.

Parameters

None.

Example

This statement checks whether 10 seconds have passed since the last mouse click, rollover, or key press and, if so, sends the playhead to the marker Help:

```
if the lastEvent > 10 * 60 then go to "Help"
```

See also

lastClick(), lastKey, lastRoll, milliseconds

length()

Usage

```
string.length
length(string)
```

Description

Function; returns the number of characters in the string specified by *string*, including spaces and control characters such as TAB and RETURN.

Parameters

None.

Example

This statement displays the number of characters in the string "Macro"&"media":

```
put ("Macro" & "media").length
-- 10
```

This statement checks whether the content of the field cast member Filename has more than 31 characters and if it does, displays an alert:

```
-- Lingo syntax
if member("Filename").text.length > 31 then
    alert "That filename is too long."
end if

// JavaScript syntax
if (member("Filename").text.length > 31) {
    _player.alert("That filename is too long.");
}
```

See also

chars(), offset() (string function)

light()

Usage

```
member(whichCastmember).light(whichLight)
member(whichCastmember).light[index]
member(whichCastmember).light(whichLight).whichLightProperty
member(whichCastmember).light[index].whichLightProperty
```

Description

3D element; an object at a vector position from which light emanates.

For a complete list of light properties and commands, see the Using Director topics in the Director Help Panel.

Example

This example shows the two ways of referring to a light. The first line uses a string in parentheses and the second line uses the a number in brackets. The string is the light's name and the number is the position of the light in the cast member's list of lights.

```
-- Lingo
thisLight = member("3D World").light("spot01")
thisLight = member("3D World").light[2]

// Javascript
thisLight = member("3D World").light[1];
```

See also

newLight, deleteLight

lineHeight()

Usage

```
-- Lingo syntax
memberObjRef.lineHeight(lineNumber)

// JavaScript syntax
memberObjRef.lineHeight(lineNumber);
```

Description

Function; returns the height, in pixels, of a specific line in a specified field cast member.

Parameters

lineNumber Required. An integer that specifies the line to measure.

Example

This statement determines the height, in pixels, of the first line in the field cast member Today's News and assigns the result to the variable `headline`:

```
--Lingo syntax
headline = member("Today's News").lineHeight(1)

// JavaScript syntax
var headline = member("Today's News").lineHeight(1);
```

linePosToLocV()

Usage

```
-- Lingo syntax
memberObjRef.linePosToLocV(lineNumber)

// JavaScript syntax
memberObjRef.linePosToLocV(lineNumber);
```

Description

Function; returns a specific line's distance, in pixels, from the top edge of the field cast member.

Parameters

lineNumber Required. An integer that specifies the line to measure.

Example

This statement measures the distance, in pixels, from the second line of the field cast member Today's News to the top of the field cast member and assigns the result to the variable `startOfString`:

```
--Lingo syntax
startOfString = member("Today's News").linePosToLocV(2)

// JavaScript syntax
```

```
var startOfString = member("Today's News").linePosToLocV(2);
```

linkAs()

Usage

```
castMember.linkAs()
```

Description

Script cast member function; opens a save dialog box, allowing you to save the contents of the script to an external file. The script cast member is then linked to that file.

Linked scripts are imported into the movie when you save it as a projector or a movie with Shockwave content. This differs from other linked media, which remains external to the movie unless you explicitly import it.

Parameters

None.

Example

These statements, typed in the Message window, opens a Save dialog box to save the script Random Motion as an external file:

```
-- Lingo
member("Random Motion").linkAs()
importFileInto, linked

// Javascript
member("Random Motion").linkAs() ;
```

list()

Usage

```
-- Lingo syntax
list()
[]
list(stringValue1, stringValue2, ...)
[stringValue1, stringValue2, ...]

// JavaScript syntax
list();
list(stringValue1, stringValue2, ...);
```

Description

Top level function; creates a linear list.

When creating a list using the syntax `list()`, with or without parameters, the index of list values begins with 1.

When creating a list using the syntax `[]`, with or without parameters, the index of list values begins with 0.

The maximum length of a single line of executable script is 256 characters. Large lists cannot be created using `list()`. To create a list with a large amount of data, enclose the data in square brackets ([]), put the data into a field, and then assign the field to a variable. The variable's content is a list of the data.

Parameters

strigValue1, stringValue2 ... Optional. A list of strings that specify the initial values in the list.

Example

This statement sets the variable named `designers` equal to a linear list that contains the names Gee, Kayne, and Ohashi:

```
-- Lingo syntax
designers = list("Gee", "Kayne", "Ohashi") -- using list()
designers = ["Gee", "Kayne", "Ohashi"] -- using brackets

// JavaScript syntax
var designers = list("Gee", "Kayne", "Ohashi");
```

See also

`propList()`

listP()

Usage

```
listP(item)
```

Description

Function; indicates whether a specified item is a list, rectangle, or point (1 `or TRUE`) or not (0 `or FALSE`).

Parameters

item Required. Specifies the item to test.

Example

This statement checks whether the list in the variable `designers` is a list, rectangle, or point, and displays the result in the Message window:

```
put listP(designers)
```

The result is 1, which is the numerical equivalent of `TRUE`.

See also

`ilk()`, `objectP()`

loadFile()

Usage

```
member(whichCastmember).loadFile(fileName {, overwrite, generateUniqueNames})
```

Description

3D cast member command; imports the assets of a W3D file into a cast member.

The cast member's `state` property must be either -1 (error) or 4 (loaded) before the `loadFile` command is used.

Parameters

fileName Required. Specifies the W3D file that contains the assets to import.

overwrite Optional. Indicates whether the assets of the W3D file replace the assets of the cast member (TRUE) or are added to the assets of the cast member (FALSE). The default value of *overwrite* is TRUE.

generateUniqueNames Optional. If set to TRUE, any element in the W3D file with the same name as a corresponding element in the cast member is renamed. If FALSE, elements in the cast member are overwritten by corresponding elements in the W3D file with the same name. The default value of *generateUniqueNames* is TRUE.

Example

The following statement imports the contents of the file named Truck.W3d into the cast member named Roadway. The contents of Truck.W3d will be added to the contents of Roadway. If any imported objects have the same names as objects already in Roadway, Director will create new names for them.

```
-- Lingo
member("Roadway").loadFile("Truck.W3d", FALSE, TRUE)

// Javascript
member("Roadway").loadFile("Truck.W3d",false,true);
```

The following statement imports the contents of the file named Chevy.W3d into the cast member named Roadway. Chevy.W3d is in a folder named Models one level down from the movie. The contents of Roadway will be replaced by the contents of Chevy.W3d. The third parameter is irrelevant because the value of the second parameter is TRUE.

```
-- Lingo
member("Roadway").loadFile(the moviePath & "Models\Chevy.W3d", TRUE, TRUE)

// Javascript
member("Roadway").loadFile(_movie.Path + "Models\Chevy.W3d",true,true);
```

See also

state (3D)

locToCharPos()

Usage

```
-- Lingo syntax
memberObjRef.locToCharPos(location)

// JavaScript syntax
memberObjRef.locToCharPos(location);
```

Description

Function; returns a number that identifies which character in a specified field cast member is closest to a point within the field.

The value 1 corresponds to the first character in the string, the value 2 corresponds to the second character in the string, and so on.

Parameters

location Required. A point within the field cast member. The value for *location* is a point relative to the upper left corner of the field cast member.

Example

The following statement determines which character is closest to the point 100 pixels to the right and 100 pixels below the upper left corner of the field cast member Today's News. The statement then assigns the result to the variable `PageDesign`.

```
--Lingo syntax
pageDesign = member("Today's News").locToCharPos(point(100, 100))

// JavaScript syntax
var pageDesign = member("Today's News").locToCharPos(point(100, 100));
```

locVToLinePos()

Usage

```
-- Lingo syntax
memberObjRef.locVToLinePos(locV)

// JavaScript syntax
memberObjRef.locVToLinePos(locV);
```

Description

Function; returns the number of the line of characters that appears at a specified vertical position.

Parameters

`locV` Required. Specifies the vertical position of the line of characters. This value is the number of pixels from the top of the field cast member, not the part of the field cast member that currently appears on the Stage.

Example

This statement determines which line of characters appears 150 pixels from the top of the field cast member Today's News and assigns the result to the variable `pageBreak`:

```
--Lingo syntax
pageBreak = member("Today's News").locVToLinePos(150)

// JavaScript syntax
var pageBreak = member("Today's News").locVToLinePos(150);
```

log()

Usage

```
log(number)
```

Description

Math function (Lingo only); calculates the natural logarithm of a specified number.

In JavaScript syntax, use the Math object's `log()` function.

Parameters

`number` Required. A number from which the natural logarithm is calculated. This number must be a decimal number greater than 0.

Example

This statement assigns the natural logarithm of 10.5 to the variable `Answer`.

```
-- Lingo
Answer = log(10.5)

// Javascript
Answer = Math.log(10.5);
```

Example

This statement calculates the natural logarithm of the square root of the value `Number` and then assigns the result to the variable `Answer`:

```
-- Lingo
Answer = log(Number.sqrt)

// Javascript
Answer = Math.log(Math.sqrt(Number));
```

makeList()

Usage

```
--Lingo syntax
parserObject.makeList()

// JavaScript syntax
parserObject.makeList();
```

Description

Function; returns a property list based on the XML document parsed using `parseString()` or `parseURL()`.

Parameters

None.

Example

This handler parses of an XML document and returns the resulting list:

```
-- Lingo syntax
on ConvertToList xmlString
    parserObject = new(xtra "xmlparser")
    errorCode = parserObject.parseString(xmlString)
    errorString = parserObject.getError()
    if voidP(errorString) then
        parsedList = parserObject.makeList()
    else
        alert "Sorry, there was an error" && errorString
        exit
    end if
    return parsedList
end

// JavaScript syntax
function ConvertToList(xmlString) {
    parserObject = new Xtra("xmlparser"); // check syntax
    errorCode = parserObject.parseString(xmlString);
    errorString = parserObject.getError();
```

```
    if (voidP(errorString)) {
        parsedList = parserObject.makeList();
    } else {
        alert("Sorry, there was an error" + errorString);
        return false;
    }
    return parsedList;
}
```

See also

`makeSubList()`

makeScriptedSprite()

Usage

```
-- Lingo syntax
spriteChannelObjRef.makeScriptedSprite({memberObjRef, loc})

// JavaScript syntax
spriteChannelObjRef.makeScriptedSprite({memberObjRef, loc});
```

Description

Sprite Channel method; switches control of a sprite channel from the Score to script, and optionally places a sprite from a specified cast member at a specified location on the Stage.

Call `removeScriptedSprite()` to switch control of the sprite channel back to the Score.

Parameters

`memberObjRef` Optional. A reference to the cast member from which a scripted sprite is created. Providing only this parameter places the sprite in the center of the Stage.

`loc` Optional. A point that specifies the location on the Stage at which the scripted sprite is placed.

Example

The following statement creates a scripted sprite in sprite channel 5 from the cast member named kite, and places it at a specific point on the Stage:

```
--- Lingo syntax
channel(5).makeScriptedSprite(member("kite"), point(35, 70))

// JavaScript syntax
channel(5).makeScriptedSprite(member("kite"), point(35, 70));
```

See also

`removeScriptedSprite()`, `Sprite Channel`

makeSubList()

Usage

```
XMLnode.makeSubList()
```

Description

Function; returns a property list from a child node the same way that `makeList()` returns the root of an XML document in list format.

Parameters

None.

Example

Beginning with the following XML:

```
<e1>
        <tagName attr1="val1" attr2="val2"/>
        <e2> element 2</e2>
        <e3>element 3</e3>
</e1>
```

This statement returns a property list made from the contents of the first child of the tag `<e1>`:

```
put gparser.child[ 1 ].child[ 1 ].makeSubList()
-- ["tagName": ["!ATTRIBUTES": ["attr1": "val1", "attr2": "val2"]]]
```

See also

`makeList()`

map()

Usage

```
map(targetRect, sourceRect, destinationRect)
map(targetPoint, sourceRect, destinationRect)
```

Description

Function; positions and sizes a rectangle or point based on the relationship of a source rectangle to a target rectangle.

The relationship of the *`targetRect`* to the *`sourceRect`* governs the relationship of the result of the function to the *`destinationRect`*.

Parameters

`targetRect` Required. The target rectangle in the relationship.

`targetPoint` Required. The target point in the relationship.

`sourceRect` Required. The source rectangle in the relationship.

`destinationRect` Required. The destination rectangle.

Example

In this behavior, all of the sprites have already been set to draggable. Sprite 2b contains a small bitmap. Sprite 1s is a rectangular shape sprite large enough to easily contain sprite 2b. Sprite 4b is a larger version of the bitmap in sprite 2b. Sprite 3s is a larger version of the shape in sprite 1s. Moving sprite 2b or sprite 1s will cause sprite 4b to move. When you drag sprite 2b, its movements are mirrored by sprite 4b. When you drag sprite 1s, sprite 4b moves in the opposite direction. Resizing sprite 2b or sprite 1s will also produce interesting results.

```
on exitFrame
```

```
    sprite(4b).rect = map(sprite(2b).rect, sprite(1s).rect, sprite(3s).rect)
    go the frame
end
```

lingo_map.dcr

map (3D)

Usage

```
member(whichCastmember).motion(whichMotion).map(whichOtherMotion {, boneName})
```

Description

3D motion command; maps a specified motion into the current motion, and applies it to a bone and all of the children of that bone. This command replaces any motion previously mapped to the specified bone and its children. This command does not change a model's playlist.

Parameters

whichOtherMotion Required. A string that specifies the name of the motion to map.

boneName Optional. A string that specifies the name of the bone to which the mapped motion is applied. If omitted, the root bone is used.

Example

The following statement maps the motion named LookUp into the motion named SitDown starting from the bone named Neck. The model will sit down and look up at the same time.

```
member("Restaurant").motion("SitDown").map("LookUp", "Neck")
```

See also

motion(), duration (3D), cloneMotionFromCastmember

mapMemberToStage()

Usage

```
sprite(whichSpriteNumber). mapMemberToStage(whichPointInMember)
mapMemberToStage(sprite whichSpriteNumber, whichPointInMember)
```

Description

Function; uses the specified sprite and point to return an equivalent point inside the dimensions of the Stage. This properly accounts for the current transformations to the sprite using `quad`, or the rectangle if not transformed.

This is useful for determining if a particular area of a cast member has been clicked, even if there have been major transformations to the sprite on the Stage.

If the specified point on the Stage is not within the sprite, a `VOID` is returned.

Parameters

whichPointInMember Required. A point from which an equivalent point is returned.

Example

This statement uses the specified sprite(1) and point(10,10) to return an equivalent point inside the dimensions of the Stage

```
-- Lingo
sprite(1). mapMemberToStage(point(10,10))

// Javascript
sprite(1). mapMemberToStage(point(10,10)) ;
```

See also

`map()`, `mapStageToMember()`

mapStageToMember()

Usage

```
sprite(whichSpriteNumber). mapStageToMember(whichPointOnStage)
mapStageToMember(sprite whichSpriteNumber, whichPointOnStage)
```

Description

Function; uses the specified sprite and point to return an equivalent point inside the dimensions of the cast member. This properly accounts for any current transformations to the sprite using `quad`, or the rectangle if not transformed.

This is useful for determining if a particular area on a cast member has been clicked even if there have been major transformations to the sprite on the Stage.

If the specified point on the Stage is not within the sprite, this function returns `VOID`.

Parameters

`whichPointOnStage` Required. A point from which an equivalent point is returned.

See also

`map()`, `mapMemberToStage()`

marker()

Usage

```
-- Lingo syntax
_movie.marker(markerNameOrNum)

// JavaScript syntax
_movie.marker(markerNameOrNum);
```

Description

Movie method; returns the frame number of markers before or after the current frame.

This method is useful for implementing a Next or Previous button or for setting up an animation loop.

If the parameter *`markerNameOrNum`* is an integer, it can evaluate to any positive or negative integer or 0. For example:

- `marker(2)`—Returns the frame number of the second marker after the current frame.

- `marker(1)`—Returns the frame number of the first marker after the current frame.
- `marker(0)`—Returns the frame number of the current frame if the current frame is marked, or the frame number of the previous marker if the current frame is not marked.
- `marker(-1)`—Returns the frame number of the first marker before the marker(0).
- `marker(-2)`—Returns the frame number of the second marker before the marker(0).

If the parameter *markerNameOrNum* is a string, `marker()` returns the frame number of the first frame whose marker label matches the string.

Parameters

markerNameOrNum Required. A string that specifies a marker label, or an integer that specifies a marker number.

Example

The following statement sends the playhead to the beginning of the current frame if the current frame has a marker; otherwise, it sends the playhead to the previous marker

```
-- Lingo syntax
_movie.go(_movie.marker(0))

// JavaScript syntax
_movie.go(_movie.marker(0));
```

This statement sets the variable `nextMarker` equal to the next marker in the Score:

```
-- Lingo syntax
nextMarker = _movie.marker(1)

// JavaScript syntax
nextMarker = _movie.marker(1);
```

See also

`frame`, `frameLabel`, `go()`, `label()`, `markerList`, `Movie`

matrixAddition()

Usage

```
<Matrix> matrixAddition(matrix1, matrix2)
```

Description

Global function; Performs matrix addition between the two matrices and returns the result as a matrix. matrix1 and matrix2 should be of same dimension.

Parameters

matrix1 Required. First matrix.

matrix2 Required. Second matrix.

Example

The following code snippet creates two matrices and adds them using matrixAddition().

```
--Lingo
mat1 = newMatrix(2, 2, [1,2,3,4])
mat2 = newMatrix(2, 2, [5,6,7,8])
```

```
mat3 = matrixAddition(mat1, mat2)
put(mat3.getVal(2,2))
-- 12.0000

//Java Script
mat1 = newMatrix(2, 2, list(1,2,3,4));
mat2 = newMatrix(2, 2, list(5,6,7,8));
mat3 = matrixAddition(mat1, mat2);
put(mat3.getVal(2,2));
// 12.0000
```

See also

getVal(), setVal(), numRows(), numColumns(), matrixMultiply(), matrixMultiplyScalar(), matrixTranspose(), newMatrix()

matrixMultiply()

Usage

```
<Matrix> matrixMultiply(matrix1, matrix2)
```

Description

Global function; Performs matrix multiplication between the two matrices and returns the result as a matrix. Number of columns of matrix1 should be same as the number of rows of matrix2.

Parameters

matrix1 Required. First matrix.

matrix2 Required. Second matrix.

Example

The following code snippet creates two matrices and adds them using matrixMultiply ().

```
--Lingo
mat1 = newMatrix(2, 2, [1,2,3,4])
mat2 = newMatrix(2, 2, [5,6,7,8])
mat3 = matrixMultiply(mat1, mat2)
put(mat3.getVal(2,2))

-- 50.0000


//Java Script
mat1 = newMatrix(2, 2, list(1,2,3,4));
mat2 = newMatrix(2, 2, list(5,6,7,8));
mat3 = matrixMultiply(mat1, mat2);
put(mat3.getVal(2,2));

// 50.0000
```

See also

getVal(), setVal(), numRows(), numColumns(), matrixAddition(), matrixMultiplyScalar(), matrixTranspose(), newMatrix()

matrixMultiplyScalar()

Usage

```
<Matrix> matrixMultiplyScalar(matrix, scalarMultiplier)
```

Description

Global function; Multiplies each element of the matrix with given scalar and returns the result as a matrix.

Parameters

matrix Required. Matrix involved in sacalar multiplication.

scalarMultiplier Required. Scalar value with which matrix is to be multiplied.

Example

Following code snippet creates two matrices and adds them using matrixMultiply ().

```
--Lingo
mat1 = newMatrix(2, 2, [1,2,3,4])
mat2 = matrixMultiplyScalar(mat1, 10)
put(mat2.getVal(2,2))

-- 40.0000

//Java Script
mat1 = newMatrix(2, 2, list(1,2,3,4));
mat2 = matrixMultiplyScalar(mat1, 10);
put(mat2.getVal(2,2));

// 40.0000
```

See also

getVal(), setVal(), numRows(), numColumns(), matrixAddition(), matrixMultiply(), matrixTranspose(), newMatrix()

matrixTranspose()

Usage

```
<Matrix> matrixTranspose(matrix)
```

Description

Global function; performs transpose operation on the given matrix and returns the result as another matrix.

Parameters

matrix Required. Matrix to be transposed.

Example

The following code snippet creates a matrix and transposes it using matrixTranspose()

```
--Lingo
mat1 = newMatrix(2, 2, [1,2,3,4])
mat2 = matrixTranspose(mat1)
put(mat1.getVal(1,2))
put(mat2.getVal(1,2))

-- 2.0000
```

```
-- 3.0000

//Java Script
mat1 = newMatrix(2, 2, list(1,2,3,4));
mat2 = matrixTranspose(mat1);
put(mat1.getVal(1,2));
put(mat2.getVal(1,2));

// 2.0000
// 3.0000
```

See also

getVal(), setVal(), numRows(), numColumns(), matrixAddition(), matrixMultiply(), matrixMultiplyScalar(), newMatrix()

max()

Usage

```
list.max()
max(list)
max(value1, value2, value3, ...)
```

Description

Function (Lingo only); returns the highest value in the specified list or the highest of a given series of values.

The `max` function also works with ASCII characters, similar to the way < and > operators work with strings.

Parameters

value1, value2, value3, ... Optional. A list of values from which the highest value is chosen.

Example

The following handler assigns the variable `Winner` the maximum value in the list `Bids`, which consists of `[#Castle:600, #Schmitz:750, #Wang:230]`. The result is then inserted into the content of the field cast member Congratulations.

```
-- Lingo syntax
on findWinner Bids
    Winner = Bids.max()
    member("Congratulations").text =
    "You have won, with a bid of $" & Winner &"!"
end

// JavaScript syntax
function findWinner(Bids) {
    Winner = Bids.max();
    member("Congratulations").text = "You have won, with a bid of $" +
    Winner + "!");
}
```

maximize()

Usage

```
-- Lingo syntax
```

```
windowObjRef.maximize()

// JavaScript syntax
windowObjRef.maximize();
```

Description

Window method; maximizes a window.

Use this method when making custom titlebars.

Parameters

None.

Example

These statements maximize the window named Artists if it is not already maximized.

```
-- Lingo syntax
if (window("Artists").sizeState <> #maximized) then
    window("Artists").maximize()
end if

// JavaScript syntax
if (window("Artists").sizeState != symbol("maximized")) {
    window("Artists").maximize();
}
```

See also

minimize(), Window

mci

Usage

```
mci "string"
```

Description

Command; for Windows only, passes the strings specified by *string* to the Windows Media Control Interface (MCI) for control of multimedia extensions.

Note: *Microsoft no longer recommends using the 16-bit MCI interface. Consider using third-party Xtra extensions for this functionality instead.*

Parameters

string Required. A string that is passed to the MCI.

Example

The following statement makes the command `play cdaudio from 200 to 600 track 7` play only when the movie plays back in Windows:

```
mci "play cdaudio from 200 to 600 track 7"
```

member()

Usage

```
-- Lingo syntax
member(memberNameOrNum {, castNameOrNum})

// JavaScript syntax
member(memberNameOrNum {, castNameOrNum});
```

Description

Top level function; creates a reference to a cast member, and optionally specifies the cast library that contains the member.

The `member()` method is a specific reference to both a cast library and a member within it if used with both the *`memberNameOrNum`* and *`castNameOrNum`* parameters:

```
trace(sprite(1).member);
// (member 1 of castLib 1)
```

This method differs from the `spriteNum` property of a sprite, which is always an integer designating position in a cast library, but does not specify the cast library:

```
trace(sprite(2).spriteNum);
// 2
```

The number of a member is also an absolute reference to a particular member in a particular cast library:

```
trace(sprite(3).member.number)
// 3
```

Parameters

`memberNameOrNum` Required. A string that specifies the name of the cast member to reference, or an integer that specifies the index position of the cast member to reference.

`castNameOrNum` Optional. A string that specifies the cast library name to which the member belongs, or an integer that specifies the index position of the cast library to which the member belongs. If omitted, `member()` searches all cast libraries until a match is found.

Example

This statements sets the variable `memWings` to the cast member named Planes, which is in the cast library named Transportation.

```
-- Lingo syntax
memWings = member("Planes", "Transportation")

// JavaScript syntax
var memWings = member("Planes", "Transportation");
```

See also

Member, Sprite, spriteNum

mergeDisplayTemplate()

Usage

```
-- Lingo syntax
_movie.mergeDisplayTemplate(propList)

// JavaScript syntax
_movie.mergeDisplayTemplate(propList);
```

Description

Movie method; merges an arbitrary number of display template properties into the existing set of display template properties all at once.

Parameters

propList Required. A property list that contains the display template properties to merge into the existing set of display template properties. In Lingo, *propList* can be either a comma-separated list of name/value pairs or a comma-separated list of symbol/value pairs. In JavaScript syntax, *propList* can only be a comma-separated list of name/value pairs.

Example

This statement merges a value for the `title property` into the `displayTemplate`:

```
-- Lingo syntax
_movie.mergeDisplayTemplate(propList(#title, "Welcome!"))

// JavaScript syntax
_movie.mergeDisplayTemplate(propList("title", "Welcome!"))
```

See also

appearanceOptions, displayTemplate, Movie, propList(), titlebarOptions

mergeProps()

Usage

```
-- Lingo syntax
windowObjRef.mergeProps(propList)

// JavaScript syntax
windowObjRef.mergeProps(propList);
```

Description

Windows method. Merges an arbitrary number of window properties, all at once, into the existing set of window properties.

Parameters

propList Required. A set of window properties to merge into the existing set of window properties. The properties are specified by the `appearanceOptions` and `titlebarOptions` properties.

- In Lingo, *propList* can be either a comma-separated list of name/value pairs or a comma-separated list of symbol/value pairs.
- In JavaScript syntax, *propList* can only be a comma-separated list of name/value pairs.

Example

This statement sets various properties for the window named Cars.

```
-- Lingo syntax
window("Cars").mergeProps([#title:"Car pictures", #resizable:FALSE,
#titlebarOptions:[#closebox:TRUE, #icon:member(2)], #appearanceOptions:[#border:#line,
#shadow:TRUE]])

// JavaScript syntax
window("Cars").mergeProps(propList("title","Car pictures",
"resizable",false,"titlebarOptions",propList("closebox",true, "icon",member(2)),
"appearanceOptions",propList("border","line", "shadow",true)));
```

See also

appearanceOptions, titlebarOptions, Window

mesh (property)

Usage

```
member(whichCastmember).model(whichModel).meshdeform.mesh[index].meshProperty
```

Description

3D command; allows access to the mesh properties of models that have the `meshDeform` modifier attached. When used as `mesh.count` this command returns the total number of meshes within the referenced model.

The properties of each mesh that are accessible are as follows:

- `colorList` allows you to get or set the list of colors used by the specified mesh.
- `vertexList` allows you to get or set the list of vertices used by the specified mesh.
- `normalList` allows you to get or set the list of normal vectors used by the specified mesh.
- `textureCoordinateList` allows you to get or set the texture coordinates used by the first texture layer of the specified mesh. To get or set the texture coordinates for any other texture layers in the specified mesh, use `meshdeform.mesh[`*index*`].texturelayer[`*index*`].textureCoordinateList`.
- `textureLayer[index]` allows you get and set access to the properties of the specified texture layer.
- `face[index]` allows you to get or set the vertices, normals, texture coordinates, colors, and shaders used by the faces of the specified mesh.
- `face.count` allows you to obtain the total number of faces found within the specified mesh.

Note: *For complete information about these properties, see the individual entries (listed in the "See also" section of this entry).*

Parameters

None.

Example

The following Lingo adds the `#meshDeform` modifier to the model named thing1 and then displays the vertexList for the first mesh in the model named thing1.

```
-- Lingo
member("newAlien").model("thing1").addModifier(#meshDeform)
put member("newalien").model("thing1").meshDeform.mesh[1].vertexList

// javascript
member("newAlien").getProp("model",1).addModifier(symbol("meshDeform"));
put
(member("newalien").getProp("model",1).getPropRef("meshDeform").getPropRef("mesh",1).verte
xList ;
-- [vector(239.0, -1000.5, 27.4), vector(162.5, -1064.7, 29.3), vector(115.3, -1010.8, -
40.6),

vector(239.0, -1000.5, 27.4), vector(115.3, -1010.8, -40.6),
vector(162.5, -1064.7, 29.3), vector(359.0, -828.5, -46.3),
vector(309.9, -914.5, -45.3)]
```

The following statement displays the number of meshes found within the model named "Aircraft".

```
-- Lingo
put member("world").model("Aircraft").meshDeform.mesh.count
-- 4
```

See also

meshDeform (modifier), colorList, textureCoordinateList, textureLayer, normalList, vertexList (mesh deform), face[ ]

meshDeform (modifier)

Usage

```
member(whichCastmember).model(whichModel).meshDeform.propertyName
```

Description

3D modifier; allows control over the various aspects of the referenced model's mesh structure. Once you have added the `#meshDeform` modifier (using the `addModifier` command) to a model you have access to the following properties of the `#meshDeform` modifier:

Note: *For more detailed information about the following properties see the individual property entries referenced in the see also section of this entry.*

- `face.count` returns the total number of faces in the referenced model.
- `mesh.count` returns the number of meshes in the referenced model.
- `mesh[index]` allows access to the properties of the specified mesh.

Parameters

None.

Example

The following statement displays the number of faces in the model named gbFace:

```
-- Lingo
put member("3D World").model("gbFace").meshDeform.face.count
-- 432
```

```
// Javascript
member("3D World").getProp("model", a).getPropRef("meshDeform") ;
// where a refers to the number index of the model "gbFace"
```

The following statement displays the vertexList of meshes in the model named gbFace:

```
-- Lingo
put member("3D World").model("gbFace").meshDeform.mesh[1].vertexList
-- 2

// Javscript
member("3D World").getProp("model", a).getPropRef("meshDeform").getPropRef("mesh"
,1).vertexList ;
```

The following statement displays the number of faces in the second mesh of the model named gbFace:

```
put member("3D World").model("gbFace").meshDeform.mesh[2].face.count
-- 204
```

See also

mesh (property), addModifier

min

Usage

```
list.min
min(list)
min (a1, a2, a3...)
```

Description

Function (Lingo only); specifies the minimum value in a list.

Parameters

a1, a2, a3, ... Optional. A list of values from which the lowest value is chosen.

Example

The following handler assigns the variable `vLowest` the minimum value in the list `bids`, which consists of `[#Castle:600, #Shields:750, #Wang:230]`. The result is then inserted in the content of the field cast member Sorry:

```
on findLowest bids
    vLowest = bids.min()
    member("Sorry").text = \
    "We're sorry, your bid of $" & vLowest && "is not a winner!"
end
```

See also

max()

minimize()

Usage

```
-- Lingo syntax
windowObjRef.minimize()

// JavaScript syntax
windowObjRef.minimize();
```

Description

Window method; minimizes a window.

Use this method when making custom titlebars.

Parameters

None.

Example

These statements minimize the window named Artists if it is not already minimized.

```
-- Lingo syntax
if (window("Artists").sizeState <> #minimized) then
    window("Artists").minimize()
end if

// JavaScript syntax
if (window("Artists").sizeState.toString() != symbol("minimized").toString())
{
    window("Artists").minimized();
}
```

See also

`maximize()`, `Window`

model

Usage

```
member(whichCastmember).model(whichModel)
member(whichCastmember).model[index]
member(whichCastmember).model.count
member(whichCastmember).model(whichModel).propertyName
member(whichCastmember).model[index].propertyName
```

Description

3D command; returns the model found within the referenced cast member that has the name specified by *`whichModel`*, or is found at the index position specified by *`index`*. If no model exists for the specified parameter, the command returns `void`. As `model.count`, the command returns the number of models found within the referenced cast member. This command also allows access to the specified model's properties.

Model name comparisons are not case-sensitive. The index position of a particular model may change when objects at lower index positions are deleted.

If no model is found that uses the specified name or no model is found at the specified index position then this command returns void.

Parameters

whichModel Optional. A string that specifies the name of the model to return.

Example

This statement stores a reference to the model named Player Avatar in the variable `thismodel`:

```
thismodel = member("3DWorld").model("Player Avatar")
```

This statement stores a reference to the eighth model of the cast member named 3DWorld in the variable `thismodel`.

```
thismodel = member("3DWorld").model[8]
```

This statement shows that there are four models in the member of sprite 1.

```
put sprite(1).member.model.count
-- 4
```

modelResource

Usage

```
member(whichCastmember).modelResource(whichModelResource)
member(whichCastmember).modelResource[index]
member(whichCastmember).modelResource.count
member(whichCastmember).modelResource(whichModelResource).propertyName
member(whichCastmember).modelResource[index].propertyName
```

Description

3D command; returns the model resource found within the referenced cast member that has the name specified by *whichModelResource*, or is found at the index position specified by the *index* parameter. If no model resource exists for the specified parameter, the command returns `void`. As `modelResource.count`, the command returns the number of model resources found within the referenced cast member. This command also allows access to the specified model resource's properties.

Model resource name string comparisons are not case-sensitive. The index position of a particular model resource may change when objects at lower index positions are deleted.

Parameters

whichModelResource Optional. A string that specifies the name of the model resource to return.

Example

This statement stores a reference to the model resource named HouseA in the variable `thismodelResource`.

```
thismodelResource = member("3DWorld").modelResource("HouseA")
```

This statement stores a reference to the fourteenth model resource of the cast member named 3DWorld in the variable `thismodelResource`.

```
thismodelResource = member("3DWorld").modelResource[14]
```

This statement shows that there are ten model resources in the member of sprite 1.

```
put sprite(1).member.modelResource.count
--10
```

modelsUnderLoc

Usage

```
member(whichCastmember).camera(whichCamera).modelsUnderLoc(pointWithinSprite, optionsList)
```

Description

3D command; returns a list of models found under a specified point within the rect of a sprite using the referenced camera. The list of models can also be compared to a set of optional parameters before being returned.

Within the returned list, the first model listed is the one closest to the viewer and the last model listed is the furthest from the viewer.

Only one intersection (the closest intersection) is returned per model.

The command returns an empty list if there are no models found under the specified point.

Parameters

pointWithinSprite Required. A point under which a list of models is found. This point is relative to the upper left corner of the sprite, in pixels.

optionsList Optional. A list that specifies the maximum number of models to return, the level of information detail, a list of models among which to cast, and the maximum distance to draw the ray. All of these properties are optional.

maxNumberOfModels Optional. An integer that specifies the maximum length of the returned list. If omitted, the command returns a list containing references for all of the models found under the specified point.

levelOfDetail Optional. A symbol that specifies the level of detail of the information returned. Valid values include the following:

- `#simple` returns a list containing references to the models found under the point. This is the default setting.
- `#detailed` returns a list of property lists, each representing an intersected model. Each property list will have the following properties:
    - `#model` is a reference to the intersected model object.
    - `#distance` is the distance from the camera to the point of intersection with the model.
    - `#isectPosition` is a vector representing the world space position of the point of intersection.
    - `#isectNormal` is the world space normal vector to the mesh at the point of intersection.
    - `#meshID` is the meshID of the intersected mesh, which can be used as in index into the mesh list of the `meshDeform` modifier.
    - `#faceID` is the face ID of the intersected face, which can be used as an index into the face list of the `meshDeform` modifier.
    - `#vertices` is a three-element list of vectors that represent the world space positions of the vertices of the intersected face.
    - `#uvCoord` is a property list with properties `#u` and `#v` that represent the u and v barycentric coordinates of the face.

modelList Optional. A list of model references that are included if they are found under the specified ray. Model references not included in this list are ignored, even if they are under the specified ray. Use the model references, not the string names of the models. Specify each model you want to include. Adding a parent model reference does not automatically include its child model references.

Example

This statement creates a list of ten models:

```
tModelList = [member("3D").model("foo"),member("3D").model[10]]
```

This statement builds a list of options that would return a maximum of ten models, include simple detail, and draw results from `tModelList`:

```
tOptionsList = [#maxNumberOfModels: 10, #levelOfDetail: #simple, #modelList: tModelList]
```

After the options list is built, the first line in this handler translates the location of the cursor from a point on the Stage to a point within sprite 5. The second line uses the `modelsUnderLoc` command to obtain the first three models found under that point. The third line displays the returned information about the models in the message window.

```
-- Lingo syntax
on mouseUp
    pt = the mouseLoc - point(sprite(5).left, sprite(5).top)
    m = sprite(5).camera.modelsUnderLoc(pt, tOptionsList)
    put m
    end

// JavaScript syntax
function mouseUp() {
    pt = _mouse.mouseLoc - point(sprite(5).left, sprite(5).top);
    m = sprite(5).camera.modelsUnderLoc(pt, tOptionsList);
    put(m);
}
```

See also

modelsUnderRay, modelUnderLoc

modelsUnderRay

Usage

```
member(whichCastmember).modelsUnderRay(locationVector, directionVector, optionsList)
```

Description

3D command; returns a list of models found under a ray drawn from a specified position and pointing in a specified direction, with both vectors being specified in world-relative coordinates. The list of models can also be compared to a set of optional parameters before being returned.

Within the returned list, the first model listed is the one closest to the position specified by *locationVector* and the last model listed is the furthest from that position.

Only one intersection (the closest intersection) is returned per model.

The command returns an empty list if there are no models found under the specified ray.

Parameters

locationVector Required. A vector from which a ray is drawn and under which a list of models is found.

directionVector Required. A vector that specifies the direction the ray is pointing.

optionsList Optional. A list that specifies the maximum number of models to return, the level of information detail, a list of models among which to cast, and the maximum distance to draw the ray. All of these properties are optional.

maxNumberOfModels Optional. An integer that specifies the maximum length of the returned list. If omitted, the command returns a list containing references for all of the models found under the specified ray.

levelOfDetail Optional. A symbol that specifies the level of detail of the information returned. Valid values include the following:

- `#simple` returns a list containing references to the models found under the point. This is the default setting.
- `#detailed` returns a list of property lists, each representing an intersected model. Each property list will have the following properties:
    - `#model` is a reference to the intersected model object.
    - `#distance` is the distance from the world position specified by *locationVector* to the point of intersection with the model.
    - `#isectPosition` is a vector representing the world space position of the point of intersection.
    - `#isectNormal` is the world space normal vector to the mesh at the point of intersection.
    - `#meshID` is the meshID of the intersected mesh which can be used to index into the mesh list of the `meshDeform` modifier.
    - `#faceID` is the face ID of the intersected face which can be used to index into the face list of the `meshDeform` modifier.
    - `#vertices` is a 3-element list of vectors that represent the world space positions of the vertices of the intersected face.
    - `#uvCoord` is a property list with properties `#u` and `#v` that represent the u and v barycentric coordinates of the face.

modelList Optional. A list of model references that are included if they are found under the specified ray. Model references not included in this list are ignored, even if they are under the specified ray. Use the model references, not the string names of the models. Specify each model you want to include. Adding a parent model reference does not automatically include its child model references.

maxDistance Optional. The maximum distance from the world position specified by *locationVector*. If a model's bounding sphere is within the maximum distance specified, that model is included. If the bounding sphere is in range, then it may contain polygons in range and thus might be intersected.

Example

This statement creates a list of ten models:

```
tModelList = [member("3D").model("foo"),member("3D").model[10]]
```

This statement builds a list of options that would return a maximum of ten models, include simple detail, draw results from `tModelList`, and have a maximum ray distance of 50:

```
tOptionsList = [#maxNumberOfModels: 10, #levelOfDetail: #simple, #modelList: tModelList,
#maxDistance: 50]
```

After the option list is built, this statement includes it under a ray drawn from the position vector (0, 0, 300) and pointing down the -z axis:

```
put member("3d").modelsUnderRay(vector(0, 0, 300), vector(0, 0, -1), tOptionsList)
```

See also

modelsUnderLoc, modelUnderLoc

modelUnderLoc

Usage

```
member(whichCastmember).camera(whichCamera).modelUnderLoc(pointWithinSprite)
```

Description

3D command; returns a reference to the first model found under a specified point within the rect of a sprite using the referenced camera.

This command returns `void` if there is no model found under the specified point.

For a list of all of the models found under a specified point, and detailed information about them, see modelsUnderLoc.

Parameters

pointWithinSprite Required. A point under which the first model is found. The location of *pointWithinSprite* is relative to the upper left corner of the sprite, in pixels.

Example

The first line in this handler translates the location of the cursor from a point on the Stage to a point within sprite 5. The second line determines the first model under that point. The third line displays the result in the message window.

```
-- Lingo syntax
on mouseUp
    pt = the mouseLoc - point(sprite(5).left, sprite(5).top)
    m = sprite(5).camera.modelUnderLoc(pt)
    put m
end

// JavaScript syntax
function mouseUp() {
    pt = _mouse.mouseLoc - point(sprite(5).left, sprite(5).top);
    m = sprite(5).camera.modelUnderLoc(pt);
    put(m);
}
```

See also

modelsUnderLoc, modelsUnderRay

motion()

Usage

```
member(whichCastmember).motion(whichMotion)
member(whichCastmember).motion[index]
member(whichCastmember).motion.count
```

Description

3D command; returns the motion found within the referenced cast member that has the name specified by *whichMotion*, or is found at the index position specified by the *index*. As `motion.count`, this property returns the total number of motions found within the cast member.

Object name string comparisons are not case-sensitive. The index position of a particular motion may change when objects at lower index positions are deleted.

If no motion is found that uses the specified name or no motion is found at the specified index position then this command returns void.

Example

```
thisMotion = member("3D World").motion("Wing Flap")
thisMotion = member("3D World").motion[7]
put member("scene").motion.count
-- 2
```

See also

duration (3D), map (3D)

move()

Usage

```
-- Lingo syntax
memberObjRef.move({intPosn, castLibName})

// JavaScript syntax
memberObjRef.move({intPosn, castLibName});
```

Description

Member method; moves a specified cast member to either the first empty location in its containing cast, or to a specified location in a given cast.

For best results, use this method during authoring, not at runtime, because the move is typically saved with the file. The actual location of a cast member does not affect most presentations during playback for an end user. To switch the content of a sprite or change the display during runtime, set the member of the sprite.

Parameters

intPosn Optional. An integer that specifies the position in the cast library *castLibName* to which the member is moved.

castLibName Optional. A string that specifies the name of the cast library to which the member is moved.

Example

This statement moves cast member Shrine to the first empty location in the Cast window:

```
-- Lingo syntax
member("shrine").move()

// JavaScript syntax
member("shrine").move();
```

This statement moves cast member Shrine to location 20 in the Bitmaps Cast window:

```
-- Lingo syntax
member("shrine").move(member(20, "Bitmaps"))

// JavaScript syntax
member("shrine").move(member(20, "Bitmaps"));
```

See also

Member

moveToBack()

Usage

```
-- Lingo syntax
windowObjRef.moveToBack()

// JavaScript syntax
windowObjRef.moveToBack();
```

Description

Window method; moves a window behind all other windows.

Parameters

None.

Example

These statements move the first window in windowList behind all other windows:

```
-- Lingo syntax
myWindow = _player.windowList[1]
myWindow.moveToBack()

// JavaScript syntax
var myWindow = _player.windowList[1];
myWindow.moveToBack();
```

If you know the name of the window you want to move, use the syntax:

```
-- Lingo syntax
window("Demo Window").moveToBack()

// JavaScript syntax
window("Demo Window").moveToBack();
```

See also

moveToFront(), Window

moveToFront()

Usage

```
-- Lingo syntax
windowObjRef.moveToFront()

// JavaScript syntax
```

```
windowObjRef.moveToFront();
```

Description

Window method; moves a window in front of all other windows.

Parameters

None.

Example

These statements move the first window in `windowList` in front of all other windows:

```
-- Lingo syntax
myWindow = _player.windowList[1]
myWindow.moveToFront()

// JavaScript syntax
var myWindow = _player.windowList[1];
myWindow.moveToFront();
```

If you know the name of the window you want to move, use the syntax:

```
-- Lingo syntax
window("Demo Window").moveToFront()

// JavaScript syntax
window("Demo Window").moveToFront();
```

See also

`moveToBack()`, `Window`

moveVertex()

Usage

```
-- Lingo syntax
memberObjRef.moveVertex(vertexIndex, xChange, yChange)

// JavaScript syntax
memberObjRef.moveVertex(vertexIndex, xChange, yChange);
```

Description

Function; moves the vertex of a vector shape cast member to another location.

The horizontal and vertical coordinates for the move are relative to the current position of the vertex point. The location of the vertex point is relative to the origin of the vector shape member.

Changing the location of a vertex affects the shape in the same way as dragging the vertex in an editor.

Parameters

`vertexIndex` Required. Specifies the index position of the vertex to move.

`xChange` Required. Specifies the amount to move the vertex horizontally.

`yChange` Required. Specifies the amount to move the vertex vertically.

Example

This statement shifts the first vertex point in the vector shape Archie 25 pixels to the right and 10 pixels down from its current position:

```
-- Lingo syntax
member("Archie").moveVertex(1, 25, 10)

// JavaScript syntax
member("Archie").moveVertex(1, 25, 10);
```

See also

addVertex(), deleteVertex(), moveVertexHandle(), originMode, vertexList

moveVertexHandle()

Usage

```
-- Lingo syntax
memberObjRef.moveVertexHandle(vertexIndex, handleIndex, xChange, yChange)

// JavaScript syntax
memberObjRef.moveVertexHandle(vertexIndex, handleIndex, xChange, yChange);
```

Description

Function; moves the vertex handle of a vector shape cast member to another location.

The horizontal and vertical coordinates for the move are relative to the current position of the vertex handle. The location of the vertex handle is relative to the vertex point it controls.

Changing the location of a control handle affects the shape in the same way as dragging the vertex in the editor.

Parameters

vertexIndex Required. Specifies the index position of the vertex that contains the handle to move.

handleIndex Required. Specifies the index position of the handle to move.

xChange Required. Specifies the amount to move the vertex handle horizontally.

yChange Required. Specifies the amount to move the vertex handle vertically.

Example

This statement shifts the first control handle of the second vertex point in the vector shape Archie 15 pixels to the right and 5 pixels up:

```
-- Lingo syntax
moveVertexHandle(member("Archie"), 2, 1, 15, -5)

// JavaScript syntax
member("Archie").moveVertexHandle(2, 1, 15, -5);
```

See also

addVertex(), deleteVertex(), originMode, vertexList

multiply()

Usage

```
transform.multiply(transform2)
```

Description

3D command; applies the positional, rotational, and scaling effects of *transform2* after the original transform.

Parameters

transform2 Required. Specifies the transform that contains the effects to apply to another transform.

Example

This statement applies the positional, rotational, and scaling effects of the model Mars's transform to the transform of the model Pluto. This has a similar effect as making Mars be Pluto's parent for a frame.

```
-- Lingo
member("scene").model("Pluto").transform.multiply(member("scene").model("Mars").transform)

// Javascript
member("scene").getProp("model" ,i ).transform.multiply(member("scene").getProp("model" ,
j).transform) ;
```

neighbor

Usage

```
member(whichCastmember).model(whichModel).meshdeform.mesh[index].face[index].neighbor[inde
x]
```

Description

3D command; meshDeform command that returns a list of lists describing the neighbors of a particular face of a mesh opposite the face corner specified by the neighbor index `(1,2,3)`. If the list is empty, the face has no neighbors in that direction. If the list contains more than one list, the mesh is non-manifold. Usually the list contains a single list of four integer values: `[meshIndex, faceIndex, vertexIndex, flipped]`.

The value `meshIndex` is the index of the mesh containing the neighbor face. The value `faceIndex` is the index of the neighbor face in that mesh. The value `vertexIndex` is the index of the nonshared vertices of the neighbor face. The value `flipped` describes whether the face orientation is the same as `(1)` or opposite `(2)` that of the original face.

Parameters

None.

See also

`meshDeform (modifier)`

netAbort

Usage

```
netAbort(URL)
netAbort(netID)
```

Description

Command; cancels a network operation without waiting for a result.

Using a network ID is the most efficient way to stop a network operation. The ID is returned when you use a network function such as `getNetText()` or `postNetText()`.

In some cases, when a network ID is not available, you can use a URL to stop the transmission of data for that URL. The URL must be identical to that used to begin the network operation. If the data transmission is complete, this command has no effect.

Parameters

`URL` Required. Specifies the URL to cancel.

`netID` Optional. Specifies the ID of the network operation to cancel.

Example

This statement passes a network ID to netAbort to cancel a particular network operation:

```
-- Lingo syntax
on mouseUp
    netAbort(myNetID)
end

// JavaScript syntax
function mouseUp() {
    netAbort(myNetID);
}
```

See also

`getNetText()`, `postNetText`

netDone()

Usage

```
netDone()
netDone(netID)
```

Description

Function; indicates whether a background loading operation (such as `getNetText, preloadNetThing, gotoNetMovie`, `gotoNetPage`, or `netTextResult`) is finished or was terminated by a browser error (`TRUE`, default) or is still in progress (`FALSE`).

- Use `netDone()` to test the last network operation.
- Use `netDone(`*`netID`*`)` to test the network operation identified by `netID`.

The `netDone` function returns 0 when a background loading operation is in progress.

Parameters

`netID` Optional. Specifies the ID of the network operation to test.

Example

The following handler uses the netDone function to test whether the last network operation has finished. If the operation is finished, text returned by `netTextResult` is displayed in the field cast member Display Text.

```
-- Lingo syntax
on exitFrame
    if netDone() = 1 then
        member("Display Text").text = netTextResult()
    end if
end

// JavaScript syntax
function exitFrame() {
    if (netDone() == 1) {
        member("Display Text").text = netTextResult();
    }
}
```

This handler uses a specific network ID as an argument for netDone to check the status of a specific network operation:

```
-- Lingo syntax
on exitFrame
    -- stay on this frame until the net operation is
    -- completed
    global mynetID
    if netDone(mynetID) = FALSE then
        go to the frame
    end if
end

// JavaScript syntax
function exitFrame() {
    // stay on this frame until the net operation is completed
    global mynetID;
    if (!(netDone(mynetID))) {
        _movie.go(_movie.frame);
    }
}
```

See also

`getNetText()`, `netTextResult()`, `gotoNetMovie`, `preloadNetThing()`

netError()

Usage

```
netError()
netError(netID)
```

Description

Function; determines whether an error has occurred in a network operation and, if so, returns an error number corresponding to an error message. If the operation was successful, this function returns a code indicating that everything is okay. If no background loading operation has started, or if the operation is in progress, this function returns an empty string.

- Use `netError()` to test the last network operation.
- Use `netError(netID)` to test the network operation specified by `netID`.

Several possible error codes may be returned:

0	Everything is okay.
1	Occurs when the local path points to a directory which doesn't exist
4	Bad MOA class. The required network or nonnetwork Xtra extensions are improperly installed or not installed at all.
5	Bad MOA Interface. See 4.
6	Bad URL or Bad MOA class. The required network or nonnetwork Xtra extensions are improperly installed or not installed at all.
20	Internal error. Returned by `netError()` in the Netscape browser if the browser detected a network or internal error.
900	File to be written to is read-only
903	Disk is full.
905	Bad filespec
2018	postNetText error usually happens when you use postNetText with the parameters in the URL. M.Kloss recommends this syntax to avoid 2018: pNetID = postNetText(myURL, myParamPropList)
4144	Failed network operation
4145	Failed network operation
4146	Connection could not be established with the remote host.
4149	Data supplied by the server was in an unexpected format.
4150	Unexpected early closing of connection.
4154	Operation could not be completed due to timeout.
4155	Not enough memory available to complete the transaction.
4156	Protocol reply to request indicates an error in the reply.
4157	Transaction failed to be authenticated.
4159	Invalid URL.
4164	Could not create a socket.
4165	Requested object could not be found (URL may be incorrect).
4166	Generic proxy failure.
4167	Transfer was intentionally interrupted by client.
4242	Download stopped by `netAbort(url)`.
4836	Download stopped for an unknown reason, possibly a network error, or the download was abandoned.
4153	Failed network operation
4154	Operation could not be completed due to timeout
4155	Not enough memory available to complete the transaction
4157	Transaction failed to be authenticated
4159	Invalid URL

4160	Failed network operation
4161	Failed network operation
4162	Failed network operation
4163	Failed network operation
4164	Could not create a socket
4165	Requested Object could not be found (URL may be incorrect)
4166	Generic proxy failure
4167	Transfer was intentionally interrupted by client
4168	Failed network operation
4240	The network xtras weren't initialized properly. (Lewis Francis)
4242	Download stopped by netAbort(url)
4836	Download stopped for an unknown reason. May have been a network error, or the download may have been abandoned.

Parameters

netID Optional. Specifies the ID of the network operation to test.

Example

This statement passes a network ID to `netError` to check the error status of a particular network operation:

```
--Lingo syntax
on exitFrame
    global mynetID
    if netError(mynetID)<>"OK" then beep
end

// JavaScript syntax
function exitFrame() {
    global mynetID;
    if (netError(mynetID) != "OK") {
        _sound.beep();
    }
}
```

netLastModDate()

Usage

```
netLastModDate()
```

Description

Function; returns the date last modified from the HTTP header for the specified item. The string is in Universal Time (GMT) format: *Ddd, nn Mmm yyyy hh:mm:ss* `GMT` (for example, Thu, 30 Jan 1997 12:00:00 AM GMT). There are variations where days or months are spelled completely. The string is always in English.

The `netLastModDate` function can be called only after `netDone` and `netError` report that the operation is complete and successful. After the next operation starts, the Director movie or projector discards the results of the previous operation to conserve memory.

The actual date string is pulled directly from the HTTP header in the form provided by the server. However, this string is not always provided, and in that case `netLastModDate` returns `EMPTY`.

Parameters

None.

Example

These statements check the date of a file downloaded from the Internet:

```
-- Lingo syntax
if netDone() then
    theDate = netLastModDate()
    if theDate.char[6..11] <> "Jan 30" then
        alert "The file is outdated."
    end if
end if

// JavaScript syntax
if (netDone()) {
    theDate = netLastModDate();
    if (theDate.char[6..11] != "Jan 30") {
        alert("The file is outdated");
    }
}
```

See also

`netDone()`, `netError()`

netMIME()

Usage

```
netMIME()
```

Description

Function; provides the MIME type of the Internet file that the last network operation returned (the most recently downloaded HTTP or FTP item).

The `netMIME` function can be called only after `netDone` and `netError` report that the operation is complete and successful. After the next operation starts, the Director movie or projector discards the results of the previous operation to conserve memory.

Parameters

None.

Example

This handler checks the MIME type of an item downloaded from the Internet and responds accordingly:

```
-- Lingo syntax
on checkNetOperation theURL
    if netDone (theURL) then
        set myMimeType = netMIME()
        case myMimeType of
            "image/jpeg": go frame "jpeg info"
            "image/gif": go frame "gif info"
```

```
            "application/x-director": goToNetMovie theURL
            "text/html": gotoNetPage theURL
            otherwise: alert "Please choose a different item."
        end case
    else
        go the frame
    end if
end

// JavaScript syntax
function checkNetOperation(theURL) {
    if (netDone(theURL)) {
        myMimeType = netMIME();
        switch (myMimeType) {
            case "image/jpeg":
                _movie.go("jpeg info");
                break;
            case "image/gif":
                _movie.go("gif info");
                break;
            case "application/x-director":
                goToNetMovie(theURL);
                break;
            case "text/html":
                gotoNetPage(theURL);
                break;
            default:
                alert("Please choose a different item.");
        }
    } else {
        _movie.go(_movie.frame);
    }
}
```

See also

netDone(), netError(), getNetText(), postNetText, preloadNetThing()

netStatus

Usage

```
netStatus msgString
```

Description

Command; displays the specified string in the status area of the browser window.

The `netStatus` command doesn't work in projectors.

Parameters

`msgString` Required. Specifies the string to display.

Example

This statement would place the string "This is a test" in the status area of the browser the movie is running in:

```
-- Lingo syntax
on exitFrame
    netStatus "This is a test"
```

```
end

// JavaScript syntax
function exitFrame() {
    _movie.netStatus("This is a test");
}
```

netTextResult()

Usage

```
netTextResult(netID)
netTextResult()
```

Description

Function; returns the text obtained by the specified network operation. If no net ID is specified, `netTextResult` returns the result of the last network operation.

If the specified network operation was `getNetText()`, the text is the text of the file on the network.

If the specified network operation was `postNetText`, the result is the server's response.

After the next operation starts, Director discards the results of the previous operation to conserve memory.

Director retains results for the last six network operations in all playback environments. After netTextResult() is used to retrieve the data, the results are discarded for that network request.

Parameters

`netID` Optional. Specifies the ID of the network operation that contains the text to return.

Example

This handler uses the "`netDone` and `netError`" functions to test whether the last network operation finished successfully. If the operation is finished, text returned by `netTextResult` is displayed in the field cast member Display Text.

```
-- Lingo syntax
global gNetID
on exitFrame
    if (netDone(gNetID) = TRUE) and (netError(gNetID) = "OK") then
        member("Display Text").text = netTextResult()
    end if
end

// JavaScript syntax
global gNetID;
function exitFrame() {
    if (netDone(gNetID) && (netError(gNetID) == "OK")) {
        member("Display Text").text = netTextResult();
    }
}
```

See also

`netDone()`, `netError()`, `postNetText`

new()

Usage

```
new(type)
new(type, castLib whichCast)
new(type, member whichCastMember of castLib whichCast)
variableName = new(parentScript arg1, arg2, ...)
new(script parentScriptName, value1, value2, ...)
timeout("name").new(timoutPeriod, #timeoutHandler, {, targetObject})
new(xtra "xtraName")
```

Description

Function; creates a new cast member, child object, timeout object, or Xtra instance and allows you to assign of individual property values to child objects.

For cast members, the *type* parameter sets the cast member's type. Possible predefined values correspond to the existing cast member types: `#bitmap`, `#field`, and so on. The `new` function can also create Xtra cast member types, which can be identified by any name that the author chooses.

It's also possible to create a new color cursor cast member using the Custom Cursor Xtra. Use `new(#cursor)` and set the properties of the resulting cast member to make them available for use.

The optional *whichCastMember* and *whichCast* parameters specify the cast member slot and Cast window where the new cast member is stored. When no cast member slot is specified, the first empty slot is used. The `new` function returns the cast member slot.

When the argument for the `new` function is a parent script, the `new` function creates a child object. The parent script should include an `on new` handler that sets the child object's initial state or property values and returns the `me` reference to the child object.

The child object has all the handlers of the parent script. The child object also has the same property variable names that are declared in the parent script, but each child object has its own values for these properties.

Because a child object is a value, it can be assigned to variables, placed in lists, and passed as a parameter.

As with other variables, you can use the `put` command to display information about a child object in the Message window.

When `new()` is used to create a timeout object, the timeoutPeriod sets the number of milliseconds between timeout events sent by the timeout object. The *#timeoutHandler* is a symbol that identifies the handler that will be called when each timeout event occurs. The *targetObject* identifies the name of the child object that contains the *#timeoutHandler*. If no *targetObject* is given, the *#timeoutHandler* is assumed to be in a movie script. The timeout creation syntax might vary depending on the `scriptExecutionStyle` setting.

```
-- Lingo syntax when scriptExecutionStyle is set to 9
x = timeout(name).new(period,handler,targetData)

-- Lingo syntax when scriptExecutionStyle is set to 10
x = timeout().new(name, period, handler, targetData)
y = new timeout(name, period, handler, targetData)

// JavaScript syntax
x = new timeout(name, period, function, targetData)
```

When a timeout object is created, it enables its *targetObject* to receive the system events prepareMovie, startMovie, stopMovie, prepareFrame, and exitFrame. To take advantage of this, the *targetObject* must contain handlers for these events. The events do not need to be passed in order for the rest of the movie to have access to them.

Note: *A Lingo-created timeout object can call a JavaScript syntax function, and vice versa.*

To see an example of new() used in a completed movie, see the Parent Scripts, and Read and Write Text movies in the Learning/Lingo folder inside the Director application folder.

Example

To create a new bitmap cast member in the first available slot, you use this syntax:

```
set newMember = new(#bitmap)
```

After the line has been executed, newMember will contain the member reference to the cast member just created:

```
put newMember
-- (member 1 of castLib 1)
```

If you are using JavaScript syntax to create a new cast members, use the movie object's newMember() method. This statement creates a new bitmap cast member:

```
var tMem = _movie.newMember(symbol("bitmap"))
```

The following startMovie script creates a new Flash cast member using the new command, sets the newly created cast member's linked property so that the cast member's assets are stored in an external file, and then sets the cast member's pathName property to the location of a Flash movie on the World Wide Web:

```
on startMovie
    flashCastMember = new(#flash)
    member(flashCastMember).pathName = "http://www.someURL.com/myFlash.swf"
end
```

When the movie starts, this handler creates a new animated color cursor cast member and stores its cast member number in a variable called customCursor. This variable is used to set the castMemberList property of the newly created cursor and to switch to the new cursor.

```
on startmovie
    customCursor = new(#cursor)
    member(customCursor).castMemberList = [member 1, member 2, member 3]
    cursor (member(customCursor))
end
```

These statements from a parent script include the on new handler to create a child object. The parent script is a script cast member named Bird, which contains these handlers.

```
on new me, nameForBird
    return me
end

on fly me
    put "I am flying"
end
```

The first statement in the following example creates a child object from the above script in the preceding example, and places it in a variable named myBird. The second statement makes the bird fly by calling the fly handler in the Bird parent script:

```
myBird = script("Bird").new()
myBird.fly()
```

This statement uses a new Bird parent script, which contains the property variable `speed`:

```
property speed
on new me, initSpeed
    speed = initSpeed
    return me
end
on fly me
    put "I am flying at " & speed & "mph"
end
```

The following statements create two child objects called myBird1 and myBird2. They are given different starting speeds: 15 and 25, respectively. When the `fly` handler is called for each child object, the speed of the object is displayed in the Message window.

```
myBird1 = script("Bird").new(15)
myBird2 = script("Bird").new(25)
myBird1.fly()
myBird2.fly()
```

This message appears in the Message window:

```
-- "I am flying at 15 mph"
-- "I am flying at 25 mph"
```

This statement creates a new timeout object called intervalTimer that will send a timeout event to the `on minuteBeep` handler in the child object playerOne every 60 seconds:

```
timeout("intervalTimer").new(60000, #minuteBeep, playerOne)
```

This statement creates a sample JavaScript function:

```
function sampleTimeout () { trace("hello"); }
```

Elsewhere in your movie you can use this statement to create a timeout object that calls the JavaScript function:

```
-- Lingo syntax
gTO = timeout().new("test",50,"sampleTimeout",0)

// JavaScript syntax
_global.gTO = new timeout("test",50,"sampleTimeout",0)
```

See also

on stepFrame, actorList, ancestor, end, type (Member), timeout()

newCamera

Usage

```
member(whichCastmember).newCamera(newCameraName)
```

Description

3D command; creates a new camera within a cast member.

Parameters

newCameraName Required. Specifies the name of the new camera. The name of the new camera must be unique within the cast member.

Example

This statement creates a new camera called in-car camera:

```
--Lingo
member("3D World").newCamera("in-car camera")

// Javascript
member("3D World").newCamera("in-car camera") ;
```

newColorRatio

Usage

```
<ColorRatio> newColorRatio(color, ratio, alpha)
```

Description

Global function; Creates a new ColorRatio object which is used to mention the coloring proportions to be applied at different ratio levels when using a gradient glow or gradient bevel filter.

Parameters

Parameter	Description
color	Required. Color object consisting of red, green and blue components of the color.
ratio	Required. Value between 0-255 determining the start point for the color distribution w.r.t the current ColorRatio object in the ColorList of gradient filters.
alpha	Optional. Transparancy level(0-255) of the color.

Example

The following code snippet creates a GradientGlowFilter and assigns various color ratios to the filter.

```
--Lingo

on mouseUp me
  fil = filter(#GradientGlowFilter, [#distance:0, #blurX:10, #blurY:10, #strength:1.3,
#inner:0])
  fil.colorList.append(newColorRatio(color(255,255,255),1,0))
  fil.colorList.append(newColorRatio(color(255,0,0),63,255))
  fil.colorList.append(newColorRatio(color(255,255,0),126,25))
  fil.colorList.append(newColorRatio(color(0,204,255),255,255))
  sprite(me.spriteNum).filterList.append(fil)
end

//Java Script

function mouseUp(me)

{
  fil = filter(symbol("GradientGlowFilter"), propList(symbol("distance"),0,
symbol("blurX"), 10, symbol("blurY"), 10, symbol("strength"), 1.3, symbol("inner"), 0));
  fil.colorList.append(newColorRatio(color(255,255,255),1,0));
  fil.colorList.append(newColorRatio(color(255,0,0),63,255));
  fil.colorList.append(newColorRatio(color(255,255,0),126,255));
  fil.colorList.append(newColorRatio(color(0,204,255),255,255));
  sprite(me.spriteNum).filterList.append(fil);
}
```

newCurve()

Usage

```
-- Lingo syntax
memberObjRef.newCurve(positionInVertexList)

// JavaScript syntax
memberObjRef.newCurve(positionInVertexList);
```

Description

Function; adds a `#newCurve` symbol to the `vertexList` of *`vectorCastMember`*, which adds a new shape to the vector shape. You can break apart an existing shape by calling `newCurve()` with a position in the middle of a series of vertices.

Parameters

`positionInVertexList` Required. Specifies the position in the `vertexList` at which the `#newCurve` symbol is added.

Example

These statements add a new curve to cast member 2 at the third position in the cast member's vertexList. The second line of the example replaces the contents of curve 2 with the contents of curve 3.

```
-- Lingo syntax
member(2).newCurve(3)
member(2).curve[2] = member(2).curve[3]

// JavaScript syntax
member(2).newCurve(3);
member(2).curve[2] = member(2).curve[3]
```

See also

`curve`, `vertexList`

newGroup

Usage

```
member(whichCastmember).newGroup(newGroupName)
```

Description

3D command; creates a new group and adds it to the group palette.

Parameters

`newGroupName` Required. Specifies the name of the new group. The name of the new group must be unique within the group palette.

Example

This statement creates a group called gbGroup2 within the cast member Scene, and a reference to it is stored in the variable `ng`:

```
-- Lingo
ng = member("Scene").newGroup("gbGroup2")
// Javascript
```

```
var ng = member("Scene").newGroup("gbGroup2") ;
```

newLight

Usage

```
member(whichCastmember).newLight(newLightName, #typeIndicator)
```

Description

3D command; creates a new light with a specified type, and adds it to the light palette.

Parameters

newLightName Required. Specifies the name of the new light. The name of the new light must be unique within the light palette.

typeIndicator Required. A symbol that specifies the type of the new light. Valid values include the following:

- `#ambient` is a generalized light in the 3D world.
- `#directional` is a light from a specific direction.
- `#point` is a light source like a light bulb.
- `#spot` is a spotlight effect.

Example

The following statement creates a new light in the cast member named 3D World. It is an ambient light called "ambient room light".

```
-- Lingo
member("3D World").newLight("ambient room light", #ambient)

// Javascript
member("3D World").newLight("ambient room light", symbol("ambient"));
```

newMatrix()

Usage

```
<Matrix> newMatrix(numRows, numColumns, {elementList})
```

Description

Global Function; creates a new matrix object with the specified number of rows and columns. Indices for rows and columns start from 1.

Note: *To see the values of a matrix created using newmatrix in the debug mode, query the values using the getval(i,j) method.*

Parameters

numRows Required. Specifies the number of rows in the matrix.

numColumns Required. Specifies the number of columns in the matrix.

elementList Required. A linear list of floating point numbers.

Example

The following statement creates a matrix with 2 rows and 3 columns with first row as [1,2,3] and second row as [4,5,6].

```
-- Lingo
mat = newMatrix(2, 3, [1,2,3,4,5,6])

// Javascript
mat = newMatrix(2, 3, list(1,2,3,4,5,6));
```

See also

getVal(), setVal(), numRows(), numColumns(), matrixAddition(), matrixMultiply(), matrixMultiplyScalar(), matrixTranspose()

newMember()

Usage

```
-- Lingo syntax
_movie.newMember(symbol)
_movie.newMember(stringMemberType)

// JavaScript syntax
_movie.newMember(stringMemberType);
```

Description

Movie method; creates a new cast member and allows you to assign individual property values to child objects.

For new cast members, the *symbol* or *stringMemberType* parameter sets the cast member's type. Possible predefined values correspond to the existing cast member types: #bitmap, #field, and so on. The newMember() method can also create Xtra cast member types, which can be identified by any name that the author chooses.

It's also possible to create a new color cursor cast member using the Custom Cursor Xtra. Use newMember(#cursor) and set the properties of the resulting cast member to make them available for use.

After newMember() is called, the new cast member is placed in the first empty cast library slot.

To see an example of newMember() used in a completed movie, see the Parent Scripts, and Read and Write Text movies in the Learning/Lingo folder inside the Director application folder.

Parameters

symbol (Lingo only) Required. A symbol that specifies the type of the new cast member.

stringMemberType Required. A string that specifies the type of the new cast member.

Example

The following statements create a new bitmap cast member and assign it to the variable newBitmap.

```
-- Lingo syntax
newBitmap = _movie.newMember(#bitmap) -- using a symbol
newBitmap = _movie.newMember("bitmap") -- using a string

// JavaScript syntax
var newBitmap = _movie.newMember(symbol("bitmap")) ; //using a symbol
var newBitmap = _movie.newMember("bitmap") ; // using a string
```

See also

Movie, type (Member)

newMesh

Usage

```
member(whichCastmember).newMesh(name,numFaces, numVertices,
numNormals,numColors,numTextureCoordinates)
```

Description

3D command; creates a new mesh model resource. After creating a mesh, you must set values for at least the `vertexList` and `face[index].vertices` properties of the new mesh, followed by a call to its `build()` command, in order to actually generate the geometry.

Parameters

meshName Required. Specifies the name of the new mesh model resource.

numFaces Required. Specifies the desired total number of triangles you want in the mesh.

numVertices Required. Specifies the total number of vertices used by all the (triangular) faces. A vertex may be shared by more than one face.

numNormals Optional. Specifies the total number of normals. A normal may be shared by more than one face. The normal for a corner of a triangle defines which direction is outward, affecting how that corner is illuminated by lights. Enter 0 or omit this parameter if you are going to use the mesh's `generateNormals()` command to generate normals.

numColors Optional. Specifies the total number of colors used by all the faces. A color may be shared by more than one face. You can specify a color for each corner of each face. Specify colors for smooth color gradation effects. Enter 0 or omit this parameter to get default white color per face corner.

numTextureCoordinates Optional. Specifies the number of user-specified texture coordinates used by all the faces. Enter 0 or omit this parameter to get the default texture coordinates generated via a planar mapping. (See the explanation of `#planar` in the `shader.textureWrapMode` entry for more details). Specify texture coordinates when you need precise control over how textures are mapped onto the faces of the mesh.

Example

This example creates a model resource of the type `#mesh`, specifies its properties, and then creates a new model from the model resource. The process is outlined in the following line-by-line explanation of the example code:

Line 1 creates a mesh containing 6 faces, composed of 5 unique vertices and 3 unique colors. The number of normals and the number of `textureCoordinates` are not set. The normals will be created by the `generateNormals` command.

Line 2 defines the five unique vertices used by the faces of the mesh.

Line 3 defines the three unique colors used by the faces of the mesh.

Lines 4 through 9 assign which vertices to use as the corners of each face in the Pyramid. Note the clockwise ordering of the vertices. `GenerateNormals()` relies on a clockwise ordering.

Lines 10 through 15 assign colors to the corners of each face. The colors will spread across the faces in gradients.

Line 16 creates the normals of Triangle by calling the `generateNormals()` command.

Line 17 calls the `build` command to construct the mesh.

```
nm = member("Shapes").newMesh("pyramid",6 , 5, 0, 3)
nm.vertexList = [ vector(0,0,0), vector(40,0,0), vector(40,0,40), vector(0,0,40),
vector(20,50,20) ]
nm.colorList = [ rgb(255,0,0), rgb(0,255,0), rgb(0,0,255) ]
nm.face[1].vertices = [ 4,1,2 ]
nm.face[2].vertices = [ 4,2,3 ]
nm.face[3].vertices = [ 5,2,1 ]
nm.face[4].vertices = [ 5,3,2 ]
nm.face[5].vertices = [ 5,4,3 ]
nm.face[6].vertices = [ 5,1,4 ]
nm.face[1].colors = [3,2,3]
nm.face[2].colors = [3,3,2]
nm.face[3].colors = [1,3,2]
nm.face[4].colors = [1,2,3]
nm.face[5].colors = [1,3,2]
nm.face[6].colors = [1,2,3]
nm.generateNormals(#flat)
nm.build()
nm = member("Shapes").newModel("Pyramid1", nm)
```

See also

newModelResource

newModel

Usage

```
member( whichCastmember ).newModel( newModelName {, whichModelResource } )
```

Description

3D command; creates a new model in the referenced cast member. All new models have their `resource` property set to `VOID` by default.

Parameters

`newModelName` Required. Specifies the name of the new model. The name of the new model must be unique.

`whichModelResource` Optional. Specifies a model resource to create the model from.

Example

This statement creates a model called New House within the cast member 3D World.

```
-- Lingo
member("3D World").newModel("New House")

// Javascript
member("3D World").newModel("New House") ;
```

Alternatively, the model resource for the new model can be set with the optional `whichModelResource` parameter.

```
member("3D World").newModel("New House", member("3D World").modelResource("bigBox"))
```

newModelResource

Usage

```
member(whichCastmember).newModelResource(newModelResourceName { ,#type, #facing })
```

Description

3D command; creates a new model resource, optionally of a give type and facing, and adds it to the model resource palette.

If you do not choose to specify the *facing* parameter and specify #box, #sphere, #particle or #cylinder for the *type* parameter, only the front faces are generated. If you specify #plane, both the front and back faces are generated. Model resources of the type #plane have two meshes generated (one for each side), and consequently has two shaders in the shaderList.

A facing of #both creates the double amount of meshes and consequently produces double the number of shader entries in the shaderList. There will be 2 for planes and spheres (for the inside and outside of the model respectively), 12 for cubes (6 on the outside, 6 on the inside), and 6 for cylinders (top, hull and bottom outside, and another set for the inside).

Parameters

newModelResourceName Required. Specifies the name of the new model resource.

type Optional. Specifies the primitive type of the new model resource. Valid values are as follows:

- #plane
- #box
- #sphere
- #cylinder
- #particle

facing Optional. Specifies the face of the new model resource. Valid values are as follows:

- #front
- #back
- #both

Example

The following handler creates a box. The first line of the handler creates a new model resource called box10. Its type is #box, and it is set to show only its back. The next three lines set the dimensions of box10 and the last line creates a new model which uses box10 as its model resource.

```
on makeBox
    nmr = member("3D").newModelResource("box10", #box, #back)
    nmr.height = 50
    nmr.width = 50
    nmr.length = 50
    aa = member("3D").newModel("gb5", nmr)
end
```

This statement creates a box-shaped model resource called hatbox4.

```
member("Shelf").newModelResource("hatbox4", #box)
```

See also

primitives

newMotion()

Usage

```
member(whichCastmember).newMotion(name)
```

Description

3D command; creates a new motion within a referenced cast member, and returns a reference to the new motion. A new motion can be used to combine several previously existing motions from the member's motion list via the `map()` command.

Parameters

name Required. Specifies the name of the new motion. The name of the new motion must be unique within the referenced cast member.

Example

This Lingo creates a new motion in member 1 called `runWithWave` that is used to combine the run and wave motions from the member's motion list:

```
runWithWave = member(1).newMotion("runWithWave")
runWithWave.map("run", "pelvisBone")
runWithWave.map("wave", "shoulderBone")
```

newObject()

Usage

```
-- Lingo syntax
spriteObjRef.newObject(objectType {, arg1, arg2 ....})

// JavaScript syntax
spriteObjRef.newObject(objectType {, arg1, arg2 ....});
```

Description

Flash sprite command; creates an ActionScript object of the specified type.

The following syntax creates an object within a Flash sprite:

```
flashSpriteReference.newObject("objectType" {, arg1, arg2 ....})
```

The following syntax creates a global object:

```
newObject("objectType" {, arg1, arg2 ....})
```

Note: *If you have not imported any Flash cast members, you must manually add the Flash Asset Xtra to your movie's Xtra list in order for global Flash commands to work correctly in the Shockwave Player and projectors. You add Xtra extensions to the Xtra list by choosing Modify > Movie > Xtras. For more information about managing Xtra extensions for distributed movies, see the Using Director topics in the Director Help Panel.*

Parameters

objectType Required. Specifies the type of new object to create.

arg1, arg2, ... Optional. Specifies any initialization arguments required by the object. Each argument must be separated by a comma.

Example

This Lingo sets the variable `tLocalConObject` to a reference to a new `LocalConnection` object in the Flash movie in sprite 3:

```
-- Lingo syntax
tLocalConObject = sprite(3).newObject("LocalConnection")

// JavaScript syntax
var tLocalConObject = sprite(3).newObject("LocalConnection");
```

The following Lingo sets the variable `tArrayObject` to a reference to a new array object in the Flash movie in sprite 3. The array contains the 3 integer values 23, 34, and 19.

```
-- Lingo syntax
tArrayObject = sprite(3).newObject("Array",23,34,19)

// JavaScript syntax
var tArrayObject = sprite(3).newObject("Array",23,34,19);
```

See also

`setCallback()`, `clearAsObjects()`

newShader

Usage

```
member(whichCastmember).newShader(newShaderName, #shaderType)
```

Description

3D command; creates a new shader of a specified shader type within a referenced cast member's shader list and returns a reference to the new shader.

Each type of shader has a specific group of properties that can be used with that type of shader, in addition all shader types have access to the `#standard` shader properties. However, although you can assign any `#standard` shader property to a shader of another type, the property may not have a visual effect. This happens in cases where the `#standard` property, if applied, would override the nature of the shader type. An example of this is the `diffuseLightMap` standard shader property, which is ignored by `#engraver`, `#newsprint`, and `#painter` type shaders.

Parameters

newShaderName Required. Specifies the name of the new shader. The name of the new shader must be unique in the shader list.

shaderType Required. A symbol that determines the style in which the shader is applied. Valid values include the following:

- `#standard` shaders are photorealistic, and have the following properties: `ambient`, `blend`, `blendConstant`, `blendConstantList`, `blendFunction`, `blendFunctionList`, `blendSource`, `blendSourceList`, `diffuse`, `diffuseLightMap`, `emissive`, `flat`, `glossMap`, `ilk`, `name`, `region`, `renderStyle`, `silhouettes`, `specular`, `specularLightMap`, `texture`, `textureMode`, `textureModeList`, `textureRepeat`, `textureRepeatList`, `textureTransform`, `textureTransformList`, `transparent`, `useDiffuseWithTexture`, `wrapTransform`, and `wrapTransformList`.
- `#painter` shaders are smoothed out, have the appearance of a painting, and have the following properties in addition to all of the `#standard` properties: `colorSteps`, `hilightPercentage`, `hilightStrength`, `name`, `shadowPercentage`, `shadowStrength`, and `style`.
- `#engraver` shaders are lined, have the appearance of an engraving, and have the following properties in addition to all of the #standard properties: `brightness`, `density`, `name`, and `rotation`.
- `#newsprint` shaders are in a simulated dot style, have the appearance of a newspaper reproduction, and have the following properties in addition to all of the #standard properties: `brightness`, `density`, and `name`.

Example

This statement creates a `#painter` shader called newPainter:

```
-- Lingo
newPainter = member("3D World").newShader("newPainter",#painter)

// Javascript
var newPainter = member("3D World").newShader("newPainter",symbol("painter")) ;
```

See also

shadowPercentage

newTexture

Usage

```
member(whichCastmember).newTexture(newTextureName {,#typeIndicator,
sourceObjectReference})
```

Description

3D command; creates a new texture within the referenced member's texture palette and returns a reference to the new texture. The only way cast member textures will work is if you specify the cast member in the `newTexture` constructor.

Parameters

`newTextureName` Required. Specifies the name of the new texture. The name of the new texture must be unique in the referenced cast member's texture palette.

`typeIndicator` Optional. Specifies the type of the new texture. If omitted, the new texture is created with no specific type. Valid values include the following:

- `#fromCastMember` (a cast member)
- `#fromImageObject` (a Lingo image object)

sourceObjectReference Optional. Specifies a reference to the source cast member or Lingo image object. If omitted, the new texture is created from no specific source. *sourceObjectReference* must refer to a cast member if *typeIndicator* is `#fromCastMember`, and it must refer to a Lingo image object if *typeIndicator* is `#fromImageObject`.

Example

The first line of this statement creates a new texture called Grass 02 from cast member 5 of castlib 1. The second line creates a blank new texture called Blank.

```
-- Lingo
member("3D World").newTexture("Grass 02",#fromCastMember,member(5,1))
member("3D World").newTexture("Blank")

// Javascript
member("3D World").newTexture("Grass 02",symbol("fromCastMember"),member(5,1)) ;
member("3D World").newTexture("Blank") ;
```

normalize

Usage

```
normalize(vector)
vector.normalize()
```

Description

3D command; normalizes a vector by dividing the *x, y,* and *z* components by the vector's magnitude. Vectors that have been normalized always have a magnitude of 1.

Parameters

None.

Example

This statement shows the value of the vector MyVec before and after being normalized:

```
-- Lingo
MyVec = vector(-209.9019, 1737.5126, 0.0000)
MyVec.normalize()
put MyVec
-- vector(-0.1199, 0.9928, 0.0000)
put MyVec.magnitude
-- 1.0000

// Javascript
MyVec = vector(-209.9019, 1737.5126, 0.0000);
MyVec.normalize();
trace(MyVec);
// vector(-0.1199, 0.9928, 0.0000)
trace(MyVec.magnitude);
// 1.0000
```

This statement shows the value of the vector ThisVector before and after being normalized.

```
-- Lingo
ThisVector = vector(-50.0000, 0.0000, 0.0000)
normalize(ThisVector)
put ThisVector
```

```
-- vector(-1.0000, 0.0000, 0.0000)

// Javascript
ThisVector = vector(-50.0000, 0.0000, 0.0000);
normalize(ThisVector);
trace(ThisVector);
// vector(-1.0000, 0.0000, 0.0000)
```

See also

getNormalized, randomVector(), magnitude

nothing

Usage

```
nothing
```

Description

Command; does nothing. This command is useful for making the logic of an `if...then` statement more obvious. A nested `if...then...else` statement that contains no explicit command for the `else` clause may require `else nothing`, so that Lingo does not interpret the `else` clause as part of the preceding `if` clause.

Parameters

None.

Example

The nested `if...then...else` statement in this handler uses the `nothing` command to satisfy the statement's `else` clause:

```
-- Lingo syntax
on mouseDown
    if the clickOn = 1 then
        if sprite(1).moveableSprite = TRUE then member("Notice").text = "Drag the ball"
        else nothing
    else member("Notice").text = "Click again"
    end if
end

// JavaScript syntax
function mouseDown() {
    if (_mouse.clickOn == 1) {
        if (sprite(1).moveableSprite) {
            member("Notice").text = "Drag the ball";
        } else {
            // do nothing
        }
    } else {
        member("Notice").text = "Click again";
    }
}
```

This handler instructs the movie to do nothing so long as the mouse button is being pressed:

```
-- Lingo syntax
on mouseDown
    repeat while the stillDown
        nothing
```

```
    end repeat
end mouseDown

// JavaScript syntax
function mouseDown() {
    do {
        // do nothing
    } while _mouse.stillDown;
}
```

See also

if

nudge()

Usage

```
-- Lingo syntax
spriteObjRef.nudge(#direction)

// JavaScript syntax
spriteObjRef.nudge(#direction);
```

Description

QuickTime VR command; nudges the view perspective of the specified QuickTime VR sprite in a specified direction.

Nudging to the right causes the image of the sprite to move to the left. The `nudge` command has no return value.

Parameters

`direction` Required. Specifies the direction to nudge the view perspective. Valid values include the following:

- `#down`
- `#downLeft`
- `#downRight`
- `#left`
- `#right`
- `#up`
- `#upLeft`
- `#upRight`

Example

This handler causes the perspective of the QTVR sprite to move to the left as long as the mouse button is held down on the sprite:

```
-- Lingo syntax
on mouseDown me
    repeat while the stillDown
        sprite(1).nudge(#left)
    end repeat
end
```

```
// JavaScript syntax
function mouseDown() {
    do {
        sprite(1).nudge(symbol("left"));
    } while _mouse.stillDown;
}
```

numColumns()

Usage

```
<matrixObject>.numColumns
```

Description

Matrix property; gets the number of columns of the matrix. You can only get the numColumns; It can't be changed.

Example

The following code snippet creates a matrix and prints its numRows and numColumns.

```
--Lingo

mat1 = newMatrix(2, 3, [1,2,3,4,5,6])
put(mat1.numRows)
put(mat1.numColumns)
-- 2
-- 3

//Java Script

mat1 = newMatrix(2, 3, list(1,2,3,4,5,6));
put(mat1.numRows);
put(mat1.numColumns);

// 2

// 3
```

See also

getVal(), setVal(), numRows(), matrixAddition(), matrixMultiply(), matrixMultiplyScalar(), matrixTranspose(), newMatrix()

numRows()

Usage

```
<matrixObject>.numRows
```

Description

Matrix property; gets the number of rows of the matrix. You can only get the numRows; It can't be changed.

Example

The following code snippet creates a matrix and prints its numRows and numColumns.

```
-- Lingo
mat1 = newMatrix(2, 3, [1,2,3,4,5,6])
put(mat1.numRows)
```

```
put(mat1.numColumns)

-- 2
-- 3

// Javascript
mat1 = newMatrix(2, 3, list(1,2,3,4,5,6));
put(mat1.numRows);
put(mat1.numColumns);

// 2
// 3
```

See also

getVal(), setVal(), numColumns(), matrixAddition(), matrixMultiply(), matrixMultiplyScalar(), matrixTranspose(), newMatrix()

numToChar()

Usage

```
numToChar(integerExpression)
```

Description

Function; displays a string containing the single character whose ASCII number is the value of a specified expression. This function is useful for interpreting data from outside sources that are presented as numbers rather than as characters.

ASCII values up to 127 are standard on all computers. Values of 128 or greater refer to different characters on different computers.

Parameters

integerExpression Required. Specifies the ASCII number whose corresponding character is returned.

Example

This handler removes any nonalphabetic characters from any arbitrary string and returns only capital letters:

```
-- Lingo syntax
on ForceUppercase input
    output = EMPTY
    num = length(input)
    repeat with i = 1 to num
        theASCII = charToNum(input.char[i])
        if theASCII = min(max(96, theASCII), 123) then
            theASCII = theASCII - 32
            if theASCII = min(max(63, theASCII), 91) then
                put numToChar(theASCII) after output
            end if
        end if
    end repeat
    return output
end

// JavaScript syntax
function ForceUpperCase(input) {
    output = "";
    num = input.length;
```

```
    for (i=1;i<=num;i++) {
        theASCII = charToNum (input.getProp("char",i);
        if (theASCII == list(list(96, theASCII).max(), 123).min()) {
            theASCII = theASCII - 32;
            if (theASCII == list(list(63, theASCII).max(), 91).min()) {
                output = output + numToChar (theASCII);
            }
        }
    }
    return output;
}
```

See also

charToNum()

objectP()

Usage

```
objectP(expression)
```

Description

Function; indicates whether a specified expression is an object produced by a parent script, Xtra, or window (TRUE) or not (FALSE).

The *P* in objectP stands for *predicate*.

It is good practice to use objectP to determine which items are already in use when you create objects by parent scripts or Xtra instances.

To see an example of objectP() used in a completed movie, see the Read and Write Text movie in the Learning/Lingo folder inside the Director application folder.

Parameters

expression Required. Specifies the expression to test.

Example

This Lingo checks whether the global variable gDataBase has an object assigned to it and, if not, assigns one. This check is commonly used when you perform initializations at the beginning of a movie or section that you don't want to repeat.

```
-- Lingo syntax
if objectP(gDataBase) then
 nothing
else
 gDataBase = script("Database Controller").new()
end if

// JavaScript syntax
if (objectP(gDataBase)) {
 // do nothing
} else {
 gDataBase = script("Database Controller").new();
}
```

See also

floatP(), ilk(), integerP(), stringP(), symbolP()

offset() (string function)

Usage

```
offset(stringExpression1, stringExpression2)
```

Description

Function; returns an integer indicating the position of the first character of a string in another string. This function returns 0 if the first string is not found in the second string. Lingo counts spaces as characters in both strings.

On the Mac, the string comparison is not sensitive to case or diacritical marks. For example, Lingo considers *a* and *Å* to be the same character on the Mac.

Parameters

stringExpression1 Required. Specifies the sub-string to search for in *stringExpression2*.

stringExpression2 Required. Specifies the string that contains the sub-string *stringExpression1*.

Example

This statement displays in the Message window the beginning position of the string "media" within the string "kleptomedia":

```
put offset("media","kleptomedia")
```

The result is 7.

This statement displays in the Message window the beginning position of the string "Micro" within the string "Adobe":

```
put offset("Micro", "Adobe")
```

The result is 0, because "Adobe" doesn't contain the string "Micro".

This handler finds all instances of the string represented by `stringToFind` within the string represented by `input` and replaces them with the string represented by `stringToInsert`:

```
-- Lingo syntax
on SearchAndReplace input, stringToFind, stringToInsert
    output = ""
    findLen = stringToFind.length - 1
    repeat while input contains stringToFind
        currOffset = offset(stringToFind, input)
        output = output & input.char [1..currOffset]
        delete the last char of output
        output = output & stringToInsert
        delete input.char [1.. (currOffset + findLen)]
    end repeat
    set output = output & input
    return output
end

// JavaScript syntax
function SearchAndReplace(input, stringToFind, stringToInsert) {
    output = "";
```

```
    findLen = stringToFind.length - 1;
    do {
        currOffset = offset(stringToFind, input);
        output = output + input.char[0..currOffset];
        output = output.substr(0,output.length-2);
        output = output + stringToInsert;
        input = input.substr(currOffset+findLen,input.length);
    } while (input.indexOf(stringToFind) >= 0);
    output = output + input;
    return output;
}
```

See also

chars(), length(), contains, starts

offset() (rectangle function)

Usage

```
rectangle.offset(horizontalChange, verticalChange)
offset (rectangle, horizontalChange, verticalChange)
```

Description

Function; yields a rectangle that is offset from the rectangle specified by *rectangle*.

Parameters

horizontalChange Required. Specifies the horizontal offset, in pixels. When *horizontalChange* is greater than 0, the offset is toward the right of the Stage; when *horizontalChange* is less than 0, the offset is toward the left of the Stage.

verticalChange Required. Specifies the vertical offset, in pixels. When *verticalChange* is greater than 0, the offset is toward the top of the Stage; when *verticalChange* is less than 0, the offset is toward the bottom of the Stage.

Example

This handler moves sprite 1 five pixels to the right and five pixels down:

```
-- Lingo syntax
on diagonalMove
    newRect=sprite(1).rect.offset(5, 5)
    sprite(1).rect=newRect
end

// JavaScript syntax
function diagonalMove() {
    newRect = sprite(1).rect.offset(5,5);
    sprite(1).rect = newRect;
}
```

open() (Player)

Usage

```
-- Lingo syntax
_player.open({stringDocPath,} stringAppPath)
```

```
// JavaScript syntax
_player.open({stringDocPath,} stringAppPath);
```

Description

Player method; opens a specified application, and optionally opens a specified file when the applicatin opens.

When either *stringDocPath* or *stringAppPath* are in a different folder than the current movie, you must specify the full pathname to the file or files.

The computer must have enough memory to run both Director and other applications at the same time.

This is a very simple method for opening an application or a document within an application. For more control, look at options available in third-party Xtra extensions.

Parameters

stringDocPath Optional. A string that specifies the document to open when the application specified by *stringAppPath* opens.

stringAppPath Required. A string that specifies the path to the application to open.

Example

This statement opens the TextEdit application, which is in the folder Applications on the drive HD (Mac), and the document named Storyboards:

```
-- Lingo syntax
_player.open("Storyboards", "HD:Applications:TextEdit")

// JavaScript syntax
_player.open("Storyboards", "HD:Applications:TextEdit");
```

See also

Player

open() (Window)

Usage

```
-- Lingo syntax
windowObjRef.open()

// JavaScript syntax
windowObjRef.open();
```

Description

Window method; opens a window and positions it in front of all other windows.

If no movie is assigned to the window on which `open()` is called, the Open File dialog box appears.

If the reference to the window object *windowObjRef* is replaced with a movie's filename, the window uses the filename as the window name. However, a movie must then be assigned to the window by using the window's `fileName` property.

If the reference to the window object *windowObjRef* is replaced with a window name, the window takes that name. However, a movie must then be assigned to the window by using the window's `fileName` property.

To open a window that uses a movie from a URL, use `downloadNetThing()` to download the movie's file to a local disk first, and then use the file on the disk. This procedure minimizes problems with waiting for the movie to download.

When using a local movie, use `preloadMovie()` to load at least the first frame of the movie prior to calling `open()`. This procedure reduces the possibility of movie load delays.

Opening a movie in a window is currently not supported in playback using a browser.

Parameters

None.

Example

This statement opens the window Control Panel and brings it to the front:

```
-- Lingo syntax
window("Control Panel").open()

// JavaScript syntax
window("Control Panel").open();
```

See also

close(), downloadNetThing, fileName (Window), preLoadMovie(), Window

openFile()

Usage

```
-- Lingo syntax
fileioObjRef.openFile(stringFileName, intMode)

// JavaScript syntax
fileioObjRef.openFile(stringFileName, intMode)
```

Description

Fileio method; Opens a specified file with a specified mode.

Parameters

stringFileName Required. A string that specifies the full path and name of the file to open.

intMode Required. An integer that specifies the mode of the file. Valid values include:

- 0—Read/write
- 1—Read-only
- 2—Writeable

Example

The following statement opens a file c:\xtra.txt with Read/Write permission.

```
-- Lingo
objFileio = new xtra("fileio")
objFileio.openFile("c:\xtra.txt",0)

// JavaScript syntax
```

```
var objFileio = new xtra("fileio");
objFileio.openFile("c:\xtra.txt",0);
```

See also

Fileio

openXlib

Usage

```
openXlib whichFile
```

Description

Command; opens a specified Xlibrary file.

It is good practice to close any file you have opened as soon as you are finished using it. The `openXlib` command has no effect on an open file.

The `openXlib` command doesn't support URLs as file references.

Xlibrary files contain Xtra extensions. Unlike `openResFile`, `openXlib` makes these Xtra extensions known to Director.

When you open a Scripting Xtra extension using `openXlib`, you must use `closeXlib` to close it when Director is finished using it.

In Windows, the .dll extension is optional.

Note: *This command is not supported in Shockwave Player.*

Parameters

`whichFile` Required. Specifies the Xlibrary file to open. If the file is not in the folder containing the current movie, *`whichFile`* must include the pathname.

Example

This statement opens the Xlibrary file Video Disc Xlibrary:

```
openXlib "Video Disc Xlibrary"
```

This statement opens the Xlibrary file Xtras, which is in a different folder than the current movie:

```
openXlib "My Drive:New Stuff:Transporter Xtras"
```

See also

closeXlib, Interface()

param()

Usage

```
param(parameterPosition)
```

Description

Function; provides the value of a parameter passed to a handler.

To avoid errors in a handler, this function can be used to determine the type of a particular parameter.

Parameters

parameterPosition Required. Specifies the parameter's position in the arguments passed to a handler.

Example

This handler accepts any number of arguments, adds all the numbers passed in as parameters, and then returns the sum:

```
--Lingo syntax
on AddNumbers
    sum = 0
    repeat with currentParamNum = 1 to the paramCount
        sum = sum + param(currentParamNum)
    end repeat
    return sum
end

// JavaScript syntax
function AddNumbers() {
    sum = 0;
    for (currentParamNum=0;currentParamNum<arguments.length;currentParamNum++){
        sum = sum + arguments[currentParamNum];
    }
    return sum;
}
```

You would use it by passing in the values you wanted to add:

```
put AddNumbers(3, 4, 5, 6)
-- 18
put AddNumbers(5, 5)
-- 10
```

See also

getAt, param(), paramCount(), return (keyword)

paramCount()

Usage

```
the paramCount
```

Description

Function; indicates the number of parameters sent to the current handler.

Parameters

None.

Example

This statement sets the variable `counter` to the number of parameters that were sent to the current handler:

```
set counter = the paramCount
```

parseString()

Usage

```
parserObject.parseString(stringToParse)
```

Description

Function; used to parse an XML document that is already fully available to the Director movie. The first parameter is the variable containing the parser object. The return value is `<VOID>` if the operation succeeds, or an error code number string if it fails. Failure is usually due to a problem with the XML syntax or structure. Once the operation is complete, the parser object contains the parsed XML data.

To parse XML at a URL, use `parseURL()`.

Parameters

`stringToParse` Required. Specifies the string of XML data to parse.

Example

This statement parses the XML data in the text cast member XMLtext. Once the operation is complete, the variable `gParserObject` will contain the parsed XML data.

```
-- Lingo
errorCode = gParserObject.parseString(member("XMLtext").text)

// Javascript
errorCode = gParserObject.parseString(member("XMLtext").text);
```

See also

`getError() (XML)`, `parseURL()`

parseURL()

Usage

```
parserObject.parseURL(URLstring {,#handlerToCallOnCompletion} {, objectContainingHandler})
```

Description

Function; parses an XML document that resides at an external Internet location. The first parameter is the parser object containing an instance of the XML Parser Xtra.

This function returns immediately, so the entire URL may not yet be parsed. It is important to use the `doneParsing()` function in conjunction with `parseURL()` to determine when the parsing operation is complete.

Since this operation is asynchronous, meaning it may take some time, you can use optional parameters to call a specific handler when the operation completes.

The return value is void if the operation succeeds, or an error code number string if it fails.

To parse XML locally, use `parseString()`.

Parameters

`URLstring` Required. Specifies the actual URL at which the XML data resides.

handlerToCallOnCompletion Optional. Specifies the name of the handler that is to be executed once the URL is fully parsed.

objectContainingHandler Optional. Specifies the name of the script object containing the handler *handlerToCallOnCompletion*. If omitted, the handler is assumed to be a movie handler.

Example

This statement parses the file sample.xml at MyCompany.com. Use `doneParsing()` to determine when the parsing operation has completed.

```
--Lingo syntax
errorCode = gParserObject.parseURL("http://www.MyCompany.com/sample.xml")

// JavaScript syntax
errorCode = _global.gParserObject.parseURL("http://- www.MyCompany.com/sample.xml");
```

Note: *This example supposes that an instance of the Xtra has already been created, and a reference to that has been stored in the global variable named gParserObject.*

This Lingo parses the file sample.xml and calls the `on parseDone` handler. Because no script object is given with the `doneParsing()` function, the `on parseDone` handler is assumed to be in a movie script.

```
errorCode = gParserObject.parseURL("http://www.MyCompany.com/sample.xml", #parseDone)
```

The movie script contains the `on parseDone` handler:

```
on parseDone
    global gParserObject
    if voidP(gParserObject.getError()) then
        put "Successful parse"
    else
        put "Parse error:"
        put " " & gParserObject.getError()
    end if
end
```

This JavaScript syntax parses the file sample.xml and calls the `parseDone` function. Because no script object is given with the `doneParsing()` function, the `parseDone` function is assumed to be in a movie script

```
errorCode = _global.gParserObject.parseURL("http://- www.MyCompany.com/sample.xml",
symbol("parseDone"));
```

Note: *This example supposes that an instance of the Xtra has already been created, and a reference to that has been stored in the global variable named gParserObject.*

The movie script contains the `on parseDone` handler:

```
// JavaScript syntax
function parseDone () {
    if (_global.gParserObject.getError() == undefined) {
        trace("successful parse");
    } else {
        trace("Parse error:");
        trace(" " + _global.gParserObject.getError());
    }
}
```

This Lingo parses the document sample.xml at MyCompany.com and calls the `on parseDone` handler in the script object `testObject`, which is a child of the parent script TestScript:

```
parserObject = new(xtra "XMLParser")
testObject = new(script "TestScript", parserObject)
```

```
errorCode = gParserObject.parseURL("http://www.MyCompany.com/sample.xml", #parseDone,
testObject)
```

Here is the parent script TestScript:

```
property myParserObject

on new me, parserObject
    myParserObject = parserObject
end

on parseDone me
    if voidP(myParserObject.getError()) then
        put "Successful parse"
    else
        put "Parse error:"
        put " " & myParserObject.getError()
    end if
end
```

This JavaScript syntax parses the document sample.xml at MyCompany.com and calls the parseDone function in the object testObject, which is an instance of the defined TestScript class:

```
parserObject = new xtra("XMLParser");
testObject = new TestScript(parserObject);
errorCode = parserObject .parseURL("http://www.MyCompany.com/sam- ple.xml",
symbol("parseDone"), testObject)
```

Here is the TestScript class definition:

```
TestScript = function (aParser) {
    this.myParserObject = aParser;
}
TestScript.prototype.parseDone = function () {
    if (this.myParserObject.getError() == undefined) {
        trace("successful parse");
    } else {
        trace("Parse error:");
        trace(" " + this.myParserObject.getError());
    }
}
```

See also

getError() (XML), parseString()

pass

Usage

```
pass
```

Description

Command; passes an event message to the next location in the message hierarchy and enables execution of more than one handler for a given event.

The `pass` command branches to the next location as soon as the command runs. Any Lingo that follows the `pass` command in the handler does not run.

By default, an event message stops at the first location containing a handler for the event, usually at the sprite level.

If you include the `pass` command in a handler, the event is passed to other objects in the hierarchy even though the handler would otherwise intercept the event.

Parameters

None.

Example

This handler checks the key presses being entered, and allows them to pass through to the editable text sprite if they are valid characters:

```
-- Lingo syntax
on keyDown me
    legalCharacters = "1234567890"
    if legalCharacters contains the key then
        pass
    else
        beep
    end if
end

// JavaScript syntax
function keyDown() {
    legalCharacters = "1234567890";
    if (legalCharacters.indexOf(_key.key) >= 0) {
        pass();
    } else {
        _sound.beep();
    }
}
```

See also

`stopEvent()`

pasteClipBoardInto()

Usage

```
-- Lingo syntax
memberObjRef.pasteClipBoardInto()

// JavaScript syntax
memberObjRef.pasteClipBoardInto();
```

Description

Member method; pastes the contents of the Clipboard into a specified cast member, and erases the existing cast member.

Any item that is in a format that Director can use as a cast member can be pasted.

When copying a string from another application, the string's formatting is not retained.

This method provides a convenient way to copy objects from other movies and from other applications into the Cast window. Because copied cast members must be stored in RAM, avoid using this command during playback in low memory situations.

When using this method in Shockwave Player, or in the authoring environment and projectors with the `safePlayer` property set to `TRUE`, a warning dialog will allow the user to cancel the paste operation.

Parameters

None.

Example

This statement pastes the Clipboard contents into the bitmap cast member Shrine:

```
-- Lingo syntax
member("shrine").pasteClipBoardInto()

// JavaScript syntax
member("shrine").pasteClipBoardInto();
```

See also

Member, safePlayer

pause() (DVD)

Usage

```
-- Lingo syntax
dvdObjRef.pause()

// JavaScript syntax
dvdObjRef.pause();
```

Description

DVD method; pauses playback.

Parameters

None.

Example

This statement pauses playback:

```
-- Lingo syntax
member(1).pause()

// JavaScript syntax
member(1).pause();
```

See also

DVD

pause() (Sound Channel)

Usage

```
-- Lingo syntax
soundChannelObjRef.pause()
```

```
// JavaScript syntax
soundChannelObjRef.pause();
```

Description

Sound Channel method; suspends playback of the current sound in a sound channel.

A subsequent `play()` method will resume playback.

Parameters

None.

Example

This statement pauses playback of the sound cast member playing in sound channel 1:

```
-- Lingo syntax
sound(1).pause()

// JavaScript syntax
sound(1).pause();
```

See also

`breakLoop()`, `play()` (Sound Channel), `playNext()` (Sound Channel), `queue()`, `rewind()` (Sound Channel), Sound Channel, `stop()` (Sound Channel)

pause() (3D)

Usage

```
member(whichCastmember).model(whichModel).bonesPlayer.pause()
member(whichCastmember).model(whichModel).keyframePlayer.pause()
```

Description

3D `#keyframePlayer` and `#bonesPlayer` modifier command; halts the motion currently being executed by the model. Use the `play()` command to unpause the motion.

When a model's motion has been paused by using this command, the model's `bonesPlayer.playing` property will be set to `FALSE`.

Parameters

None.

Example

This statement pauses the current animation of the model named Ant3:

```
member("PicnicScene").model("Ant3").bonesplayer.pause()
```

See also

`play()` (3D), `playing` (3D), `playlist`

pause() (RealMedia, SWA, Windows Media)

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.pause()

// JavaScript syntax
memberOrSpriteObjRef.pause();
```

Description

RealMedia and Windows Media sprite or cast member method; pauses playback of the media stream.

The `mediaStatus` value becomes `#paused`.

Calling this method while the RealMedia or Windows Media stream is playing does not change the `currentTime` property and does not clear the media buffer; this allows subsequent `play` commands to resume playback without rebuffering the stream.

Parameters

None.

Example

The following examples pause the playing of sprite 2 or the cast member Real.

```
-- Lingo syntax
sprite(2).pause()
member("Real").pause()

// JavaScript syntax
sprite(2).pause();
member("Real").pause();
```

See also

mediaStatus (RealMedia, Windows Media), play() (RealMedia, SWA, Windows Media), seek(), stop() (RealMedia, SWA, Windows Media)

perlinNoise()

Usage

```
-- Lingo syntax
member(membername).Image.perlinNoise(baseX, baseY, numOctaves, seed, bStitch,
bFractalNoise, [#channelOptions:value, #grayscale:bolean, #offsets:listOfPoints])
// JavaScript syntax
```

Description

The Perlin noise generation algorithm interpolates and combines individual random noise functions (called octaves) into a single function that generates more natural-seeming random noise.

Parameters

Property	Description	Value Range	Default
baseX: Number	Determines the x (size) value of patterns created.		
baseY: Number	Determines the y (size) value of the patterns created.		
numOctaves: number	Number of octaves or individual noise functions to combine to create this noise. Larger numbers of octaves create images with greater detail but also require more processing time.		
randomSeed: Number	The random seed number		
bStitch: Boolean	If set to true, this method attempts to stitch (or smooth) the transition edges of the image to create seamless textures for tiling as a bitmap fill.	True/false	
bFractalNoise: Boolean	This parameter relates to the edges of the gradients being generated by the method. If set to true, the method generates fractal noise that smooths the edges of the effect. If set to false, it generates turbulence. An image with turbulence has visible discontinuities in the gradient that can make it better approximate sharper visual effects, like flames and ocean waves.	True/false	
channelOptions:Number (Optional)	It specifies to which color channel (of the bitmap) the noise pattern is applied. The number can be a combination of any of the four color channel RGBA(1, 2, 4 and 8). The default value is 7.		0
grayScale: Boolean (Optional)	If set to true, it applies the randomSeed value to the bitmap pixels, effectively washing all color out of the image. The default value is false.	0 or 1	0
Offsets: List or a point (Optional)	This can be a List of points or a point that correspond to x and y offsets for each octave. By manipulating the offset values, you can smoothly scroll the layers of the image. Each point in the offset List affects a specific octave noise function. The default value is null. If you specify a point, the same values are used for every octave noise function.		Point (0,0)

Example

The following examples pause the playing of sprite 2 or the cast member Real.

```
-- Lingo syntax
myList = [point(0,1), point(5,5)] --List of Points
member("myMember").Image.perlinNoise(300, 300, 2, 2, true, true, [#channelOptions:7,
#grayscale:false, #offsets:myList])
```

```
// JavaScript syntax
myList = list(point(0,1), point(5,5)) //List of Points
member("myMember").image.perlinNoise(300, 300, 2, 2, true,
true, propList(symbol("channelOptions"),7, symbol("grayscale"),false,
symbol("offsets"),myList));
```

perpendicularTo

Usage

```
vector1.perpendicularTo(vector2)
```

Description

3D vector command; returns a vector perpendicular to both the original vector and a second vector. This command is equivalent to the vector `crossProduct` command.

Parameters

`vector2` Required. Specifies the second vector.

Example

In this example, pos1 is a vector on the x axis and pos2 is a vector on the y axis. The value returned by `pos1.perpendicularTo(pos2)` is `vector( 0.0000, 0.0000, 1.00000e4 )`. The last two lines of the example show the vector which is perpendicular to both pos1 and pos2.

```
-- Lingo
pos1 = vector(100, 0, 0)
pos2 = vector(0, 100, 0)
put pos1.perpendicularTo(pos2)
-- vector( 0.0000, 0.0000, 1.00000e4 )

// Javascript
pos1 = vector(100, 0, 0);
pos2 = vector(0, 100, 0);
trace(pos1.perpendicularTo(pos2));
// vector( 0.0000, 0.0000, 1.00000e4 )
```

See also

`crossProduct()`, `cross`

pictureP()

Usage

```
-- Lingo syntax
pictureP(pictureValue)

// JavaScript syntax
pictureP(pictureValue);
```

Description

Function; reports whether the state of the `picture` member property for the specified cast member is `TRUE` (1) or `FALSE` (0).

Because pictureP doesn't directly check whether a picture is associated with a cast member, you must test for a picture by checking the cast member's picture member property.

Parameters

pictureValue Required. Specifies a reference to the picture of a cast member.

Example

The first statement in this example assigns the value of the picture member property for the cast member Shrine, which is a bitmap, to the variable pictureValue. The second statement checks whether Shrine is a picture by checking the value assigned to pictureValue.

```
-- Lingo syntax
pictureValue = member("Shrine").picture
put pictureP(pictureValue)

// JavaScript syntax
var pictureValue = member("Shrine").picture;
put(pictureP(pictureValue));
```

The result is 1, which is the numerical equivalent of TRUE.

play() (3D)

Usage

```
member(whichCastmember).model(whichModel).bonesPlayer.play()
member(whichCastmember).model(whichModel).keyframePlayer.play()
member(whichCastmember).model(whichModel).bonesPlayer.play(motionName {, looped, startTime,
endTime, scale, offset})
member(whichCastmember).model(whichModel).keyframePlayer.play(motionName {, looped,
startTime, endTime, scale, offset})
```

Description

3D #keyframePlayer and #bonesPlayer command; initiates or unpauses the execution of a motion.

When a model's motion has been initiated or resumed by using this command, the model's bonesPlayer.playing property will be set to TRUE.

Use play() with no parameters to resume the execution of a motion that has been paused with the pause() command.

Using the play() command to initiate a motion inserts the motion at the beginning of the modifier's playlist. If this interrupts playback of another motion, the interrupted motion remains in the playlist in the next position after the newly initiated motion. When the newly initiated motion ends (if it is non-looping) or if the playNext() command is issued, the interrupted motion will resume playback at the point where it was interrupted.

Parameters

motionName Required. Specifies the name of the motion to execute. When *motionName* is the only parameter passed to play(), the motion is executed once by the model from beginning to end at the speed set by the modifier's playRate property.

looped Optional. Specifies whether the motion plays once (FALSE) or continuously (TRUE).

startTime Optional. Measured in milliseconds from the beginning of the motion. When *looped* is TRUE, the first iteration of the loop begins at *offset* and ends at *endTime* with all subsequent repetitions of the motion beginning at *startTime* and end at *endTime*.

endTime Optional. Measured in milliseconds from the beginning of the motion. When *looped* is FALSE, the motion begins at *offset* and ends at *endTime*. When *looped* is TRUE, the first iteration of the loop begins at *offset* and ends at *endTime* with all subsequent repetitions beginning at *startTime* and end at *endTime*. Set *endTime* to -1 if you want the motion to play to the end.

playRate Optional. Specifies the actual speed of the motion's playback. *playRate* is multiplied by the model's #keyframePlayer or #bonesPlayer modifier's playRate property to determine the actual speed of the motion's playback.

offset Optional. Measured in milliseconds from the beginning of the motion. When *looped* is FALSE, the motion begins at *offset* and ends at *endTime*. When *looped* is TRUE, the first iteration of the loop begins at *offset* and ends at *endTime* with all subsequent repetitions beginning at *startTime* and end at *cropEnd*. You can alternately specify the *offset* parameter with a value of #synchronized in order to start the motion at the same relative position in its duration as the currently playing animation is through its own duration.

Example

This command causes the model named Walker to begin playback of the motion named Fall. After playing this motion, the model will resume playback of any previously playing motion.

```
sprite(1).member.model("Walker").bonesPlayer.play("Fall", 0, 0, -1, 1, 0)
```

This command causes the model named Walker to begin playback of the motion named Kick. If Walker is currently executing a motion, it is interrupted by Kick and a section of Kick will play in a continuous loop. The first iteration of the loop will begin 2000 milliseconds from the motion's beginning. All subsequent iterations of the loop will begin 1000 milliseconds from Kick's beginning and will end 5000 milliseconds from Kick's beginning. The rate of playback will be three times the playRate property of the model's bonesPlayer modifier.

```
sprite(1).member.model("Walker").bonesPlayer.play("Kick", 1, 1000, 5000, 3, 2000)
```

See also

queue() (3D), playNext() (3D), playRate (3D), playlist, pause() (3D), removeLast(), playing (3D)

play() (DVD)

Usage

```
-- Lingo syntax
dvdObjRef.play()
dvdObjRef.play(beginTitle, beginChapter, endTitle, endChapter)
dvdObjRef.play(beginTimeList, endTimeList)

// JavaScript syntax
dvdObjRef.play();
dvdObjRef.play(beginTitle, beginChapter, endTitle, endChapter);
dvdObjRef.play(beginTimeList, beginTimeList);
```

Description

DVD method; starts or resumes playback.

Without parameters, this method resumes playback if paused, or, if stopped, starts playback at the top of a disc or at the value specified by the `startTimeList` property. Playback continues until the value specified by the `stopTimeList` property, if set.

With the *`beginTitle`*, *`beginChapter`*, and *`endTitle`*, *`endChapter`* parameters, this method starts playback at a given title, chapter. Playback continues until the specified *`endTitle`*, *`endChapter`* parameters, if set.

With the *`beginTimeList`* and *`endTimeList`* parameters, this method plays from the value specified by the `beginTimeList` parameter to the value specified by the `endTimeList` parameter.

The list formats used for `beginTimeList` and `endTimeList`:

```
[#title:1, #chapter:1, #hours:0, #minutes:1, #seconds:1]
```

or

```
[#title:1, #hours:0, #minutes:1, #seconds:1]
```

This method returns 0 if successful.

Parameters

`beginTitle` Required if starting playback at a given title and chapter. A number that specifies the title that contains the chapter to play. This parameter will override the member's `startTimeList` property.

`beginChapter` Required if starting playback at a given title and chapter. A number that specifies the chapter to play. This parameter will override the member's `startTimeList` property.

`endTitle` Required if stopping playback at a given title and chapter. A number that specifies the title where playback will stop. This parameter will override the member's `stopTimeList` property.

`endChapter` Required if stopping playback at a given title and chapter. A number that specifies the chapter to play. This parameter will override the member's `stopTimeList` property.

`beginTimeList` Required if starting playback at a given start time. A property list that specifies the time at which playback starts. This parameter will override the member's `startTimeList` property.

`endTimeList` Required if starting playback at a given start time. A property list that specifies the time at which playback stops. This parameter will override the member's `stopTimeList` property.

Example

This statement resumes playback of a paused sprite:

```
-- Lingo syntax
member(12).play()

// JavaScript syntax
member(12).play();
```

These statements start playing at chapter 2 of title 1 and finish playing at chapter 4:

```
member(15).play([#title:1, #chapter:2], [#title:1, #chapter:4])
```

or

```
member(15).play(1,2,1,4)
```

These statements start playing 10 seconds into chapter 2 and finish playing at 17 seconds:

```
member(15).play([#title:2, #seconds:10], [#title:2, #seconds:17])
```

See also

DVD, startTimeList, stopTimeList

play() (Sound Channel)

Usage

```
-- Lingo syntax
soundChannelObjRef.play()
soundChannelObjRef.play(memberObjRef)
soundChannelObjRef.play(propList)

// JavaScript syntax
soundChannelObjRef.play();
soundChannelObjRef.play(memberObjRef);
soundChannelObjRef.play(propList);
```

Description

Sound Channel method; begins playing any sounds queued in a sound channel, or queues and begins playing a given cast member.

Sound cast members take some time to load into RAM before they can begin playback. It's recommended that you queue sounds with `queue()` before you want to begin playing them and then use the first form of this method. The second two forms do not take advantage of the pre-loading accomplished with the `queue()` command.

By using an optional property list, you can specify exact playback settings for a sound.

To see an example of `play()` used in a completed movie, see the Sound Control movie in the Learning/Lingo folder inside the Director application folder.

Parameters

`memberObjRef` Required if playing a specific cast member. A reference to the cast member object to queue and play.

`propList` Required if specifying playback settings for a sound. A property list that specifies the exact playback settings for the sound. These properties may be optionally set:

Property	Description
#member	The sound cast member to queue. This property must be provided; all others are optional.
#startTime	The time within the sound at which playback begins, in milliseconds. The default is the beginning of the sound. See `startTime`.
#endTime	The time within the sound at which playback ends, in milliseconds. The default is the end of the sound. See `endTime`.
#loopCount	The number of times to play a loop defined with `#loopStartTime` and `#loopEndTime`. The default is 1. See `loopCount`.
#loopStartTime	The time within the sound to begin a loop, in milliseconds. See `loopStartTime`.
#loopEndTime	The time within the sound to end a loop, in milliseconds. See `loopEndTime`.
#preloadTime	The amount of the sound to buffer before playback, in milliseconds. See `preloadTime`.

Example

This statement plays cast member introMusic in sound channel 1:

```
-- Lingo syntax
sound(1).play(member("introMusic"))

// JavaScript syntax
sound(1).play(member("introMusic"));
```

The following statement plays cast member creditsMusic in sound channel 2. Playback begins 4 seconds into the sound and ends 15 seconds into the sound. The section from 10.5 seconds to 14 seconds loops 6 times.

```
-- Lingo syntax
sound(2).play([#member:member("creditsMusic"), #startTime:4000, #endTime:15000,
#loopCount:6, #loopStartTime:10500, #loopEndTime:14000])

// JavaScript syntax
sound(2).play(propList("member",member("creditsMusic"), "startTime",4000,"endTime",15000,
"loopCount",6, "loopStartTime",10500, "loopEndTime",14000));
```

See also

endTime, loopCount, loopEndTime, loopStartTime, pause() (Sound Channel), preLoadTime, queue(), Sound Channel, startTime, stop() (Sound Channel)

play() (RealMedia, SWA, Windows Media)

Usage

```
-- Lingo syntax
windowsMediaObjRef.play()
realMediaObjRef.play()

// JavaScript syntax
windowsMediaObjRef.play();
realMediaObjRef.play();
```

Description

Windows Media or RealMedia cast member or sprite method; plays the Windows Media or RealMedia cast member or plays the sprite on the Stage.

For cast members, only audio is rendered if present in the movie. If the cast member is already playing, calling this method has no effect.

Parameters

None.

Example

The following examples start the streaming process for the stream in sprite 2 and the cast member Real.

```
-- Lingo syntax
sprite(2).play()
member("Real").play()

// JavaScript syntax
sprite(2).play();
member("Real").play();
```

See also

RealMedia, Windows Media

playFile()

Usage

```
-- Lingo syntax
soundChannelObjRef.playFile(stringFilePath)

// JavaScript syntax
soundChannelObjRef.playFile(stringFilePath);
```

Description

Sound Channel method; plays the AIFF, SWA, AU, or WAV sound in a sound channel.

For the sound to be played properly, the correct MIX Xtra must be available to the movie, usually in the Xtras folder of the application.

When the sound file is in a different folder than the movie, *stringFilePath* must specify the full path to the file.

To play sounds obtained from a URL, it's usually a good idea to use `downloadNetThing()` or `preloadNetThing()` to download the file to a local disk first. This approach can minimize problems that may occur while the file is downloading.

The `playFile()` method streams files from disk rather than playing them from RAM. As a result, using `playFile()` when playing digital video or when loading cast members into memory can cause conflicts when the computer tries to read the disk in two places at once.

Parameters

stringFilePath Required. A string that specifies the name of the file to play. When the sound file is in a different folder than the currently playing movie, *stringFilePath* must also specify the full path to the file.

Example

This statement plays the file named Thunder in channel 1:

```
-- Lingo syntax
sound(1).playFile("Thunder.wav")

// JavaScript syntax
sound(1).playFile("Thunder.wav");
```

This statement plays the file named Thunder in channel 3:

```
-- Lingo syntax
sound(3).playFile(_movie.path & "Thunder.wav")

// JavaScript syntax
sound(3).playFile(_movie.path + "Thunder.wav");
```

See also

`play() (Sound Channel)`, `Sound Channel`, `stop() (Sound Channel)`

playFromToTime()

Usage

```
-- Lingo syntax
windowsMediaObjRef.playFromToTime(intStartTime, intEndTime)
```

```
// JavaScript syntax
windowsMediaObjRef.playFromToTime(intStartTime, intEndTime);
```

Description

Windows Media sprite method. Starts playback from a specified start time and plays to a specified end time.

Parameters

intStartTime Required. An integer that specifies the time, in milliseconds, at which playback begins.

intEndTime Required. An integer that specifies the time, in milliseconds, at which playback ends.

Example

This statement specifies that the sprite named Video should play from the 30 second mark to the 40 second mark.

```
-- Lingo syntax
sprite("Video").playFromToTime(30000, 40000)

// JavaScript syntax
sprite("Video").playFromToTime(30000, 40000);
```

See also

Windows Media

playNext() (Sound Channel)

Usage

```
-- Lingo syntax
soundChannelObjRef.playNext()

// JavaScript syntax
soundChannelObjRef.playNext();
```

Description

Sound Channel method; immediately interrupts playback of the current sound playing in a sound channel and begins playing the next queued sound.

If no more sounds are queued in the given channel, the sound simply stops playing.

Parameters

None.

Example

This statement plays the next queued sound in sound channel 2:

```
-- Lingo syntax
sound(2).playNext()

// JavaScript syntax
sound(2).playNext();
```

See also

pause() (Sound Channel), play() (Sound Channel), Sound Channel,stop() (Sound Channel)

playNext() (3D)

Usage

```
member(whichMember).model(whichModel).bonesPlayer.playNext()
member(whichMember).model(whichModel).keyframePlayer.playNext()
```

Description

3D `#keyframePlayer` and `#bonesPlayer` modifier command; initiates playback of the next motion in the playlist of the model's `#keyframePlayer` or `#bonesPlayer` modifier. The currently playing motion, which is the first entry in the playlist, is interrupted and removed from the playlist.

If motion blending is enabled, and there are two or more motions in the playlist, blending between the current motion and the next one in the playlist will begin when `playNext()` is called.

Example

This statement interrupts the motion currently being executed by model 1 and initiates playback of the next motion in the playlist:

```
-- Lingo
member("scene").model[1].bonesPlayer.playnext()

// Javascript
member("scene").getProp("model",1).bonesPlayer.playNext();
```

See also

blend (3D), playlist

playerParentalLevel()

Usage

```
-- Lingo syntax
dvdObjRef.playerParentalLevel()

// JavaScript syntax
dvdObjRef.playerParentalLevel();
```

Description

DVD method; returns the parental level of the player.

Possible parental levels range from 1 to 8.

Parameters

None.

See also

DVD

point()

Usage

```
-- Lingo syntax
point(intH, intV)

// JavaScript syntax
point(intH, intV);
```

Description

Top level function and data type. Returns a point that has specified horizontal and vertical coordinates.

A point has both a `locH` and a `locV` property.

Point coordinates can be changed by arithmetic operations using Lingo only. For example, the following two points can be added together using Lingo, but `NaN` is returned using JavaScript syntax:

```
-- Lingo
pointA = point(10,10)
pointB = point(5,5)
put(pointA + pointB)
-- point(15,15)

// JavaScript syntax
var pointA = point(10,10);
var pointB = point(5,5);
trace(pointA + pointB);
// NaN
```

To see an example of `point()` used in a completed movie, see the Imaging and Vector Shapes movies in the Learning/Lingo folder inside the Director application folder.

Parameters

intH Required. An integer that specifies the horizontal coordinate of the point.

intV Required. An integer that specifies the vertical coordinate of the point.

Example

This statement sets the variable `lastLocation` to the point (250, 400):

```
-- Lingo syntax
lastLocation = point(250, 400)

// JavaScript syntax
var lastLocation = point(250, 400);
```

This statement adds 5 pixels to the horizontal coordinate of the point assigned to the variable `myPoint`:

```
-- Lingo syntax
myPoint.locH = myPoint.locH + 5

// JavaScript syntax
myPoint.locH = myPoint.locH + 5;
```

In Lingo only, the following statements set a sprite's Stage coordinates to `mouseH` and `mouseV` plus 10 pixels. The two statements are equivalent.

```
-- Lingo syntax
sprite(_mouse.clickOn).loc = point(_mouse.mouseH, _mouse.mouseV) + point(10, 10)
sprite(_mouse.clickOn).loc = _mouse.mouseLoc + 10
```

See also

locH, locV

pointAt

Usage

```
member(whichCastmember).model(whichModel).pointAt(vectorPosition{, vectorUp})
member(whichCastmember).camera(whichCamera).pointAt(vectorPosition{, vectorUp})
member(whichCastmember).light(whichLight).pointAt(vectorPosition{, vectorUp})
member(whichCastmember).group(whichGroup).pointAt(vectorPosition{, vectorUp})
```

Description

3D command; rotates the referenced object so that its forward direction vector points at a specified world relative position, then it rotates the referenced object to point it's up direction vector in the direction hinted at by a specified relative vector.

The object's front and up direction vectors are defined by the object's `pointAtOrientation` property.

Parameters

`vectorPosition` Required. Specifies the world relative position. This value can also be a node reference.

`vectorUp` Optional. Specfies a world relative vector that hints at where the object's up vector should point. If this parameter isn't specified, then `pointAt` defaults to using the world's y axis as the up hinting vector. If you attempt to point the object at a position such that the object's forward vector is parallel to the world's y axis, then the world's x axis is used as the up hinting vector. The direction at which you wish to point the object's forward direction and the direction specified by *`vectorUp`* do not need to be perpendicular to each other being as this command only uses the *`vectorUp`* parameter as a hinting vector.

Example

This example points three objects at the model named Mars: the camera named MarsCam, the light named BrightSpot, and the model named BigGun:

```
thisWorldPosn = member("Scene").model("Mars").worldPosition
member("Scene").camera("MarsCam").pointAt(thisWorldPosn)
member("Scene").light("BrightSpot").pointAt(thisWorldPosn)
member("Scene").model("BigGun").pointAt(thisWorldPosn, vector(0,0,45))
```

If you use non-uniform scaling and a custom `pointAtOrientation` on the same node, e.g., a model, using `pointAt` will likely cause unexpected non-uniform scaling. This is due to the order in which the non-uniform scaling and the rotation to properly orient the node are applied. To workaround this issue, do one of the following:

- Avoid using non-uniform scaling and non-default pointAtOrientation together on the same node.
- Remove your scale prior to using pointAt, and then reapply it afterwards.

  For example:

  ```
  scale = node.transform.scale
  node.scale = vector( 1, 1, 1 )
  node.pointAt(vector(0, 0, 0)) -- non-default pointAtOrientation
  node.transform.scale = scale
  ```

See also

pointAtOrientation

pointInHyperlink()

Usage

```
-- Lingo syntax
spriteObjRef.pointInHyperlink(point)

// JavaScript syntax
spriteObjRef.pointInHyperlink(point);
```

Description

Text sprite function; returns a value (`TRUE` or `FALSE`) that indicates whether the specified point is within a hyperlink in the text sprite. Typically, the point used is the cursor position. This is useful for setting custom cursors.

Parameters

`point` Required. Specifies the point to test.

Example

This statement checks whether the mouse is moving over a hyperlink in sprite(1).

```
-- Lingo
Put Sprite(1).pointInHyperLink(_mouse.mouseLoc)

// Javascript
trace(sprite(1).pointInHyperLink(_mouse.mouseLoc));
```

See also

`cursor()`, `mouseLoc`

pointToChar()

Usage

```
-- Lingo syntax
spriteObjRef.pointToChar(pointToTranslate)

// JavaScript syntax
spriteObjRef.pointToChar(pointToTranslate);
```

Description

Function; returns an integer representing the character position located within the text or field sprite at a specified screen coordinate, or returns -1 if the point is not within the text.

This function can be used to determine the character under the cursor.

Parameters

`pointToTranslate` Required. Specifies the screen coordinate to test.

Example

These statements display the number of the character being clicked, as well as the letter, in the Message window:

```
--Lingo syntax
property spriteNum

on mouseDown me
```

```
    pointClicked = _mouse.mouseLoc
    currentMember = sprite(spriteNum).member
    charNum = sprite(spriteNum).pointToChar(pointClicked)
    actualChar = currentMember.char[charNum]
    put("Clicked character" && charNum & ", the letter" && actualChar)
end

// JavaScript syntax
function mouseDown() {
    var pointClicked = _mouse.mouseLoc;
    var currentMember = sprite(this.spriteNum).member;
    var charNum = sprite(this.spriteNum).pointToChar(pointClicked);
    var actualChar = currentMember.getProp("char", charNum);
    put("Clicked character " + charNum +", the letter " + actualChar);
}
```

See also

mouseLoc, pointToWord(), pointToItem(), pointToLine(), pointToParagraph()

pointToItem()

Usage

```
-- Lingo syntax
spriteObjRef.pointToItem(pointToTranslate)

// JavaScript syntax
spriteObjRef.pointToItem(pointToTranslate);
```

Description

Function; returns an integer representing the item position in the text or field sprite at a specified screen coordinate, or returns -1 if the point is not within the text. Items are separated by the `itemDelimiter` property, which is set to a comma by default.

This function can be used to determine the item under the cursor.

Parameters

`pointToTranslate` Required. Specifies the screen coordinate to test.

Example

These statements display the number of the item being clicked, as well as the text of the item, in the Message window:

```
--Lingo syntax
property spriteNum

on mouseDown me
    pointClicked = _mouse.mouseLoc
    currentMember = sprite(spriteNum).member
    itemNum = sprite(spriteNum).pointToItem(pointClicked)
    itemText = currentMember.item[itemNum]
    put("Clicked item" && itemNum & ", the text" && itemText)
end

// JavaScript syntax
function mouseDown() {
    var pointClicked = _mouse.mouseLoc;
```

```
    var currentMember = sprite(this.spriteNum).member;
    var itemNum = sprite(this.spriteNum).pointToItem(pointClicked);
    var itemText = currentMember.getProp("item",itemNum);
    trace( "Clicked item " + itemNum + ", the text " + itemText);
}
```

See also

itemDelimiter, mouseLoc, pointToChar(), pointToWord(), pointToItem(), pointToLine(), pointToParagraph()

pointToLine()

Usage

```
-- Lingo syntax
spriteObjRef.pointToLine(pointToTranslate)

// JavaScript syntax
spriteObjRef.pointToLine(pointToTranslate);
```

Description

Function; returns an integer representing the line position in the text or field sprite at a specified screen coordinate, or returns -1 if the point is not within the text. Lines are separated by carriage returns in the text or field cast member.

This function can be used to determine the line under the cursor.

Parameters

pointToTranslate Required. Specifies the screen coordinate to test.

Example

These statements display the number of the line being clicked, as well as the text of the line, in the Message window:

```
-- Lingo syntax
property spriteNum

on mouseDown me
    pointClicked = _mouse.mouseLoc
    currentMember = sprite(spriteNum).member
    lineNum = sprite(spriteNum).pointToLine(pointClicked)
    lineText = currentMember.line[lineNum]
    put("Clicked line" && lineNum & ", the text" && lineText)
end

// JavaScript syntax
functionmouseDown() {
    var pointClicked = _mouse.mouseLoc;
    var currentMember = sprite(this.spriteNum).member;
    var lineNum = sprite(this.spriteNum).pointToLine(pointClicked);
    var lineText = currentMember.getProp("line", lineNum);
    put("Clicked line " + lineNum + ", the text " + lineText);
}
```

See also

itemDelimiter, mouseLoc, pointToChar(), pointToWord(), pointToItem(), pointToLine(), pointToParagraph()

pointToParagraph()

Usage

```
-- Lingo syntax
spriteObjRef.pointToParagraph(pointToTranslate)

// JavaScript syntax
spriteObjRef.pointToParagraph(pointToTranslate);
```

Description

Function; returns an integer representing the paragraph number located within the text or field sprite at a specified at screen coordinate, or returns -1 if the point is not within the text. Paragraphs are separated by carriage returns in a block of text.

This function can be used to determine the paragraph under the cursor.

Parameters

pointToTranslate Required. Specifies the screen coordinate to test.

Example

These statements display the number of the paragraph being clicked, as well as the text of the paragraph, in the message window:

```
-- Lingo syntax
property spriteNum

on mouseDown me
    pointClicked = _mouse.mouseLoc
    currentMember = sprite(spriteNum).member
    paragraphNum = sprite(spriteNum).pointToParagraph(pointClicked)
    paragraphText = currentMember.paragraph[paragraphNum]
    put("Clicked paragraph" && paragraphNum & ", the text" && paragraphText)
end

// JavaScript syntax
function mouseDown() {
    var pointClicked = _mouse.mouseLoc;
    var currentMember = sprite(this.spriteNum).member;
    var paragraphNum = sprite(this.spriteNum).pointToParagraph(pointClicked);
    var paragraphText = currentMember.getProp("paragraph", paragraphNum);
    trace("Clicked paragraph " + paragraphNum + ", the text " + paragraphText);
}
```

See also

itemDelimiter, mouseLoc, pointToChar(), pointToWord(), pointToItem(), pointToLine()

pointToWord()

Usage

```
-- Lingo syntax
spriteObjRef.pointToWord(pointToTranslate)

// JavaScript syntax
spriteObjRef.pointToWord(pointToTranslate);
```

Description

Function; returns an integer representing the number of a word located within the text or field sprite at a specified screen coordinate, or returns -1 if the point is not within the text. Words are separated by spaces in a block of text.

This function can be used to determine the word under the cursor.

Parameters

pointToTranslate Required. Specifies the screen coordinate to test.

Example

These statements display the number of the word being clicked, as well as the text of the word, in the Message window:

```
-- Lingo syntax
property spriteNum

on mouseDown me
    pointClicked = _mouse.mouseLoc
    currentMember = sprite(spriteNum).member
    wordNum = sprite(spriteNum).pointToWord(pointClicked)
    wordText = currentMember.word[wordNum]
    put("Clicked word" && wordNum & ", the text" && wordText)
end

// JavaScript syntax
function mouseDown(me) {
    var pointClicked = _mouse.mouseLoc;
    var currentMember = sprite(this.spriteNum).member;
    var wordNum = sprite(this.spriteNum).pointToWord(pointClicked);
    var wordText = currentMember.getProp("word", wordNum);
    trace("Clicked word " + wordNum + ", the text " + wordText);
}
```

See also

itemDelimiter, mouseLoc, pointToChar(), pointToItem(), pointToLine(), pointToParagraph()

postNetText

Usage

```
postNetText(url, propertyList {,serverOSString} {,serverCharSetString})
postNetText(url, postText {,serverOSString} {,serverCharSetString})
```

Description

Command; sends a POST request to a URL, which is an HTTP URL, with specified data.

This command is similar to getNetText(). As with getNetText(), the server's response is returned by netTextResult(*netID*) once netDone(*netID*) becomes 1, and if netError(*netID*) is 0, or okay.

The optional parameters may be omitted without regard to position.

This command also has an additional advantage over getNetText(): a postNetText() query can be arbitrarily long, whereas the getNetText() query is limited to the length of a URL (1K or 4K, depending on the browser).

Note: *If you use postNetText to post data to a domain different from the one the movie is playing from, the movie will display a security alert when playing back in Shockwave Player.*

To see an example of `postNetText` used in a completed movie, see the Forms and Post movie in the Learning/Lingo folder inside the Director application folder.

Parameters

`url` Required. Specifies the URL to send the `POST` request to.

`propertyList` or *`postText`* Required. Specifies the data to send with the request. When a property list is used instead of a string, the information is sent in the same way a browser posts an HTML form, with `METHOD=POST`. This facilitates the construction and posting of form data within a Director title. Property names correspond to HTML form field names and property values to field values.

The property list can use either strings or symbols as the property names. If a symbol is used, it is automatically converted to a string without the # at the beginning. Similarly, a numeric value is converted to a string when used as the value of a property.

Note: *If a program uses the alternate form—a string instead of property list—the string postText is sent to the server as an HTTP POST request using MIME type "text/plain." This will be convenient for some applications, but is not compatible with HTML forms posting. PHP scripts, for example, should always use a property list.*

`serverOSString` Optional. Defaults to `UNIX` but may be set to `Windows` or `Mac` and translates any carriage returns in the *`postText`* argument into those used on the server to avoid confusion. For most applications, this setting is unnecessary because line breaks are usually not used in form responses.

`serverCharSetString` Optional. Applies only if the user is running on a Shift-JIS (Japanese) system. Its possible settings are `"JIS"`, `"EUC"`, `"ASCII"`, and `"AUTO"`. Posted data is converted from Shift-JIS to the named character set. Returned data is handled exactly as by `getNetText()` (converted from the named character set to Shift-JIS). If you use `"AUTO"`, the posted data from the local character set is not translated; the results sent back by the server are translated as they are for `getNetText()`. `"ASCII"` is the default if *`serverCharSetString`* is omitted. `"ASCII"` provides no translation for posting or results.

Example

This statement omits the *`serverCharSetString`* parameter:

```
-- Lingo
netID = postNetText("www.mydomain.com\database.cgi", "Bill Jones", "Win")

// Javascript
netID = postNetText("www.mydomain.com\database.cgi", "Bill Jones", "Win");
```

This example generates a form from user-entry fields for first and last name, along with a Score. Both `serverOSString` and `serverCharSetString` have been omitted:

```
-- LingonetID = postNetText("www.mydomain.com/userbase.cgi", infoList);
lastName = member("Last Name").text
firstName = member("First Name").text
totalScore = member("Current Score").text
infoList = ["FName":firstName, "LName":lastName, "Score":totalScore]
netID = postNetText("www.mydomain.com/userbase.cgi", infoList);

// Javascript
lastName = member("Last Name").text;
firstName = member("First Name").text;
totalScore = member("Current Score").text;
infoList = propList("FName",firstName, "LName",lastName, "Score",totalScore);
netID = postNetText("www.mydomain.com/userbase.cgi", infoList);
```

See also

`getNetText()`, `netTextResult()`, `netDone()`, `netError()`

power()

Usage

```
power(base, exponent)
```

Description

Math function; calculates the value of a specified number to a specified exponent.

Parameters

`base` Required. Specifies the base number.

`exponent` Required. Specifies the exponent value.

Example

This statement sets the variable `vResult` to the value of 4 to the third power:

```
-- Lingo
set vResult = power(4,3)

// Javascript
Var vResult = Math.pow(4,3);
```

preLoad() (Member)

Usage

```
-- Lingo syntax
memberObjRef.preLoad({toMemberObjRef})

// JavaScript syntax
memberObjRef.preLoad({toMemberObjRef});
```

Description

Member method; preloads a cast member or a range of cast members into memory, and stops preloading when memory is full or when all specified cast members have been preloaded.

When used without the *`toMemberObjRef`* parameter, `preLoad()` preloads all cast members used from the current frame to the last frame of a movie.

Parameters

`toMemberObjRef` Optional. A reference to the last cast member in a range of cast members that is loaded into memory. The first cast member in the range is specified by *`memberObjRef`*.

Example

This statement reports in the Message window whether the QuickTime movie Rotating Chair can be preloaded into memory:

```
-- Lingo syntax
put(member("Rotating Chair").preload())
```

```
// JavaScript syntax
put(member("Rotating Chair").preload());
```

This `startMovie` handler sets up a Flash movie cast member for streaming and then sets its `bufferSize` property:

```
-- Lingo syntax
on startMovie
    member("Flash Demo").preload = FALSE
    member("Flash Demo").bufferSize = 65536
end

// JavaScript syntax
function startMovie() {
    member("Flash Demo").preload = false;
    member("Flash Demo").bufferSize = 65536;
}
```

See also

`Member`

preLoad() (Movie)

Usage

```
-- Lingo syntax
_movie.preLoad({frameNameOrNum})
_movie.preLoad(fromFrameNameOrNum, toFrameNameOrNum)

// JavaScript syntax
_movie.preLoad({frameNameorNum});
_movie.preLoad(fromFrameNameOrNum, toFrameNameOrNum);
```

Description

Movie method; preloads cast members in the specified frame or range of frames into memory and stops when memory is full or when all of the specified cast members have been preloaded, as follows:

- When used without arguments, this method preloads all cast members used from the current frame to the last frame of a movie.
- When used with one argument, *`frameNameOrNum`*, this method preloads all cast members used in the range of frames from the current frame to the frame *`frameNameOrNum`*, as specified by the frame number or label name.
- When used with two arguments, *`fromFrameNameOrNum`* and *`toFrameNameOrNum`*, preloads all cast members used in the range of frames from the frame *`fromFrameNameOrNum`* to the frame *`toFrameNameOrNum`*, as specified by the frame number or label name.

The `preLoad()` method also returns the number of the last frame successfully loaded. To obtain this value, use the `result()` method.

Parameters

`frameNameOrNum` Optional. A string that specifies the specific frame to preload, or an integer that specifies the number of the specific frame to preload.

`fromFrameNameOrNum` Required if preloading a range of frames. A string that specifies the name of the label of the first frame in the range of frames to preload, or an integer that specifies the number of the first frame in the range of frames to preload.

toFrameNameOrNum Required if preloading a range of frames. A string that specifies the name of the label of the last frame in the range of frames to preload, or an integer that specifies the number of the last frame in the range of frames to preload.

Example

This statement preloads the cast members used from the current frame to the frame that has the next marker:

```
-- Lingo syntax
_movie.preLoad(_movie.marker(1))

// JavaScript syntax
_movie.preLoad(_movie.marker(1));
```

This statement preloads the cast members used from frame 10 to frame 50:

```
-- Lingo syntax
_movie.preLoad(10, 50)

// JavaScript syntax
_movie.preLoad(10, 50);
```

See also

Movie, result

preLoadBuffer()

Usage

```
-- Lingo syntax
memberObjRef.preLoadBuffer()

// JavaScript syntax
memberObjRef.preLoadBuffer();
```

Description

Command; preloads part of a specified Shockwave Audio (SWA) file into memory. The amount preloaded is determined by the `preLoadTime` property. This command works only if the SWA cast member is stopped.

When the `preLoadBuffer` command succeeds, the `state` member property equals 2.

Most SWA cast member properties can be tested only after the `preLoadBuffer` command has completed successfully. These properties include: `cuePointNames`, `cuePointTimes`, `currentTime`, `duration`, `percentPlayed`, `percentStreamed`, `bitRate`, `sampleRate`, and `numChannels`.

Parameters

None.

Example

This statement loads the cast member Mel Torme into memory:

```
-- Lingo syntax
member("Mel Torme").preLoadBuffer()

// JavaScript syntax
member("Mel Torme").preLoadBuffer();
```

See also

preLoadTime

preLoadMember()

Usage

```
-- Lingo syntax
_movie.preLoadMember({memberObjRef})
_movie.preLoadMember(fromMemNameOrNum, toMemNameOrNum)

// JavaScript syntax
_movie.preLoadMember({memberObjRef});
_movie.preLoadMember(fromMemNameOrNum, toMemNameOrNum);
```

Description

Movie method; preloads cast members and stops when memory is full or when all of the specified cast members have been preloaded.

This method returns the cast member number of the last cast member successfully loaded. To obtain this value, use the `result()` method.

When used without arguments, `preLoadMember()` preloads all cast members in the movie.

When used with the *memberObjRef* argument, `preLoadMember()` preloads just that cast member. If *memberObjRef* is an integer, only the first cast library is referenced. If *memberObjRef* is a string, the first member with the string as its name will be used.

When used with the arguments *fromMemNameOrNum* and *toMemNameOrNum*, `preLoadMember()` preloads all cast members in the range specified by the cast member numbers or names.

Parameters

memberObjRef Optional. A reference to the cast member to preload.

fromMenNameOrNum Required when preloading a range of cast members. A string or an integer that specifies the first cast member in the range of cast members to preload.

toMemNameOrNum Required when preloading a range of cast members. A string or an integer that specifies the first cast member in the range of cast members to preload.

Example

This statement preloads the member " SWF " in the movie.

```
-- Lingo
_movie.preLoadMember(member("SWF"))

// Javascript
_movie.preLoadMember(member("SWF")) ;
```

See also

Movie, preLoad() (Member), result

preLoadMovie()

Usage

```
-- Lingo syntax
_movie.preLoadMovie(stringMovieName)

// JavaScript syntax
_movie.preLoadMovie(stringMovieName);
```

Description

Movie method; preloads the data and cast members associated with the first frame of the specified movie. Preloading a movie helps it start faster when it is started by the `go()` or `play()` methods.

To preload cast members from a URL, use `preloadNetThing()` to load the cast members directly into the cache, or use `downloadNetThing()` to load a movie on a local disk from which you can load the movie into memory and minimize downloading time.

Parameters

`stringMovieName` Required. A string that specifies the name of the movie to preload.

Example

This statement preloads the movie Introduction, which is located in the same folder as the current movie:

```
-- Lingo syntax
_movie.preLoadMovie("Introduction")

// JavaScript syntax
_movie.preLoadMovie("Introduction");
```

See also

`downloadNetThing`, `go()`, `Movie`, `preloadNetThing()`

preloadNetThing()

Usage

```
preloadNetThing (url)
```

Description

Function; preloads a file from the Internet to the local cache so it can be used later without a download delay. The return value is a network ID that you can use to monitor the progress of the operation.

The `preloadNetThing()` function downloads the file while the current movie continues playing. Use `netDone()` to find out whether downloading is finished.

After an item is downloaded, it can be displayed immediately because it is taken from the local cache rather than from the network.

Although many network operations can be active at a time, running more than four concurrent operations usually slows down performance unacceptably.

Neither the cache size nor the Check Documents option in a browser's preferences affects the behavior of the `preloadNetThing` function.

The `preloadNetThing()` function does not parse a Director file's links. Thus, even if a Director file is linked to casts and graphic files, `preloadNetThing()` downloads only the Director file. You still must preload other linked objects separately.

Parameters

`url` Required. Specifies the name of any valid Internet file, such as a Director movie, graphic, or FTP server location.

Example

This statement uses `preloadNetThing()` and returns the network ID for the operation:

```
-- Lingo
set mynetid = preloadNetThing("http://www.yourserver.com/menupage/mymovie.dir")

// Javascript
set mynetid = preloadNetThing("http://www.yourserver.com/menupage/mymovie.dir");
```

After downloading is complete, you can navigate to the movie using the same URL. The movie will be played from the cache instead of the URL, since it's been loaded in the cache.

See also

`netDone()`

preMultiply

Usage

```
transform1.preMultiply(transform2)
```

Description

3D transform command; alters a transform by pre-applying the positional, rotational, and scaling effects of another transform.

If *`transform2`* describes a rotation of 90° about the X axis and *`transform1`* describes a translation of 100 units in the Y axis, `transform1.multiply(`*`transform2`*`)` will alter this transform so that it describes a translation followed by a rotation. The statement `transform1.preMultiply(`*`transform2`*`)` will alter this transform so that it describes a rotation followed by a translation. The effect is that the order of operations is reversed.

Parameters

`transform2` Required. Specifies the transform from which effects are pre-applied to another transform.

Example

This statement performs a calculation that applies the transform of the model Mars to the transform of the model Pluto:

```
-- Lingo
member("scene").model("Pluto").transform.preMultiply(member("scene").model("Mars").transfo
rm)

// Javascript
member("scene").getPropRef("model" ,
i).transform.preMultiply(member("scene").getPropRef("model",j).transform) ;
// where i and j are the number index of the models "Pluto" and "Mars" respectively.
```

preRotate

Usage

```
transformReference.preRotate( xAngle, yAngle, zAngle )
transformReference.preRotate( vector )
transformReference.preRotate( positionVector, directionVector, angle )
member( whichCastmember ).node.transform.preRotate( xAngle, yAngle, zAngle )
member( whichCastmember ).node.transform.preRotate( vector )
member( whichCastmember ).node.transform.preRotate( positionVector, directionVector, angle
)
```

Description

3D transform command; applies a rotation before the current positional, rotational, and scale offsets held by the referenced transform object. The rotation may be specified as a set of three angles, each of which specify an angle of rotation about the three corresponding axes. These angles may be specified explicitly in the form of `xAngle`, `yAngle`, and `zAngle`, or by a vector, where the x component of the vector corresponds to the rotation about the *x*-axis, the y about the *y*-axis, and the z about the *z*-axis.

Alternatively, the rotation may also be specified as a rotation about an arbitrary axis. This axis is defined in space by *positionVector* and *directionVector*. The amount of rotation about this axis is specified by *angle*.

Node may be a reference to a model, group, light, or camera

Parameters

xAngle Required if applying a rotation using *x*-, *y*-, and *z*-axes. Specifies the angle of rotation around the *x*-axis.

yAngle Required if applying a rotation using *x*-, *y*-, and *z*-axes. Specifies the angle of rotation around the *y*-axis.

zAngle Required if applying a rotation using *x*-, *y*-, and *z*-axes. Specifies the angle of rotation around the *z*-axis.

vector Required if applying a rotation using a vector. Specifies the vector whose angles are used in the rotation.

positionVector Required if applying a rotation about an arbitrary axis. Specifies the position offset.

directionVector Required if applying a rotation about an arbitrary axis. Specifies the direction offset.

angle Required if applying a rotation about an arbitrary axis. Specifies the amount of rotation about an arbitrary axis.

Example

The following statement performs a rotation of 20° about each axis. Since the model's `transform` property is its position, rotation, and scale offsets relative to that model's parent, and `preRotate` applies the change in orientation prior to any existing effects of that model's `transform`, this will rotate the model in place rather than orbiting around its parent.

```
-- Lingo
member("scene").model("bip01").transform.preRotate(20, 20, 20)

// Javascript
member("scene").getPropRef("model", i).transform.preRotate(20, 20, 20) ;
// where i is the number index of the model "bip01"
```

The above is equivalent to:

```
member("scene").model("bip01").rotate(20,20,20).
// javascript
member("scene").getPropRef("model", i).rotate(20,20,20) ;
// where i is the number index of the model "bip01"
```

Generally `preRotate()` is only useful when dealing with transform variables. This line will orbit the camera about the point (100, 0, 0) in space, around the y axis, by 180°.

```
-- Lingo
t = transform()
t.position = member("scene").camera[1].transform.position
t.preRotate(vector(100, 0, 0), vector(0, 1, 0), 180)
member("scene").camera[1].transform = t

// javascript
var t = transform() ;
t.position = member("scene").getPropRef("camera", 1).transform.position ;
t.preRotate(vector(100, 0, 0), vector(0, 1, 0), 180) ;
member("scene").getPropRef("camera",1).transform = t ;
```

See also

rotate

preScale()

Usage

```
transformReference.preScale( xScale, yScale, zScale )
transformReference.preScale( vector )
member( whichCastmember ).node.transform.preScale( xScale, yScale, zScale )
member( whichCastmember ).node.transform.preScale( vector )
```

Description

3D transform command; applies a scale prior to the existing positional, rotational, and scaling effects of the given transform.

Node may be a reference to a model, group, light, or camera.

Parameters

xScale Required if applying a scale using *x*-, *y*-, and *z*-axes. Specifies the scale around the *x*-axis.

yScale Required if applying a scale using *x*-, *y*-, and *z*-axes. Specifies the scale around the *y*-axis.

zScale Required if applying a scale using *x*-, *y*-, and *z*-axes. Specifies the scale around the *z*-axis.

vector Required if applying a scale using a vector. Specifies the vector that contains the scale to apply.

Example

Line 1 of the following Lingo creates a duplicate of Moon1's transform. Remember that access to a model's transform property is by reference.

Line 2 applies a scale to that transform prior to any existing positional or rotational effects of that transform. Assume that the transform represents the positional offset and rotational orbit of Moon1 relative to its parent planet. Lets also assume Moon2's parent is the same as Moon1's. If we used `scale()` here instead of `preScale()`, then Moon2 would be pushed out twice as far and rotated about the planet twice as much as is Moon1. This is because the scaling would be applied to the transform's existing positional and rotational offsets. Using `preScale()` will apply the size change without affecting these existing positional and rotational offsets.

Line 3 applies an additional 180° rotation about the x-axis of the planet. This will put Moon2 on the opposite side of Moon1's orbit. Using `preRotate()` would have left Moon2 in the same place as Moon1, spun around its own x-axis by 180°.

Line 4 assigns this new transform to Moon2.

```
-- Lingo
t = member("scene").model("Moon1").transform.duplicate()
t.preScale(2,2,2)
t.rotate(180,0,0)
member("scene").model("Moon2").transform = t

// Javascript
var t = member("scene").getPropRef("model", i).transform.duplicate() ;
t.preScale(2,2,2) ;
t.rotate(180,0,0) ;
member("scene").getPropRef("model", i).transform = t ;
// where i the number index of model " Moon2".
```

preTranslate()

Usage

```
transformReference.preTranslate( xIncrement, yIncrement, zIncrement )
transformReference.preTranslate( vector )
member( whichCastmember ).node.transform.preTranslate(xIncrement, yIncrement, zIncrement)
member( whichCastmember ).node.transform.preTranslate( vector )

// Javascript
member( whichCastmember ).getProp("model",a).transform.preTranslate(xIncrement, yIncrement,
zIncrement) ;
```

Description

3D transform command; applies a translation before the current positional, rotational, and scale offsets held by the referenced transform object. The translation may be specified as a set of three increments along the three corresponding axes. These increments may be specified explicitly in the form of *xIncrement*, *yIncrement*, and *zIncrement*, or by a vector, where the X component of the vector corresponds to the translation about the X axis, the Y about the Y axis, and the Z about the Z axis.

After a series of transformations are done, in the following order, the model's local origin will be at (0, 0, -100), assuming the model's parent is the world:

```
model.transform.identity()
model.transform.rotate(0, 90, 0)
model.transform.preTranslate(100, 0, 0)
```

Had `translate()` been used instead of `preTranslate()`, the model's local origin would be at (100, 0, 0) and the model rotated about its own Y axis by 90°. The statement `model.transform.pretranslate(x, y, z)` is equivalent to `model.translate(x, y, z)`. Generally, `preTranslate()` is only useful when dealing with transform variables rather than `model.transform` references.

Parameters

xIncrement Required if applying a translation using *x*-, *y*-, and *z*-axes. Specifies the translation around the *x*-axis.

yIncrement Required if applying a translation using *x*-, *y*-, and *z*-axes. Specifies the translation around the *y*-axis.

zIncrement Required if applying a translation using *x*-, *y*-, and *z*-axes. Specifies the translation around the *z*-axis.

vector Required if applying a translation using a vector. Specifies the vector to use in the translation.

Example

```
-- Lingo
 t = transform()
t.transform.identity()
t.transform.rotate(0, 90, 0)
t.transform.preTranslate(100, 0, 0)
gbModel = member("scene").model("mars")
gbModel.transform = t
put gbModel.transform.position
-- vector(0.0000, 0.0000, -100.0000)

// Javascript

gbModel = member("scene").getProp("model" , a) ;
// where a is the number index for the mars model.
gbModel.transform.preTranslate(xIncrement , yIncrement, zIncrement) ;
member("scene").getProp("model" , a).transform.preTranslate(xIncrement , yIncrement,
zIncrement) ;
```

print()

Usage

```
-- Lingo syntax
spriteObjRef.print({targetName, #printingBounds})

// JavaScript syntax
spriteObjRef.print({targetName, #printingBounds});
```

Description

Command; calls the corresponding `print` ActionScript command, which was introduced in Flash 5. All frames in the Flash movie that have been labeled `#p` are printed. If no individual frames have been labeled, the whole movie prints.

Because printing of Flash movies is rather complicated, you may benefit from reviewing the section about printing in the Flash 5 documentation before using this sprite function.

Parameters

targetName Optional. Specifies the name of the target movie or movie clip to be printed. If omitted (if the target is 0), then the main Flash movie is printed.

printingBounds Optional. Specifies the options for the printing bounds. If omitted, the bounds of the target movie are used. If specified, *printingBounds* must be one of the following values:

- `#bframe`. If specified, then the printing bounds for each page are changed to match each frame that is being printed.
- `#bmax`. If specified, then the printing bounds become a large enough virtual rectangle to fit all frames to be printed.

Example

This statement prints the flash movie present as the "SWF" cast member.

```
-- Lingo
member("SWF").print()
// javascript
member("SWF").print() ;
```

printAsBitmap()

Usage

```
-- Lingo syntax
spriteObjRef.printAsBitmap({targetName, #printingBounds})

// JavaScript syntax
spriteObjRef.printAsBitmap({targetName, #printingBounds});
```

Description

Flash sprite command; functions much like the `print` command, but works only with Flash sprites. However, `printAsBitmap` can be used to print objects containing alpha channel information.

printFrom()

Usage

```
-- Lingo syntax
_movie.printFrom(startFrameNameOrNum {, endFrameNameOrNum, redux})

// JavaScript syntax
_movie.printFrom(startFrameNameOrNum {, endFrameNameOrNum, redux});
```

Description

Movie method; prints whatever is displayed on the Stage in each frame, whether or not the frame is selected, starting at the frame specified by *`startFrame`*. Optionally, you can supply *`endFrame`* and a reduction (*`redux`*) value (100%, 50%, or 25%).

The frame being printed need not be currently displayed. This command always prints at 72 dots per inch (dpi), bitmaps everything on the screen (text will not be as smooth in some cases), prints in portrait (vertical) orientation, and ignores Page Setup settings. For more flexibility when printing from within Director, see PrintOMatic Lite Xtra, which is on the installation disk.

Parameters

`startFrameNameOrNum` Required. A string or integer that specifies the name or number of the first frame to print.

`endFrameNameOrNum` Optional. A string or integer that specifies the name or number of the last frame to print.

`redux` Optional. An integer that specifies the reduction value. Valid values are 100, 50, or 25.

Example

This statement prints what is on the Stage in frame 1:

```
-- Lingo syntax
_movie.printFrom(1)

// JavaScript syntax
_movie.printFrom(1);
```

The following statement prints what is on the Stage in every frame from frame 10 to frame 25. The reduction is 50%.

```
-- Lingo syntax
_movie.printFrom(10, 25, 50)

// JavaScript syntax
_movie.printFrom(10, 25, 50);
```

See also

Movie

propList()

Usage

```
-- Lingo syntax
propList()
[:]
propList(string1, value1, string2, value2, ...)
propList(#symbol1, value1, #symbol2, value2, ...)
[#symbol1:value1, #symbol2:value2, ...]

// JavaScript syntax
propList();
propList(string1, value1, string2, value2, ...);
```

Description

Top level function; creates a property list, where each element in the list consists of a name/value pair.

When creating a property list using the syntax `propList()` or `[:]` (Lingo only), with or without parameters, the index of list values begins with 1.

The maximum length of a single line of executable script is 256 characters. Large property lists cannot be created using `propList()`. To create a property list with a large amount of data, enclose the data in square brackets (`[]`), put the data into a field, and then assign the field to a variable. The variable's content is a list of the data.

Parameters

`string1`, *`string2`*, . . . Optional. Strings that specify the name portions of the elements in the list.

`value1`, *`value2`*, . . . Optional. Values that specify the value portions of the elements in the list.

`#symbol1`, *`#symbol2`*, . . . (Lingo only) Optional. Symbols that represent the name portions of the elements in the list.

Example

This statement creates a property list with various properties and values, and then displays the various property values in the Message window:

```
-- Lingo syntax
-- using propList()
colorList = propList(#top,"red", #sides,"blue", #bottom,"green")
-- using brackets
colorList = [#top:"red", #sides:"blue", #bottom:"green"]
put(colorList.top) -- "red"
put(colorList.sides) -- "blue"
put(colorList.bottom) -- "green"
```

```
// JavaScript syntax
var colorList = propList("top","red", "sides","blue", "bottom","green");
put(colorList.top); // red
put(colorList.sides); // blue
put(colorList.bottom); // green
```

See also

`list()`

proxyServer

Usage

```
proxyServer serverType, "ipAddress", portNum
proxyServer()
```

Description

Command; sets the values of an FTP or HTTP proxy server.

Without parameters, `proxyServer()` returns the settings of an FTP or HTTP proxy server.

Parameters

`serverType` Optional. A symbol that specifies the type of proxy server. The value can be either `#ftp` or `#http`.

`ipAddress` Optional. A string that specifies the IP address.

`portNum` Optional. An integer that specifies the port number.

Example

This statement sets up an HTTP proxy server at IP address 197.65.208.157 using port 5:

```
-- Lingo
proxyServer #http,"197.65.208.157",5
// Javascript
proxyServer (symbol("http"),"197.65.208.157",5) ;
```

This statement returns the port number of an HTTP proxy server:

```
-- Lingo
put proxyServer(#http,#port)

// Javascript
put (proxyServer(symbol("http"),symbol("port"))) ;
```

If no server type is specified, the function returns 1.

This statement returns the IP address string of an HTTP proxy server:

```
-- Lingo
put proxyServer(#http)

// Javascript
put (proxyServer(symbol("http")) ;
```

This statement turns off an FTP proxy server:

```
proxyServer #ftp,#stop
// Javascript
```

```
proxyServer(symbol("ftp"),symbol("stop")) ;
```

ptToHotSpotID()

Usage

```
-- Lingo syntax
spriteObjRef.ptToHotSpotID(point)

// JavaScript syntax
spriteObjRef.ptToHotSpotID(point);
```

Description

QuickTime VR function; returns the ID of the hotspot, if any, that is at the specified point. If there is no hotspot, the function returns 0.

Parameters

point Required. Specifies the point to test.

puppetPalette()

Usage

```
-- Lingo syntax
_movie.puppetPalette(palette {, speed} {, frames})

// JavaScript syntax
_movie.puppetPalette(palette {, speed} {, frames});
```

Description

Movie method; causes the palette channel to act as a puppet and lets script override the palette setting in the palette channel of the Score and assign palettes to the movie.

The `puppetPalette()` method sets the current palette to the palette cast member specified by *palette*. If *palette* evaluates to a string, it specifies the cast library name of the palette. If *palette* evaluates to an integer, it specifies the member number of the palette.

For best results, use the `puppetPalette()` method before navigating to the frame on which the effect will occur so that Director can map to the desired palette before drawing the next frame.

You can fade in the palette by replacing *speed* with an integer from 1 (slowest) to 60 (fastest). You can also fade in the palette over several frames by replacing *frames* with an integer for the number of frames.

A puppet palette remains in effect until you turn it off using the syntax `_movie.puppetPalette(0)`. No subsequent palette changes in the Score are obeyed when the puppet palette is in effect.

Note: *The browser controls the palette for the entire Web page. Thus, Shockwave Player always uses the browser's palette.*

Parameters

palette Required. A string or integer that specifies the name or number of the new palette.

speed Optional. An integer that specifies the speed of a fade. Valid values range from 1 to 60.

frames Optional. An integer that specifies the number of frames over which a fade takes place.

Example

This statement makes Rainbow the movie's palette:

```
-- Lingo syntax
_movie.puppetPalette("Rainbow")

// JavaScript syntax
_movie.puppetPalette("Rainbow");
```

The following statement makes Rainbow the movie's palette. The transition to the Rainbow palette occurs over a time setting of 15, and over 20 frames.

```
-- Lingo syntax
_movie.puppetPalette("Rainbow", 15, 20)

// JavaScript syntax
_movie.puppetPalette("Rainbow", 15, 20);
```

See also

Movie

puppetSprite()

Usage

```
-- Lingo syntax
_movie.puppetSprite(intSpriteNum, bool)

// JavaScript syntax
_movie.puppetSprite(intSpriteNum, bool);
```

Description

Movie method; determines whether a sprite channel is a puppet and under script control (TRUE) or not a puppet and under the control of the Score (FALSE).

While the playhead is in the same sprite, turning off the sprite channel's puppetting using the syntax puppetSprite(*intSpriteNum*, FALSE) resets the sprite's properties to those in the Score.

The sprite channel's initial properties are whatever the channel's settings are when the puppetSprite() method is executed. You can use script to change sprite properties as follows:

- If a sprite channel is a puppet, any changes that script makes to the channel's sprite properties remain in effect after the playhead exits the sprite.
- If a sprite channel is not a puppet, any changes that script makes to a sprite last for the life of the current sprite only.

The channel must contain a sprite when you use the puppetSprite() method.

Making the sprite channel a puppet lets you control many sprite properties—such as member, locH, and width—from script after the playhead exits the sprite.

Use the syntax puppetSprite(*intSpriteNum*, FALSE) to return control to the Score when you finish controlling a sprite channel from script and to avoid unpredictable results that may occur when the playhead is in frames that aren't intended to be puppets.

Note: *Version 6 of Director introduced autopuppetting, which made it unnecessary to explicitly puppet a sprite under most circumstances. Explicit control is still useful if you want to retain complete control over a channel's contents even after a sprite span has finished playing.*

Parameters

intSpriteNum Required. An integer that specifies the sprite channel to test.

bool Required. A boolean value that specifies whether a sprite channel is under script control (TRUE) or under the control of the Score (FALSE).

Example

This statement makes the sprite in channel 15 a puppet:

```
-- Lingo syntax
_movie.puppetSprite(15, TRUE)

// JavaScript syntax
_movie.puppetSprite(15, true);
```

This statement removes the puppet condition from the sprite in the channel numbered i + 1:

```
-- Lingo syntax
_movie.puppetSprite(i + 1, FALSE)

// JavaScript syntax
_movie.puppetSprite(i + 1, false);
```

See also

makeScriptedSprite(), Movie, Sprite Channel

puppetTempo()

Usage

```
-- Lingo syntax
_movie.puppetTempo(intTempo)

// JavaScript syntax
_movie.puppetTempo(intTempo);
```

Description

Movie method; causes the tempo channel to act as a puppet and sets the tempo to a specified number of frames.

When the tempo channel is a puppet, script can override the tempo setting in the Score and change the tempo assigned to the movie.

It's unnecessary to turn off the puppet tempo condition to make subsequent tempo changes in the Score take effect.

Note: *Although it is theoretically possible to achieve frame rates up to 30,000 frames per second (fps) with the puppetTempo() method, you could do this only with little animation and a very powerful machine.*

Parameters

intTempo Required. An integer that specifies the tempo.

Example

This statement sets the movie's tempo to 30 fps:

```
-- Lingo syntax
_movie.puppetTempo(30)

// JavaScript syntax
_movie.puppetTempo(30);
```

This statement increases the movie's old tempo by 10 fps:

```
-- Lingo syntax
_movie.puppetTempo(oldTempo + 10)

// JavaScript syntax
_movie.puppetTempo(oldTempo + 10);
```

See also

Movie

puppetTransition()

Usage

```
-- Lingo syntax
_movie.puppetTransition(memberObjRef)
_movie.puppetTransition(int {, time} {, size} {, area})

// JavaScript syntax
_movie.puppetTransition(memberObjRef);
_movie.puppetTransition(int {, time} {, size} {, area});
```

Description

Movie method; performs the specified transition between the current frame and the next frame.

To use an Xtra transition cast member, use the `puppetTransition(`*`memberObjRef`*`)` syntax.

To use a built-in Director transition, replace *`int`* with a value in the following table. Replace *`time`* with the number of quarter seconds used to complete the transition. The minimum value is 0; the maximum is 120 (30 seconds). Replace *`size`* with the number of pixels in each chunk of the transition. The minimum value is 1; the maximum is 128. Smaller chunk sizes yield smoother transitions but are slower.

Code	Transition	Code	Transition
01	Wipe right	27	Random rows
02	Wipe left	28	Random columns
03	Wipe down	29	Cover down
04	Wipe up	30	Cover down, left
05	Center out, horizontal	31	Cover down, right
06	Edges in, horizontal	32	Cover left
07	Center out, vertical	33	Cover right
08	Edges in, vertical	34	Cover up

Code	Transition	Code	Transition
09	Center out, square	35	Cover up, left
10	Edges in, square	36	Cover up, right
11	Push left	37	Venetian blinds
12	Push right	38	Checkerboard
13	Push down	39	Strips on bottom, build left
14	Push up	40	Strips on bottom, build right
15	Reveal up	41	Strips on left, build down
16	Reveal up, right	42	Strips on left, build up
17	Reveal right	43	Strips on right, build down
18	Reveal down, right	44	Strips on right, build up
19	Reveal down	45	Strips on top, build left
20	Reveal down, left	46	Strips on top, build right
21	Reveal left	47	Zoom open
22	Reveal up, left	48	Zoom close
23	Dissolve, pixels fast*	49	Vertical blinds
24	Dissolve, boxy rectangles	50	Dissolve, bits fast*
25	Dissolve, boxy squares	51	Dissolve, pixels*
26	Dissolve, patterns	52	Dissolve, bits*

Transitions marked with an asterisk (*) do not work on monitors set to 32 bits.

There is no direct relationship between a low time value and a fast transition. The actual speed of the transition depends on the relation of *size* and *time*. For example, if *size* is 1 pixel, the transition takes longer no matter how low the time value, because the computer has to do a lot of work. To make transitions occur faster, use a larger chunk size, not a shorter time.

Replace *area* with a value that determines whether the transition occurs only in the changing area (TRUE) or over the entire Stage (FALSE, default). The *area* variable is an area within which sprites have changed.

Parameters

memberObjRef Required if using an Xtra transition cast member. A reference to the Xtra cast member to use as the transition.

int Required if using a built-in Director transition. An integer that specifies the number of the transition to use.

time Optional. An integer that specifies that number of quarter seconds used to complete the transition. Valid values range from 0 to 120.

size Optional. An integer that specifies the number of pixels in each chunk of the transition. Valid values range from 1 to 128.

area Optional. A boolean value that specifies whether the transition occurs only in the changing area (TRUE) or over the entire Stage (FALSE).

Example

The following statement performs a wipe right transition. Because no value is specified for *area*, the transition occurs over the entire Stage, which is the default

```
-- Lingo syntax
_movie.puppetTransition(1)

// JavaScript syntax
_movie.puppetTransition(1);
```

This statement performs a wipe left transition that lasts 1 second, has a chunk size of 20, and occurs over the entire Stage:

```
-- Lingo syntax
_movie.puppetTransition(2, 4, 20, FALSE)

// JavaScript syntax
_movie.puppetTransition(2, 4, 20, false);
```

See also

Movie

put()

Usage

```
-- Lingo syntax
put(value)

// JavaScript syntax
put(value);
```

Description

Top level function; evaluates an expression and displays the result in the Message window.

The functionality of this method is identical to the top level `trace()` method, which is available to both Lingo and JavaScript syntax.

This method can be used as a debugging tool by tracking the values of variables as a movie plays.

Parameters

value Required. The expression to evaluate.

Example

This statement displays the time in the Message window:

```
-- Lingo syntax
put(_system.time())

// JavaScript syntax
put(_system.time());
```

This statement displays the value assigned to the variable `bid` in the Message window:

```
-- Lingo syntax
bid = "Johnson"
put(bid) -- "Johnson"
```

```
// JavaScript syntax
var bid = "Johnson";
put(bid); // Johnson
```

See also

trace()

qtRegisterAccessKey()

Usage

```
-- Lingo syntax
qtRegisterAccessKey(categoryString, keyString)

// JavaScript syntax
qtRegisterAccessKey(categoryString, keyString);
```

Description

Command; allows registration of a key for encrypted QuickTime media.

The key is an application-level key, not a system-level key. After the application unregisters the key or shuts down, the media will no longer be accessible.

Note: *For security reasons, there is no way to display a listing of all registered keys.*

See also

qtUnRegisterAccessKey()

qtUnRegisterAccessKey()

Usage

```
-- Lingo syntax
qtUnRegisterAccessKey(categoryString, keyString)

// JavaScript syntax
qtUnRegisterAccessKey(categoryString, keyString);
```

Description

Command; allows the key for encrypted QuickTime media to be unregistered.

The key is an application-level key, not a system-level key. After the application unregisters the key, only movies encrypted with this key continue to play. Other media will no longer be accessible.

See also

qtRegisterAccessKey()

queue()

Usage

```
-- Lingo syntax
soundChannelObjRef.queue(memberObjRef)
soundChannelObjRef.queue(propList)

// JavaScript syntax
soundChannelObjRef.queue(memberObjRef);
soundChannelObjRef.queue(propList);
```

Description

Sound Channel method; adds a sound cast member to the queue of a sound channel.

Once a sound has been queued, it can be played immediately with the `play()` method. This is because Director preloads a certain amount of each sound that is queued, preventing any delay between the `play()` method and the start of playback. The default amount of sound that is preloaded is 1500 milliseconds. This parameter can be modified by passing a property list containing one or more parameters with the `queue()` method. These parameters can also be passed with the `setPlayList()` method.

To see an example of `queue()` used in a completed movie, see the Sound Control movie in the Learning/Lingo folder inside the Director application folder.

Parameters

`memberObjRef` Required if specifying a sound cast member. A reference to the sound cast member to queue.

`propList` Required if passing a property list as parameters. A property list that applies to the sound cast member to queue. These properties include:

Property	Description
#member	The sound cast member to queue. This property must be provided; all others are optional.
#startTime	The time within the sound at which playback begins, in milliseconds. The default is the beginning of the sound. See `startTime`.
#endTime	The time within the sound at which playback ends, in milliseconds. The default is the end of the sound. See `endTime`.
#loopCount	The number of times to play a loop defined with `#loopStartTime` and `#loopEndTime`. The default is 1. See `loopCount`.
#loopStartTime	The time within the sound to begin a loop, in milliseconds. See `loopStartTime`.
#loopEndTime	The time within the sound to end a loop, in milliseconds. See `loopEndTime`.
#preloadTime	The amount of the sound to buffer before playback, in milliseconds. See `preloadTime`.

Example

The following handler queues and plays two sounds. The first sound, cast member Chimes, is played in its entirety. The second sound, cast member introMusic, is played starting at its 3-second point, with a loop repeated 5 times from the 8-second point to the 8.9 second point, and stopping at the 10-second point.

```
-- Lingo syntax
on playMusic
    sound(2).queue(member("Chimes"))
    sound(2).queue([#member:member("introMusic"), #startTime:3000, #endTime:10000,
#loopCount:5, #loopStartTime:8000, #loopEndTime:8900])
```

```
    sound(2).play()
end playMusic

// JavaScript syntax
function playMusic() {
    sound(2).queue(member("Chimes"))
    sound(2).queue(propList("member",member("introMusic"), "startTime",3000,
"endTime",10000, "loopCount",5, "loopStartTime",8000, "loopEndTime",8900));
    sound(2).play();
}
```

See also

endTime, loopCount, loopEndTime, loopStartTime, pause() (Sound Channel), play() (Sound Channel), preLoadTime, setPlayList(), Sound Channel, startTime, stop() (Sound Channel)

queue() (3D)

Usage

```
member(whichCastmember).model(whichModel).bonesPlayer.queue(motionName {, looped,
startTime, endTime, scale, offset})
member(whichCastmember).model(whichModel).keyframePlayer.queue(motionName {, looped,
startTime, endTime, scale, offset})
```

Description

3D `keyframePlayer` and `bonesPlayer` modifier command; adds a specified motion to the end of the modifier's `playList` property. The motion is executed by the model when all the motions ahead of it in the playlist are finished playing.

Parameters

`motionName` Required. Specifies the name of the motion to add.

`looped` Optional. Specifies whether the motion plays once (`FALSE`) or continuously (`TRUE`).

`startTime` Optional. Measured in milliseconds from the beginning of the motion. When *`looped`* is `FALSE`, the motion begins at *`offset`* and ends at *`endTime`*. When *`looped`* is `TRUE`, the first iteration of the loop begins at *`offset`* and ends at *`endTime`*. All subsequent repetitions begin at *`startTime`* and end at *`endTime`*.

`endTime` Optional. Measured in milliseconds from the beginning of the motion. When *`looped`* is `FALSE`, the motion begins at *`offset`* and ends at *`endTime`*. When *`looped`* is `TRUE`, the first iteration of the loop begins at *`offset`* and ends at *`endTime`*. All subsequent repetitions begin at *`cropStart`* and end at *`endTime`*. Set *`endTime`* to -1 if you want the motion to play to the end.

`scale` Optional. Specifies the actual speed of the motion's playback. *`scale`* is multiplied by the `playRate` property of the model's `#keyframePlayer` modifier or `#bonesPlayer` modifier to determine the actual speed of the motion's playback.

`offset` Optional. Measured in milliseconds from the beginning of the motion. When *`looped`* is `FALSE`, the motion begins at *`offset`* and ends at *`endTime`*. When *`looped`* is `TRUE`, the first iteration of the loop begins at *`offset`* and ends at *`endTime`*. All subsequent repetitions begin at *`startTime`* and end at *`endTime`*.

Example

The following Lingo adds the motion named Fall to the end of the `bonesPlayer` playlist of the model named Walker. When all motions before Fall in the playlist have been executed, Fall will play one time from beginning to end.

```
sprite(1).member.model("Walker").bonesPlayer.queue("Fall", 0, 0, -1, 1, 0)
```

The following Lingo adds the motion named Kick to the end of the `bonesPlayer` playlist of the model named Walker. When all motions before Kick in the playlist have been executed, a section of Kick will play in a continuous loop. The first iteration of the loop will begin 2000 milliseconds from the motion's beginning. All subsequent iterations of the loop will begin 1000 milliseconds from Kick's beginning and will end 5000 milliseconds from Kick's beginning. The rate of playback will be three times the `playRate` property of the model's `bonesPlayer` modifier.

```
sprite(1).member.model("Walker").bonesPlayer.queue("Kick", 1, 1000, 5000, 3, 2000)
```

See also

`play() (3D)`, `playNext() (3D)`, `playRate (3D)`

QuickTimeVersion()

Usage

```
-- Lingo syntax
QuickTimeVersion()

// JavaScript syntax
QuickTimeVersion();
```

Description

Function; returns a floating-point value that identifies the current installed version of QuickTime and replaces the current `QuickTimePresent` function.

In Windows, if multiple versions of QuickTime 3.0 or later are installed, `QuickTimeVersion()` returns the latest version number. If a version before QuickTime 3.0 is installed, `QuickTimeVersion()` returns version number 2.1.2 regardless of the version installed.

Parameters

None.

Example

This statement uses `QuickTimeVersion()` to display in the Message window the version of QuickTime that is currently installed:

```
-- Lingo syntax
put(QuickTimeVersion())

// JavaScript syntax
put(QuickTimeVersion());
```

quit()

Usage

```
-- Lingo syntax
_player.quit()

// JavaScript syntax
_player.quit();
```

Description

Player method; exits from Director or a projector to the Windows desktop or Mac Finder.

Parameters

None.

Example

This statement tells the computer to exit to the Windows desktop or Mac Finder when the user presses Control+Q (Windows) or Command+Q (Mac):

```
-- Lingo syntax
if (_key.key = "q" and _key.commandDown) then
    _player.quit()
end if

// JavaScript syntax
if (_key.key == "q" && _key.commandDown) {
    _player.quit();
}
```

See also

Player

ramNeeded()

Usage

```
-- Lingo syntax
_movie.ramNeeded(intFromFrame, intToFrame)

// JavaScript syntax
_movie.ramNeeded(intFromFrame, intToFrame);
```

Description

Movie method; determines the memory needed, in bytes, to display a range of frames. For example, you can test the size of frames containing 32-bit artwork: if `ramNeeded()` is larger than `freeBytes()`, then go to frames containing 8-bit artwork and divide by 1024 to convert bytes to kilobytes (K).

Parameters

intFromFrame Required. An integer that specifies the number of the first frame in the range.

intToFrame Required. An integer that specifies the number of the last frame in the range.

Example

This statement sets the variable `frameSize` to the number of bytes needed to display frames 100 to 125 of the movie:

```
-- Lingo syntax
frameSize = _movie.ramNeeded(100, 125)

// JavaScript syntax
var frameSize = _movie.ramNeeded(100, 125);
```

This statement determines whether the memory needed to display frames 100 to 125 is more than the available memory, and, if it is, branches to the section using cast members that have lower color depth:

```
-- Lingo syntax
if (_movie.ramNeeded(100, 125) > _system.freeBytes) then
    _movie.go("8-bit")
end if

// JavaScript syntax
if (_movie.ramNeeded(100, 125) > _system.freeBytes) {
    _movie.go("8-bit");
}
```

See also

freeBytes(), Movie

random()

Usage

```
-- Lingo syntax
random(integerExpression)

// JavaScript syntax
random(integerExpression);
```

Description

Top level function; returns a random integer in the range 1 to a specified value. This function can be used to vary values in a movie, such as to vary the path through a game, assign random numbers, or change the color or position of sprites.

To start a set of possible random numbers with a number other than 1, subtract the appropriate amount from the `random()` function. For example, the expression `random(n + 1) - 1` uses a range from 0 to the number `n`.

Parameters

`integerExpression` Required. Specifies the maximum value of the random number.

Example

This statement assigns random values to the variable `diceRoll`:

```
-- Lingo syntax
diceRoll = (random(6) + random(6))

// JavaScript syntax
var diceRoll = (random(6) + random(6));
```

This statement randomly changes the foreground color of sprite 10:

```
-- Lingo syntax
sprite(10).foreColor = (random(256) - 1)

// JavaScript syntax
sprite(10).foreColor = (random(256) - 1);
```

This handler randomly chooses which of two movie segments to play:

```
-- Lingo syntax
on SelectScene
    if (random(2) = 2) then
        _movie.go("11a")
```

```
    else
        _movie.go("11b")
    end if
end

// JavaScript syntax
function SelectScene() {
    if (random(2) == 1) {
        _movie.go("11a");
    } else {
        _movie.go("11b");
    }
}
```

This statement produces a random multiple of 5 in the range 5 to 100:

```
-- Lingo syntax
theScore = (5 * random(20))

// JavaScript syntax
var theScore = (5 * random(20));
```

randomVector()

Usage

```
-- Lingo syntax
randomVector()

// JavaScript syntax
randomVector();
```

Description

Top level function; returns a unit vector describing a randomly chosen point on the surface of a unit sphere.

This function differs from `vector(random(10)/10.0, random(10)/10.0, random(10)/10.0,)` in that the resulting vector using `randomVector()` is guaranteed to be a unit vector.

A unit vector always has a length of one.

Parameters

None.

Example

These statements create and display two randomly defined unit vectors in the Message window:

```
-- Lingo syntax
vec1 = randomVector()
vec2 = randomVector()
put(vec1 & RETURN & vec2)

// JavaScript syntax
var vec1 = randomVector();
var vec2 = randomVector();
put(vec1 + "\n" + vec2);
```

See also

vector()

randomVector

Usage

```
randomVector()
```

Description

3D command; returns a unit vector describing a randomly chosen point on the surface of a unit sphere. This method differs from `vector( random(10)/10.0, random(10)/10.0, random(10)/10.0)`, in that the resulting vector is guaranteed to be a unit vector.

Parameters

None.

Example

These statements create and display two randomly defined unit vectors in the Message window:

```
vec = randomVector()
put vec
-- vector(-0.1155, 0.9833, -0.1408)
vec2 = randomVector()
put vec2
-- vector(0.0042, 0.8767, 0.4810)
```

See also

getNormalized, generateNormals(), normalize

rawNew()

Usage

```
parentScript.rawNew()
rawNew(parentScript)
```

Description

Function; creates a child object from a parent script without calling its `on new` handler. This allows a movie to create child objects without initializing the properties of those child objects. This is particularly useful when you want to create large numbers of child objects for later use. To initialize the properties of one of these raw child objects, call its `on new` handler.

Parameters

None.

Example

This statement creates a child object called RedCar from the parent script CarParentScript without initializing its properties:

```
RedCar = script("CarParentScript").rawNew()
```

This statement initializes the properties of the child object RedCar:

```
RedCar.new()
```

See also

new(), script()

readChar()

Usage

```
-- Lingo syntax
fileioObjRef.readChar()

// JavaScript syntax
fileioObjRef.readChar();
```

Description

Fileio method; Reads the next character of a file and returns it.

You must first open a file by calling `openFile()` before using `readChar()` to read a character. When reading unicode files, `readchar()` returns only the next byte and not the unicode character. To read unicode characters, use the `readFile()` method.

Parameters

None.

Example

The following statement opens a file c:\xtra.txt with Read/Write permission and reads all the characters in the file till it encounters the character 'e'.

```
-- Lingo
objFileio = new xtra("fileio")
objFileio.openFile("c:\xtra.txt",0)
repeat while(numToChar(objFileio.readChar()) <> 'e')
    put numToChar(objFileio.readChar())
end repeat

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile("c:\xtra.txt",0);
while(numToChar(objFileio.readChar()) != 'e')
{
    trace(numToChar(objFileio.readChar()));
}
```

See also

Fileio, openFile()

readFile()

Usage

```
-- Lingo syntax
```

```
fileioObjRef.readFile()

// JavaScript syntax
fileioObjRef.readFile();
```

Description

Fileio method; Reads from the current position to the end of a specified file, and returns the result as a string.

You must first open a file by calling `openFile()` before using `readFile()` to read a file.

Parameters

None.

Example

The following statement opens a file c:\xtra.txt with Read/Write permission and reads the contents of the file.

```
-- Lingo
objFileio = new xtra("fileio")
objFileio.openFile("c:\xtra.txt",0)
contents =objFileio.readFile()
put contents

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile("c:\xtra.txt",0);
var contents = objFileio.readFile();
trace(contents);
```

See also

Fileio, openFile()

readLine()

Usage

```
-- Lingo syntax
fileioObjRef.readLine()

// JavaScript syntax
fileioObjRef.readLine();
```

Description

Fileio method; Reads the next line of a file, including the next RETURN, and returns it as a string.

You must first open a file by calling `openFile()` before using `readLine()` to read a line.

Parameters

None.

Example

The following statement opens a file c:\xtra.txt with Read/Write permission and reads the first line of the file.

```
-- Lingo
objFileio = new xtra("fileio")
objFileio.openFile("c:\xtra.txt",0)
contents =objFileio.readLine()
```

```
put contents

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile("c:\xtra.txt",0);
contents = objFileio.readLine();
trace(contents);
```

See also

Fileio, openFile()

readToken()

Usage

```
-- Lingo syntax
fileioObjRef.readToken(stringSkip, stringBreak)

// JavaScript syntax
fileioObjRef.readToken(stringSkip, stringBreak);
```

Description

Fileio method; Reads the next token and returns it as a string.

You must first open a file by calling `openFile()` before using `readToken()` to read a token.

Parameters

`stringSkip` Required. A string that specifies the set of characters after which the token starts. The string *`stringSkip`* is not included in the returned string.

`stringBreak` Required. A string that specifies the set of characters before which the token ends. The string *`stringBreak`* is not included in the returned string.

Example

The following statement opens a file c:\xtra.txt with Read/Write permission and gets the token between "Director is good" and the string to skip is "D" and string break is "g".

```
-- Lingo
objFileio = new xtra("fileio")
objFileio.openFile("c:\xtra.txt",0)
contents =objFileio.readToken("D","g")
put contents

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile("c:\xtra.txt",0);
contents = objFileio.readToken("D","g");
trace(contents);
```

The output is "irector is".

See also

Fileio, openFile()

readWord()

Usage

```
-- Lingo syntax
fileioObjRef.readWord()

// JavaScript syntax
fileioObjRef.readWord();
```

Description

Fileio method; Reads the next word of a file and returns it as a string.

You must first open a file by calling `openFile()` before using `readWord()` to read a word.

Parameters

None.

Example

The following statement opens a file c:\xtra.txt with Read/Write permission and reads the first word of the file.

```
-- Lingo
objFileio = new xtra("fileio")
objFileio.openFile("c:\xtra.txt",0)
contents =objFileio.readWord()
put contents

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile("c:\xtra.txt",0);
contents = objFileio.readWord();
trace(contents);
```

See also

Fileio, openFile()

realPlayerNativeAudio()

Usage

```
-- Lingo syntax
realPlayerNativeAudio()

// JavaScript syntax
realPlayerNativeAudio();
```

Description

RealMedia function; allows you to get or set the global flag that determines whether the audio portion of the RealMedia cast member is processed by RealPlayer® (TRUE) or by Director (FALSE). This function returns the previous value of the flag.

To be effective, this flag must be set before RealPlayer is first loaded (when the first RealMedia cast member is encountered in the Score or with the first Lingo reference to a RealMedia cast member); any changes to this flag after RealPlayer is loaded are ignored. This flag should be executed in a `prepareMovie` event handler in a movie script. This flag is set for the entire session (from the time the Shockwave Player is launched until it is closed and relaunched), not just for the duration of the current movie.

By default, this flag is set to `FALSE` and audio is processed by Director, which allows you to set the `soundChannel` property and use the standard Lingo sound methods and properties to manipulate the audio stream of a RealMedia sprite, including mixing RealAudio® with other Director audio. If this flag is set to `TRUE`, Lingo control of the sound channel is not processed, and the sound is handled by RealPlayer.

Parameters

None.

Example

The following code shows that the `realPlayerNativeAudio()` function is set to `FALSE`, which means that audio in the RealMedia cast member will be processed by Director:

```
-- Lingo syntax
put(realPlayerNativeAudio())
-- 0

// JavaScript syntax
trace(realPlayerNativeAudio());
// 0
```

The following code sets the `realPlayerNativeAudio()` function to `TRUE`, which means that audio in the RealMedia stream will be processed by RealPlayer and all Lingo control of the sound channel will be ignored:

```
-- Lingo syntax
realPlayerNativeAudio(TRUE)

// JavaScript syntax
realPlayerNativeAudio(1);
```

See also

soundChannel (RealMedia)

realPlayerPromptToInstall()

Usage

```
-- Lingo syntax
realPlayerPromptToInstall()

// JavaScript syntax
realPlayerPromptToInstall();
```

Description

RealMedia function; allows you to get or set a global flag that determines whether automatic detection and alert for RealPlayer 8 is enabled (`TRUE`) or not (`FALSE`).

By default, this function is set to TRUE, which means that if users do not have RealPlayer 8 and attempt to load a movie containing RealMedia, they are automatically asked if they want to go to the RealNetworks® website and install RealPlayer. You can set this flag to FALSE if you want to create your own detection and alert system using the "realPlayerVersion()" on page 502 function and custom code. If this flag is set to FALSE and an alternate RealPlayer 8 detection and alert system is not in place, users without RealPlayer will be able to load movies containing RealMedia cast members, but the RealMedia sprites will not appear.

This function detects the build number of the RealPlayer installed on the user's system to determine whether RealPlayer 8 is installed. On Windows systems, build numbers 6.0.8.132 or later indicate that RealPlayer 8 is installed. On Mac systems, RealPlayer Core component build numbers 6.0.7.1001 or later indicate that RealPlayer 8 is installed.

This flag should be executed in a prepareMovie event handler in a movie script.

This function returns the previous value of the flag.

Parameters

None.

Example

The following code shows that the realPlayerPromptToInstall() function is set to TRUE, which means users who do not have RealPlayer will be prompted to install it:

```
-- Lingo syntax
put(realPlayerPromptToInstall()) -- 1

// JavaScript syntax
trace(realPlayerPromptToInstall()); // 1
```

The following code sets the realPlayerPromptToInstall() function to FALSE, which means that users will not be prompted to install RealPlayer unless you have created a detection and alert system:

```
-- Lingo syntax
realPlayerPromptToInstall(FALSE)


// JavaScript syntax
realPlayerPromptToInstall(0);
```

realPlayerVersion()

Usage

```
-- Lingo syntax
realPlayerVersion()

// JavaScript syntax
realPlayerVersion();
```

Description

RealMedia function; returns a string identifying the build number of the RealPlayer software installed on the user's system, or an empty string if RealPlayer is not installed. Users must have RealPlayer 8 or later in order to view Director movies containing RealMedia content. On Windows systems, build numbers 6.0.8.132 or later indicate that RealPlayer 8 is installed. On Mac systems, RealPlayer Core component build numbers 6.0.7.1001 or later indicate that RealPlayer 8 is installed.

The purpose of this function is to allow you to create your own RealPlayer detection and alert system, if you do not want to use the one provided by the function "realPlayerPromptToInstall()" on page 501.

If you choose to create your own detection and alert system using the `realPlayerVersion()` function, you must do the following:

- Call `realPlayerPromptToInstall(FALSE)` (by default, this function is set to `TRUE`) before any RealMedia cast members are referenced in Lingo or appear in the Score. This function should be set in a `prepareMovie` event handler in a movie script.
- Use the `xtraList` system property to verify that the Xtra for RealMedia (RealMedia Asset.x32) is listed in the Movie Xtras dialog box. The `realPlayerVersion()` function will not work if the Xtra for RealMedia is not present.

The build number returned by this function is the same as the build number you can display in RealPlayer.

To view the RealPlayer build number in Windows:

1. Launch RealPlayer.
2. Choose About RealPlayer from the Help menu.

   In the window that appears, the build number appears at the top of the screen in the second line.

To view the RealPlayer build number on the Mac:

1. Launch RealPlayer.
2. Choose About RealPlayer from the Apple® menu.

   The About RealPlayer dialog box appears. Ignore the build number listed in the second line at the top of the screen; it is incorrect.
3. Click the Version Info button.

   The RealPlayer Version Information dialog box appears.
4. Select RealPlayer Core in the list of installed components.

   The build number shown for RealPlayer Core component (for example, 6.0.8.1649) is the same as the build number returned by `realPlayerVersion()`.

Parameters

None.

Example

The following code shows that build number of the RealPlayer® installed on the system is 6.0.9.357:

```
-- Lingo syntax
put(realPlayerVersion())

// JavaScript syntax
put(realPlayerVersion());
```

recordFont

Usage

```
recordFont(whichCastMember, font {[,face]} {,[bitmapSizes]} {,characterSubset} {,
userFontName})
```

Description

Command; embeds a TrueType or Type 1 font as a cast member. Once embedded, these fonts are available to the author just like other fonts installed in the system.

You must create an empty font cast member with the `new()` command before using `recordFont`.

The command creates a Shock Font in *`whichCastMember`* using the font named in the *`font`* parameter. The value returned from the command reports whether the operation was successful. Zero indicates success.

Parameters

`font` Required. Specifies the name of original font to be recorded.

`face` Optional. Specifies a list of symbols indicating the face of the original font. Possible values are `#plain`, `#bold`, `#italic`. If you do not provide a value for this parameter, `#plain` is used.

`bitmapSizes` Optional. Specifies a list of integers specifying the sizes for which bitmaps are to be recorded. This parameter can be empty. If you omit this parameter, no bitmaps are generated. These bitmaps typically look better at smaller point sizes (below 14 points) but take up more memory.

`characterSubset` Optional. Specifies a string of characters to be encoded. Only the specified characters will be available in the font. If this parameter is omitted, all characters are encoded. If only certain characters are encoded but an unencoded character is used, that character is displayed as an empty box.

`userFontName` Optional. Specifies a string to use as the name of the newly recorded font cast member.

Example

This statement creates a simple Shock Font using only the two arguments for the cast member and the font to record:

```
-- Lingo
myNewFontMember = new(#font)
recordFont(myNewFontMember, "Lunar Lander")

// Javascript
var myNewFontMember = new(symbol("font")) ;
myNewFontMember.recordFont( "Lunar Lander") ;
```

This statement specifies the bitmap sizes to be generated and the characters for which the font data should be created:

```
-- Lingo
myNewFontMember = new(#font)
recordfont(mynewmember,"lunar lander",[],[14, 18, 45], "Lunar Lander Game High Score First
Last Name")

// Javascript
var myNewFontMember = new(symbol("font")) ;
recordfont(mynewmember,"lunar lander",[],[14, 18, 45], "Lunar Lander Game High Score First
Last Name") ;
```

Note: *Since `recordFont` resynthesizes the font data rather than using it directly, there are no legal restrictions on Shock Font distribution.*

See also

`newMember()`

rect()

Usage

```
-- Lingo syntax
rect(intLeft, intTop, intRight, intBottom)

// JavaScript syntax
rect(intLeft, intTop, intRight, intBottom);
```

Description

Top level function; defines a rectangle.

You can perform arithmetic operations on rectangles using both Lingo and JavaScript syntax. If you add a single value to a rectangle, Lingo and JavaScript syntax adds it to each element in the rectangle.

You can refer to rectangle components by list syntax or property syntax. For example, the following assignments set both `myRectWidth1` and `myRectWidth2` to 50:

```
// JavaScript syntax
var myRect = rect(40,30,90,70);
var myRectWidth1 = myRect.right - myRect.left; // 50
var myRectWidth2 = myRect[3] - myRect[1]; // 50
```

To see an example of `rect()` used in a completed movie, see the Imaging movie in the Learning/Lingo folder inside the Director application folder.

Parameters

`intLeft` Required. An integer that specifies the number of pixels that the left side of the rectangle is from the left edge of the Stage.

`intTop` Required. An integer that specifies the number of pixels that the top side of the rectangle is from the top edge of the Stage.

`intRight` Required. An integer that specifies the number of pixels that the right side of the rectangle is from the left edge of the Stage.

`intBottom` Required. An integer that specifies the number of pixels that the bottom side of the rectangle is from the top edge of the Stage.

Example

This statement sets the variable `newArea` to a rectangle whose left side is at 100, top is at 150, right side is at 300, and bottom is at 400 pixels:

```
-- Lingo syntax
newArea = rect(100, 150, 300, 400)

// JavaScript syntax
var newArea = rect(100, 150, 300, 400);
```

In Lingo only, the following statement sets the variable `newArea` to the rectangle defined by the points `firstPoint` and `secondPoint`:

```
-- Lingo syntax
firstPoint = point(100, 150)
secondPoint = point(300, 400)
newArea = rect(firstPoint, secondPoint)
```

In Lingo only, these statements add and subtract values for rectangles:

```
-- Lingo syntax
put(rect(0, 0, 100, 100) + rect(30, 55, 120, 95)) -- rect(30, 55, 220, 195)
put(rect(0, 0, 100, 100) -rect(30, 55, 120, 95)) -- rect(-30, -55, -20, 5)
```

In Lingo only, this statement adds 80 to each coordinate in a rectangle:

```
-- Lingo syntax
put(rect(60, 40, 120, 200) + 80) -- rect(140, 120, 200, 280)
```

In Lingo only, this statement divides each coordinate in a rectangle by 3:

```
-- Lingo syntax
put(rect(60, 40, 120, 200) / 3) -- rect(20, 13, 40, 66)
```

See also

`point()`, `quad`

registerForEvent()

Usage

```
member(whichCastmember).registerForEvent(eventName, handlerName, scriptObject {, begin,
period, repetitions})
```

Description

3D command; declares the specified handler as the handler to be called when the specified event occurs within the specified cast member.

The following parameter descriptions apply to both the `registerForEvent()` and the `registerScript()` commands.

Note: *You can associate the registration of a script with a particular node rather than with a cast member by using the `registerScript()` command.*

Parameters

`eventName` Required. Specifies the name of the event. The event can be any of the following predefined events, or any custom event that you define:

- `#collideAny` is a collision event.
- `#collideWith` is a collision event involving this specific model. The `setCollisionCallback()` command is a shortcut for using the `registerScript()` command for the `#collideWith` event.
- `#animationStarted` and `#animationEnded` are notification events that occur when a bones or keyframe animation starts or stops playing. The handler will receive three arguments: *`eventName`*, *`motion`*, and *`time`*. The *`eventName`* argument is either `#animationStarted` or `#animationEnded`. The *`motion`* argument is the name of the motion that has started or stopped playing, and *`time`* is the current time of the motion.
- For looping animations, the `#animationStarted` event is issued only for the first loop, not for subsequent loops. During a blend of two animations, this event will be sent when the blending begins.
- When a series of animations is queued for the model and the animation's `autoBlend` property is set to `TRUE`, the `#animationEnded` event may occur before the apparent end of a given motion. This is because the `autoBlend` property may make the motion appear to continue even though the animation has completed as defined.

- `#timeMS` is a time event. The first `#timeMS` event occurs when the number of milliseconds specified in the *`begin`* parameter have elapsed after `registerForEvent` is called. The *`period`* parameter determines the number of milliseconds between `#timeMS` events when the value of *`repetitions`* is greater then 0. If *`repetitions`* is 0, the `#timeMS` event occurs indefinitely.

`handlerName` Required. Specifies the name of the handler that will be called when the event *`eventName`* occurs; this handler is found in the script object indicated by *`scriptObject`*. The handler is sent the following arguments:

- `type` is always 0.
- `delta` is the elapsed time in milliseconds since the last `#timeMS` event.
- `time` is the number of milliseconds since the first `#timeMS` event occurred. For example, if there are three iterations with a period of 500 ms, the first iteration's time will be 0, the second iteration will be 500, and the third will be 1000.
- `duration` is the total number of milliseconds that will elapse between the `registerForEvent` call and the last `#timeMS` event. For example, if there are five iterations with a period of 500 ms, the duration is 2500 ms. For tasks with unlimited iterations, the duration is 0.
- `systemTime` is the absolute time in milliseconds since the Director movie started.

`scriptObject` Required. Specifies the script object that contains the handler *`handlerName`*. If 0 is specified for *`scriptObject`*, then the first event handler with the given name found in a movie script is called.

`begin` Optional. Specifies the number of milliseconds after `registerForEvent()` is called that the first `#timeMS` event occurs.

`period` Optional. Specifies the number of milliseconds between `#timeMS` events when the value of *`repetitions`* is greater then 0.

`repetitions` Optional. Specifies the number of repetitions for the `#timeMS` event. If *`repetitions`* is 0, the `#timeMS` event occurs indefinitely.

Example

This statement registers the `promptUser` event handler found in a movie script to be called twice at an interval of 5 seconds:

```
member("Scene").registerForEvent(#timeMS, #promptUser, 0, 5000, 5000, 2)
```

This statement registers the `promptUser` event handler found in a movie script to be called each time a collision occurs within the cast member named Scene:

```
member("Scene").registerForEvent(#collideAny, #promptUser, 0)
```

This statement declares the on `promptUser` handler in the same script that contains the `registerForEvent` command to be called when any object collides with the model named Pluto in the cast member named Scene:

```
member("Scene").registerForEvent(#collideWith, #promptUser, me,
member("Scene").model("Pluto"))
```

See also

`setCollisionCallback()`, `registerScript()`, `play() (3D)`, `playNext() (3D)`, `autoblend`, `blendTime`, `sendEvent`, `unregisterAllEvents`

registerScript()

Usage

```
member(whichCastmember).model(whichModel).registerScript(eventName, handlerName,
scriptObject {, begin, period, repetitions})
member(whichCastmember).camera(whichCamera).registerScript(eventName, handlerName,
scriptObject {, begin, period, repetitions})
member(whichCastmember).light(whichLight).registerScript(eventName, handlerName,
scriptObject {, begin, period, repetitions})
member(whichCastmember).group(whichGroup).registerScript(eventName, handlerName,
scriptObject {, begin, period, repetitions})
```

Description

3D command; registers the specified handler to be called when the specified event occurs for the referenced node.

The following parameter descriptions apply to both the `registerForEvent()` and the `registerScript()` commands.

Parameters

`eventName` Required. Specifies the name of the event. The event can be any of the following predefined events, or any custom event that you define:

- `#collideAny` is a collision event.
- `#collideWith` is a collision event involving this specific model. The `setCollisionCallback()` command is a shortcut for using the `registerScript()` command for the `#collideWith` event.
- `#animationStarted` and `#animationEnded` are notification events that occur when a bones or keyframe animation starts or stops playing. The handler will receive three arguments: *`eventName`*, *`motion`*, and *`time`*. The *`eventName`* argument is either `#animationStarted` or `#animationEnded`. The *`motion`* argument is the name of the motion that has started or stopped playing, and *`time`* is the current time of the motion.

  For looping animations, the `#animationStarted` event is issued only for the first loop, not for subsequent loops. During a blend of two animations, this event will be sent when the blending begins.

  When a series of animations is queued for the model and the animation's `autoBlend` property is set to `TRUE`, the `#animationEnded` event may occur before the apparent end of a given motion. This is because the `autoBlend` property may make the motion appear to continue even though the animation has completed as defined.
- `#timeMS` is a time event. The first `#timeMS` event occurs when the number of milliseconds specified in the *`begin`* parameter have elapsed after `registerForEvent` is called. The *`period`* parameter determines the number of milliseconds between `#timeMS` events when the value of *`repetitions`* is greater then 0. If *`repetitions`* is 0, the `#timeMS` event occurs indefinitely.

`handlerName` Required. Specifies the name of the handler that will be called when the event *`eventName`* occurs; this handler is found in the script object indicated by *`scriptObject`*. The handler is sent the following arguments:

- `type` is always 0.
- `delta` is the elapsed time in milliseconds since the last `#timeMS` event.
- `time` is the number of milliseconds since the first `#timeMS` event occurred. For example, if there are three iterations with a period of 500 ms, the first iteration's time will be 0, the second iteration will be 500, and the third will be 1000.

- `duration` is the total number of milliseconds that will elapse between the `registerForEvent` call and the last `#timeMS` event. For example, if there are five iterations with a period of 500 ms, the duration is 2500 ms. For tasks with unlimited iterations, the duration is 0.
- `systemTime` is the absolute time in milliseconds since the Director movie started.

`scriptObject` Required. Specifies the script object that contains the handler *`handlerName`*. If 0 is specified for *`scriptObject`*, then the first event handler with the given name found in a movie script is called.

`begin` Optional. Specifies the number of milliseconds after `registerForEvent()` is called that the first `#timeMS` event occurs.

`period` Optional. Specifies the number of milliseconds between `#timeMS` events when the value of *`repetitions`* is greater then 0.

`repetitions` Optional. Specifies the number of repetitions for the `#timeMS` event. If *`repetitions`* is 0, the `#timeMS` event occurs indefinitely.

Example

This statement registers the `messageReceived` event handler found in a movie script to be called when the model named Player receives the custom user defined event named `#message`:

```
member("Scene").model("Player").registerScript(#message, #messageReceived, 0)
```

This statement registers the `collisionResponder` event handler found in the same script as the `registerScript` command to be called each time a collision occurs between the model named Player and any other model using the `#collision` modifier:

```
member("Scene").model("Player").registerScript(#collideWith, #collisionResponder, me)
```

See also

`registerForEvent()`, `sendEvent`, `setCollisionCallback()`

removeBackdrop

Usage

```
member(whichCastmember).camera(whichCamera).removeBackdrop(index)
```

Description

3D command; removes the backdrop found in a specified position from the camera's list of backdrops to display.

Parameters

`index` Required. Specifies the index position of the backdrop in the list of backdrops.

Example

The following statement removes the third backdrop from the list of backdrops for camera 1 within the member named Scene. The backdrop will disappear from the stage if there are any sprites currently using this camera.

```
-- Lingo
member("Scene").camera[1].removeBackdrop(3)

// Javascript
member("Scene").getProp("camera", 1).removeBackdrop(3) ;
```

removeFromWorld

Usage

```
member(whichCastmember).model(whichModel).removeFromWorld()
member(whichCastmember).light(whichLight).removeFromWorld()
member(whichCastmember).camera(whichCamera).removeFromWorld()
member(whichCastmember).group(whichGroup).removeFromWorld()
```

Description

3D command; for models, lights, cameras or groups whose parent hierarchy terminates in the world object, this command sets their parent to void and removes them from the world.

For objects whose parent hierarchy does not terminate in the world, this command does nothing.

Parameters

None.

Example

This command removes the model named gbCyl from the 3D world of the cast member named Scene:

```
-- Lingo
member("Scene").model("gbCyl").removeFromWorld()

// Javascript
member("Scene").getPropRef("model" , a).removeFromWorld() ;
// where a is the number index for gbCyl model.
```

removeLast()

Usage

```
member(whichCastmember).model(whichModel).bonesPlayer.removeLast()
member(whichCastmember).model(whichModel).keyframePlayer.removeLast()
```

Description

3D `keyframePlayer` and `bonesPlayer` modifier command; removes the last motion from the modifier's `playlist`.

Parameters

None.

Example

This statement removes the last motion from the playlist of the `bonesPlayer` modifier for the model named Walker:

```
member("MyWorld").model("Walker").bonesPlayer.removelast()
```

removeModifier

Usage

```
member(whichCastmember).model(whichModel).removeModifier.(#whichModifier)
```

Description

3D command; removes the specified modifier from the specified model.

This command returns TRUE if it completes successfully, and FALSE if *#whichModifier* is not a valid modifier, or if the modifier was not attached to the model.

Parameters

whichModifier Required. Specifies the modifier to remove.

Example

This statement removes the #toon modifier from the model named Box:

```
-- Lingo
member("shapes").model("Box").removeModifier(#toon)

// JavaScript syntax
member("shapes").getPropRef("model" , i).removeModifier(symbol("toon"));
// where "i" is the number index.
```

See also

addModifier, modifier, modifier[], modifiers

removeOverlay

Usage

```
member(whichCastmember).camera(whichCamera).removeOverlay(index)
```

Description

3D command; removes the overlay found in a specified position from the camera's list of overlays to display.

Parameters

index Required. Specifies the index position of the overlay in the list of overlays.

Example

The following statement removes the first overlay from the list of overlays for the camera being used by sprite 5. The overlay disappears from the Stage.

```
-- Lingo
sprite(5).camera.removeOverlay(1)

// Javascript
sprite(5).camera.removeOverlay(1) ;
```

See also

overlay

removeScriptedSprite()

Usage

```
-- Lingo syntax
```

```
spriteChannelObjRef.removeScriptedSprite()

// JavaScript syntax
spriteChannelObjRef.removeScriptedSprite();
```

Description

Sprite Channel method; switches control of a sprite channel from script back to the Score.

Parameters

None.

Example

The following statement removes the scripted sprite from sprite channel 5:

```
-- Lingo syntax
channel(5).removeScriptedSprite()

// JavaScript syntax
channel(5).removeScriptedSprite();
```

See also

`makeScriptedSprite()`, `Sprite Channel`

resetWorld

Usage

```
member(whichCastmember).resetWorld()
member(whichTextCastmember).resetWorld()
```

Description

3D command; resets the member's properties of the referenced 3D cast member to the values stored when the member was first loaded into memory. The member's `state` property must be either 0 (unloaded), 4 (media loaded), or -1 (error) before this command can be used, otherwise a script error will occur.

This command differs from `revertToWorldDefaults` in that the values used are taken from the state of the member when it was first loaded into memory rather than from the state of the member when it was first created.

Parameters

None.

Example

This statement resets the properties of the cast member named Scene to the values they had when the member was first loaded into memory:

```
-- Lingo
member("Scene").resetWorld()

// Javascript
member("Scene").resetWorld() ;
```

See also

`revertToWorldDefaults`

resolveA

Usage

```
collisionData.resolveA(bResolve)
```

Description

3D collision method; overrides the collision behavior set by the `collision.resolve` property for `collisionData.modelA`. Call this function only if you wish to override the behavior set for modelA using `collision.resolve`.

Parameters

`bResolve` Required. Specifies whether the collision for modelA is resolved. If *`bResolve`* is `TRUE`, then the collision for the modelA is resolved; if *`bResolve`* is `FALSE` the collision for modelA is not resolved.

See also

`collisionData`, `registerScript()`, `resolve`, `modelA`, `setCollisionCallback()`

resolveB

Usage

```
collisionData.resolveB(bResolve)
```

Description

3D collision method; overrides the collision behavior set by the `collision.resolve` property for `collisionData.modelB`. Call this function only if you wish to override the behavior set for modelB using `collision.resolve`.

Parameters

`bResolve` Required. Specifies whether the collision for modelB is resolved. If *`bResolve`* is `TRUE`, then the collision for the modelB is resolved; if *`bResolve`* is `FALSE` the collision for modelB is not resolved.

See also

`collisionData`, `resolve`, `registerScript()`, `modelB`, `setCollisionCallback()`

restart()

Usage

```
-- Lingo syntax
_system.restart()

// JavaScript syntax
_system.restart();
```

Description

System method; closes all open applications and restarts the computer.

Parameters

None.

Example

This statement restarts the computer when the user presses Command+R (Mac) or Control+R (Windows):

```
-- Lingo syntax
if (_key.key = "r" and _key.commandDown) then
 _system.restart()
end if

// JavaScript syntax
if (_key.key = "r" && _key.commandDown) {
 _system.restart();
}
```

See also

System

restore()

Usage

```
-- Lingo syntax
windowObjRef.restore()

// JavaScript syntax
windowObjRef.restore();
```

Description

Window method; restores a window after it has been maximized.

Use this method when making custom titlebars for movies in a window (MIAWS).

Parameters

None.

Example

This statement restores the maximized window named Control Panel:

```
-- Lingo syntax
window("Control Panel").restore()

// JavaScript syntax
window("Control Panel").restore();
```

See also

maximize(), Window

result

Usage

```
the result
```

Description

Function; displays the value of the return expression from the last handler executed.

The `result` function is useful for obtaining values from movies that are playing in windows and tracking Lingo's progress by displaying results of handlers in the Message window as the movie plays.

To return a result from a handler, assign the result to a variable and then check the variable's value. Use a statement such as `set myVariable = function()`, where `function()` is the name of a specific function.

Parameters

None.

Example

This handler returns a random roll for two dice:

```
on diceRoll
    return random(6) + random(6)
end
```

In the following example, the two statements

```
diceRoll
roll = the result
```

are equivalent to this statement:

```
set roll = diceRoll()
```

The statement set `roll = diceRoll` would not call the handler because there are no parentheses following `diceRoll`; `diceRoll` here is considered a variable reference.

See also

return (keyword)

resume()

Usage

```
-- Lingo syntax
animGifSpriteRef.resume()

// JavaScript syntax
animGifSpriteRef.resume();
```

Description

Animated GIF method; causes the sprite to resume playing from the frame after the current frame if it's been paused. This command has no effect if the animated GIF sprite has not been paused.

Parameters

None.

See also

rewind() (Animated GIF, Flash)

returnToTitle()

Usage

```
-- Lingo syntax
dvdObjRef.returnToTitle()

// JavaScript syntax
dvdObjRef.returnToTitle();
```

Description

DVD method; resumes playback after a menu has been displayed.

Parameters

None.

Example

This statement resumes playback after a menu has been displayed:

```
-- Lingo syntax
member(1).returnToTitle()

// JavaScript syntax
member(1).returnToTitle()
```

See also

DVD

revertToWorldDefaults

Usage

```
member(whichCastmember).revertToWorldDefaults()
```

Description

3D command; reverts the properties of the specified 3D cast member to the values stored when the member was first created. The member's `state` property must be 4 (loaded) or -1 (error) before this command can be used, otherwise a script error will occur.

This command differs from `resetWorld` in that the values used are taken from the state of the member when it was first created rather than from the state of the member when it was first loaded into memory.

Parameters

None.

Example

This statement reverts the properties of the cast member named Scene to the values stored when the member was first created:

```
-- Lingo
member("Scene").revertToWorldDefaults()

// Javascript
member("Scene").revertToWorldDefaults() ;
```

See also

resetWorld

rewind() (Sound Channel)

Usage

```
-- Lingo syntax
soundChannelObjRef.rewind()

// JavaScript syntax
soundChannelObjRef.rewind();
```

Description

Sound Channel method; interrupts the playback of the current sound in a sound channel and restarts it at its `startTime`.

If the sound is paused, it remains paused, with the `currentTime` set to the `startTime`.

Parameters

None.

Example

This statement restarts playback of the sound cast member playing in sound channel 1 from the beginning:

```
-- Lingo syntax
sound(1).rewind()

// JavaScript syntax
sound(1).rewind();
```

See also

Sound Channel, startTime

rewind() (Windows Media)

Usage

```
-- Lingo syntax
windowsMediaObjRef.rewind()

// JavaScript syntax
windowsMediaObjRef.rewind();
```

Description

Windows Media cast member or sprite method. Rewinds to the first frame of a Windows Media cast member or sprite.

Calling this method has no effect on the `mediaStatus`.

Parameters

None.

See also

`mediaStatus (RealMedia, Windows Media)`, `Windows Media`

rewind() (Animated GIF, Flash)

Usage

```
-- Lingo syntax
animGifSpriteRef.rewind()

// JavaScript syntax
animGifSpriteRef.rewind();
```

Description

Command; returns a Flash or animated GIF movie sprite to frame 1 when the sprite is stopped or when it is playing.

Parameters

None.

Example

The following frame script checks whether the Flash movie sprite in the sprite the behavior was placed in is playing and, if so, continues to loop in the current frame. When the movie is finished, the sprite rewinds the movie (so the first frame of the movie appears on the Stage) and lets the playhead continue to the next frame.

```
-- Lingo syntax
property spriteNum

    on exitFrame
    if sprite(spriteNum).playing then
        _movie.go(_movie.frame)
    else
        sprite(spriteNum).rewind()
        _movie.updatestage()
    end if
end

// JavaScript syntax
function exitFrame(me) {
    var plg = sprite(this.spriteNum).playing;
    if (plg == 1) {
        _movie.go(_movie.frame);
    } else {
        sprite(this.spriteNum).rewind();
        _movie.updateStage();
    }
}
```

rollOver()

Usage

```
-- Lingo syntax
_movie.rollOver({intSpriteNum})

// JavaScript syntax
_movie.rollOver({intSpriteNum});
```

Description

Movie method; indicates whether the pointer (cursor) is currently over the bounding rectangle of a specified sprite (`TRUE` or 1) or not (`FALSE` or 0).

The `rollOver()` method is typically used in frame scripts and is useful for creating handlers that perform an action when the user places the pointer over a specific sprite.

If the user continues to roll the mouse, the value of `rollOver()` can change while a script is running a handler, and can result in unexpected behavior. You can make sure that a handler uses a consistent rollover value by assigning `rollOver()` to a variable when the handler starts.

When the pointer is over an area of the Stage where a sprite previously appeared, `rollOver()` still occurs and reports the sprite as still being there. Avoid this behavior by not performing rollovers over these locations, or by moving the sprite above the menu bar before removing it.

Parameters

`intSpriteNum` Optional. An integer that specifies the sprite number.

Example

This statement changes the content of the field cast member Message to "This is the place." when the pointer is over sprite 6:

```
-- Lingo syntax
if (_movie.rollOver(6)) then
    member("Message").text = "This is the place."
end if

// JavaScript syntax
if (_movie.rollOver(6)) {
    member("Message").text = "This is the place.";
}
```

The following handler sends the playhead to different frames when the pointer is over certain sprites on the Stage. It first assigns the `rollOver` value to a variable. This lets the handler use the rollOver value that was in effect when the rollover started, regardless of whether the user continues to move the mouse.

```
-- Lingo syntax
on exitFrame
    currentSprite = _movie.rollOver()
    case currentSprite of
        1: _movie.go("Left")
        2: _movie.go("Middle")
        3: _movie.go("Right")
    end case
end exitFrame

// JavaScript syntax
function exitFrame() {
```

```
    var currentSprite = _movie.rollOver();
    switch (currentSprite) {
        case 1: _movie.go("Left");
            break;
        case 2: _movie.go("Middle");
            break;
        case 3: _movie.go("Right");
            break;
    }
}
```

See also

Movie

rootMenu()

Usage

```
-- Lingo syntax
dvdObjRef.rootMenu()

// JavaScript syntax
dvdObjRef.rootMenu();
```

Description

DVD method; displays the root menu.

Parameters

None.

Example

This statement displays the root menu:

```
-- Lingo syntax
member(1).rootMenu()

// JavaScript syntax
member(1).rootMenu();
```

See also

DVD

rotate

Usage

```
member(whichCastmember).node(whichNode).rotate(xAngle, yAngle, zAngle {, relativeTo})
member(whichCastmember).node(whichNode).rotate(rotationVector {, relativeTo})
member(whichCastmember).node(whichNode).rotate(position, axis, angle {, relativeTo})
transform.rotate(xAngle, yAngle, zAngle {, relativeTo})
transform.rotate(rotationVector {, relativeTo})
transform.rotate(position, axis, angle {, relativeTo})
```

Description

3D command; applies a rotation after the current positional, rotational, and scale offsets held by the node's transform object or the directly referenced transform object. The rotation must be specified as a set of three angles, each of which specify an angle of rotation about the three corresponding axes. These angles may be specified explicitly in the form of *xAngle*, *yAngle*, and *zAngle*, or by a *rotationVector*, where the *x* component of the vector corresponds to the rotation about the X axis, *y* about Y axis, and *z* about Z axis. Alternatively, the rotation may also be specified as a rotation about an arbitrary axis passing through a point in space.

Parameters

xAngle Required if applying a rotation using *x*-, *y*-, and *z*-axes. Specifies the angle of rotation about the *x*-axis.

yAngle Required if applying a rotation using *x*-, *y*-, and *z*-axes. Specifies the angle of rotation about the *y*-axis.

zAngle Required if applying a rotation using *x*-, *y*-, and *z*-axes. Specifies the angle of rotation about the *z*-axis.

rotationVector Required if applying a rotation using a vector. Specifies the vector that contains the angles to apply.

position Required if applying a rotation about an arbitrary axis passing through a point in space. Specifies position in space.

axis Required if applying a rotation about an arbitrary axis passing through a point in space. Specifies the axis passing through the specified position *position*.

angle Required if applying a rotation about an arbitrary axis passing through a point in space. Specifies the amount of rotation about the axis *axis*.

relativeTo Optional. Specifies which coordinate system axes are used to apply the desired rotational changes. The *relativeTo* parameter can have any of the following values:

- `#self` applies the increments relative to the node's local coordinate system (the X, Y and Z axes specified for the model during authoring). This value is used as the default if you use the `rotate` command with a node reference and the *relativeTo* parameter is not specified.
- `#parent` applies the increments relative to the node's parent's coordinate system. This value is used as the default if you use the `rotate` command with a transform reference and the *relativeTo* parameter is not specified.
- `#world` applies the increments relative to the world coordinate system. If a model's parent is the world, than this is equivalent to using `#parent`.
- `nodeReference` allows you to specify a node to base your rotation upon, the command applies the increments relative to the coordinate system of the specified node.

Example

The following example first rotates the model named Moon about its own Z axis (rotating it in place), then it rotates that same model about its parent node, the model named Earth (causing Moon to move orbitally about Earth).

```
member("Scene").model("Moon").rotate(0,0,15)
member("Scene").model("Moon").rotate(vector(0, 0, 5), member("Scene").model("Moon"))
```

The following example rotates the model Ball around a position in space occupied by the model named Pole. The effect is that the model Ball moves in orbit around Pole in the x-y plane.

```
polePos = member("3d Scene").model("Pole").worldPosition
member("3d Scene").model("Ball").rotate(polePos, vector(0,0,1), 5, #world)
```

See also

pointAt, preRotate, rotation (transform), rotation (engraver shader), rotation (backdrop and overlay), preScale(), transform (property)

run

Usage

```
run(MUIObject)
```

Description

This command displays a general purpose modal dialog box created from an instance of the MUI Xtra.

Before Director can open the dialog box, use the Initialize command to define the dialog box.

To open a non-modal dialog box, use the WindowOperation command with the #show option. This command allows other Lingo to run in the movie while the non-modal dialog box is open.

Example

This handler checks whether the object MUIObject exists and displays a general purpose dialog box from UIObject if it is:

```
--Lingo syntax
on runDialog
   global MUIObject
   if objectP(MUIObject) then
      run(MUIObject)
   end if
end
```

runMode

Usage

```
the runMode
```

Description

Function; returns a string indicating the mode in which the movie is playing. Possible values are as follows:

- `Author`—The movie is running in Director.
- `Projector`—The movie is running as a projector.
- `BrowserPlugin`—The movie is running as a Shockwave Player plug-in or other scripting environment, such as LiveConnect or ActiveX.

The safest way to test for particular values in this property is to use the `contains` operator. This helps avoid errors and allows partial matches.

Parameters

None.

Example

This statement determines whether or not external parameters are available and obtains them if they are:

```
--Lingo syntax
if the runMode contains "Plugin" then
    -- decode the embed parameter
    if externalParamName(swURL) = swURL then
        put externalParamValue(swURL) into myVariable
    end if
```

```
end if

// JavaScript syntax
if (_system.environmentPropList.runMode.indexOf("Plugin") >=0) {
    // decode the embed parameter
    if (_player.externalParamName(swURL) == swURL) {
        myVariable = _player.externalParamValue(swURL);
    }
}
```

See also

environmentPropList, platform

save castLib

Usage

```
castLib(whichCast).save()
save castLib whichCast {,pathName&newFileName}
```

Description

Command; saves changes to the cast in the cast's original file or in a new file. Further operations or references to the cast use the saved cast member.

This command does not work with compressed files.

The `save CastLib` command doesn't support URLs as file references.

Parameters

pathName&newFileName Optional. Specifies the path and file name to save to. If omitted, the original cast must be linked.

Example

This statement causes Director to save the revised version of the Buttons cast in the new file UpdatedButtons in the same folder:

```
-- Lingo
castLib("Buttons").save(the moviePath & "UpdatedButtons.cst")

// Javascript
castLib("Buttons").save(_movie.path & "UpdatedButtons.cst") ;
```

See also

@ (pathname)

saveMovie()

Usage

```
-- Lingo syntax
_movie.saveMovie({stringFilePath})

// JavaScript syntax
_movie.saveMovie({stringFilePath});
```

Description

Movie method; saves the current movie.

Including the optional *stringFilePath* parameter saves the movie to the file specified. This method does not work with compressed files. The specified filename must include the .dir file extension.

The `saveMovie()` method doesn't support URLs as file references.

Parameters

stringFilePath Optional. A string that specifies the path to and name of the file to which the movie is saved.

Example

This statement saves the current movie in the Update file:

```
-- Lingo syntax
_movie.saveMovie(_movie.path & "Update.dir")

// JavaScript syntax
_movie.saveMovie(_movie.path + "Update.dir");
```

See also

Movie

scale (command)

Usage

```
member(whichCastmember).node(whichNode).scale(xScale, yScale, zScale)
member(whichCastmember).node(whichNode).scale(uniformScale)
transform.scale(xScale, yScale, zScale)
transform.scale(uniformScale)
```

Description

3D transform command; applies a scaling after the current positional, rotational, and scale offsets held by a referenced node's transform or the directly referenced transform. The scaling must be specified as either a set of three scalings along the corresponding axes or as a single scaling to be applied uniformly along all axes. You can specify the individual scalings using the *xScale*, *yScale* and *zScale* parameters, otherwise you can specify the uniform scaling amount using the *uniformScale* parameter.

A node can be a camera, group, light or model object. Using the `scale` command adjusts the referenced node's `transform.scale` property, but it does not have any visual effect on lights or cameras as they do not contain geometry.

The scaling values provided must be greater than zero.

Parameters

xScale Required if specifying three scalings. Specifies the scale along the *x*-axis.

yScale Required if specifying three scalings. Specifies the scale along the *y*-axis.

zScale Required if specifying three scalings. Specifies the scale along the *z*-axis.

uniformScale Required if specifying a single, uniform scaling. Specifies the uniform scaling.

Example

This example first displays the `transform.scale` property for the model named Moon, then it scales the model using the `scale` command, and finally, it displays the resulting `transform.scale` value.

```
-- Lingo
put member("Scene").model("Moon").transform.scale

// Javascript
put member("Scene").getProp("model", i).transform.scale ;
-- vector( 1.0000, 1.0000, 1.0000)
```

This statement scales the model named Pluto uniformly along all three axes by 0.5, resulting in the model displaying at half of its size.

```
-- Lingo
member("Scene").model("Pluto").scale(0.5)

// Javascript
member("Scene").getPropRef("model", a).scale(0.5);
// where a is the number index of the model "Pluto"
```

This statement scales the model named Oval in a nonuniform manner, scaling it along its *z*-axis but not its *x*- or *y*-axes.

```
-- Lingo
member("Scene").model("Pluto").scale(0.0, 0.0, 0.5)

// Javascript
member("Scene").getPropRef("model", a).scale(0.0, 0.0, 0.5);
// where a is the number index of the model "Pluto"
```

See also

transform (property), preScale(), scale (transform)

script()

Usage

```
-- Lingo syntax
script(memberNameOrNum {, castNameOrNum})

// JavaScript syntax
script(memberNameOrNum {, castNameOrNum});
```

Description

Top level function; creates a reference to a given cast member that contains a script, and optionally specifies the cast library that contains the member.

An error is returned if the given cast member does not contain a script, or if the given cast member does not exist.

Parameters

memberNameOrNum Required. A string that specifies the name of the cast member that contains a script, or an integer that specifies the index position of the cast member that contains a script.

castNameOrNum Optional. A string that specifies the name of the cast library that contains the member *memberNameOrNum*, or an integer that specifies the index position of the cast library that contains the member *memberNameOrNum*. If omitted, `script()` searches the first cast library.

Example

In Lingo only, these statements check whether a child object is an instance of the parent script Warrior Ant:

```
-- Lingo syntax
if (bugObject.script = script("Warrior Ant")) then
    bugObject.attack()
end if
```

This statement sets the variable `actionMember` to the script cast member Actions:

```
-- Lingo syntax
actionMember = script("Actions")

// JavaScript syntax
var actionMember = script("Actions");
```

scrollByLine()

Usage

```
-- Lingo syntax
memberObjRef.scrollByLine(amount)

// JavaScript syntax
memberObjRef.scrollByLine(amount);
```

Description

Command; scrolls the specified field or text cast member up or down by a specified number of lines. Lines are defined as lines separated by carriage returns or by wrapping.

Parameters

`amount` Required. Specifies the number of lines to scroll. When *`amount`* is positive, the field scrolls down. When *`amount`* is negative, the field scrolls up.

Example

This statement scrolls the field cast member Today's News down five lines:

```
--Lingo syntax
member("Today's News").scrollbyline(5)

// JavaScript syntax
member("Today's News").scrollbyline(5);
```

This statement scrolls the field cast member Today's News up five lines:

```
--Lingo syntax
member("Today's News").scrollByLine(-5)

// JavaScript syntax
member("Today's News").scrollByLine(-5);
```

scrollByPage()

Usage

```
-- Lingo syntax
```

```
memberObjRef.scrollByPage(amount)

// JavaScript syntax
memberObjRef.scrollByPage(amount);
```

Description

Command; scrolls the specified field or text cast member up or down by a specified number of pages. A page is equal to the number of lines of text visible on the screen.

Parameters

amount Required. Specifies the number of pages to scroll. When *amount* is positive, the field scrolls down. When *amount* is negative, the field scrolls up.

Example

This statement scrolls the field cast member Today's News down one page:

```
--Lingo syntax
member("Today's News").scrollbypage(1)

// JavaScript syntax
member("Today's News").scrollbypage(1);
```

This statement scrolls the field cast member Today's News up one page:

```
--Lingo syntax
member("Today's News").scrollbypage(-1)

// JavaScript syntax
member("Today's News").scrollbypage(-1);
```

See also

scrollTop

seek()

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.seek(milliseconds)

// JavaScript syntax
memberOrSpriteObjRef.seek(milliseconds);
```

Description

RealMedia sprite or cast member method; changes the media stream's playback location to the location specified by the number of milliseconds from the beginning of the stream. The `mediaStatus` value usually becomes `#seeking` and then `#buffering`.

You can use this method to initiate play at points other than the beginning of the RealMedia stream, or to jump forward or backward in the stream. The integer specified in *milliseconds* is the number of milliseconds from the beginning of the stream; thus, to jump backward, you would specify a lower number of milliseconds, not a negative number.

If the `seek` command is called when `mediaStatus` is `#paused`, the stream rebuffers and returns to `#paused` at the new location specified by `seek`. If `seek` is called when `mediaStatus` is `#playing`, the stream rebuffers and automatically begins playing at the new location in the stream. If `seek` is called when `mediaStatus` is `#closed`, nothing happens.

If you attempt to seek beyond the stream's `duration` value, the integer argument you specify is clipped to the range from 0 to the duration of the stream. You cannot jump ahead into a RealMedia sprite that is streaming live content.

The statement `x.seek(n)` is the same as `x.currentTime = n`, and either of these calls will cause the stream to be rebuffered.

Parameters

`milliseconds` Required. An integer that specifies the number of milliseconds from the beggining of the stream.

Example

The following examples set the current playback position of the stream to 10,000 milliseconds (10 seconds):

```
-- Lingo syntax
sprite(2).seek(10000)
member("Real").seek(10000)

// JavaScript syntax
sprite(2).seek(10000);
member("Real").seek(10000);
```

See also

`duration (RealMedia, SWA)`, `currentTime (RealMedia)`, `play() (RealMedia, SWA, Windows Media)`, `pause() (RealMedia, SWA, Windows Media)`, `stop() (RealMedia, SWA, Windows Media)`, `mediaStatus (RealMedia, Windows Media)`

selectAtLoc()

Usage

```
-- Lingo syntax
dvdObjRef.selectAtLoc(point(x, y))

// JavaScript syntax
dvdObjRef.selectAtLoc(point(x, y));
```

Description

DVD method; moves focus to the button under a specified point.

This method has the same functionality as a mouse hovering over a button.

Parameters

`point(x, y)` Required. A point in Stage coordinates that specifies the location under which a button is given focus.

Example

This statement moves focus to the button under a specified point:

```
-- Lingo syntax
member(10).selectAtLoc(point(50, 75))
```

```
// JavaScript syntax
member(10).selectAtLoc(point(50, 75));
```

See also

DVD

selectButton()

Usage

```
-- Lingo syntax
dvdObjRef.selectButton(intButton)

// JavaScript syntax
dvdObjRef.selectButton(intButton);
```

Description

DVD method; selects a specified button.

This method returns `0` if successful.

Note: *This method is not supported in Mac®-Intel®.*

Parameters

`intButton` Required. An integer that specifies the button that is given focus.

Example

This statement selects button 5:

```
-- Lingo syntax
sprite(11).selectButton(5)

// JavaScript syntax
sprite(11).selectButton(5);
```

See also

DVD

selectButtonRelative()

Usage

```
-- Lingo syntax
dvdObjRef.selectButtonRelative(direction)

// JavaScript syntax
dvdObjRef.selectButtonRelative(direction);
```

Description

DVD method; selects a button relative to the current button position in the menu.

Parameters

direction Required. A symbol (Lingo) or a string (JavaScript syntax) that specifies the direction to move from the current button position. Valid values are `left` or `right`.

Note: *This method is not supported in Mac®-Intel®.*

Example

This statement specifies the button to the left of the current button:

```
-- Lingo syntax
member(12).member.selectButtonRelative(#left)

// JavaScript syntax
member(12).member.selectButtonRelative("left");
```

See also

DVD

selection() (function)

Usage

```
the selection
```

Description

Function; returns a string containing the highlighted portion of the current editable field. This function is useful for testing what a user has selected in a field.

The `selection` function only indicates which string of characters is selected; you cannot use `selection` to select a string of characters.

Parameters

None.

Example

This statement checks whether any characters are selected and, if none are, displays the alert "Please select a word.":

```
if the selection = EMPTY then alert "Please select a word."
```

See also

selStart, selEnd

sendAllSprites()

Usage

```
-- Lingo syntax
_movie.sendAllSprites(stringEventMessage {, args})

// JavaScript syntax
_movie.sendAllSprites(stringEventMessage {, args});
```

Description

Movie method; sends a designated message to all sprites, not just the sprite that was involved in the event. As with any other message, the message is sent to every script attached to the sprite, unless the `stopEvent()` method is used.

For best results, send the message only to those sprites that will properly handle the message through the `sendSprite()` method. No error will occur if the message is sent to all the sprites, but performance may decrease. There may also be problems if different sprites have the same handler in a behavior, so avoid conflicts by using unique names for messages that will be broadcast.

After the message has been passed to all behaviors, the event follows the regular message hierarchy: cast member script, frame script, then movie script.

When you use the `sendAllSprites()` method, be sure to do the following:

- Replace *`stringEventMessage`* with the message.
- Replace *`args`* with any arguments to be sent with the message.

If no sprite has an attached behavior containing the given handler, `sendAllSprites()` returns `FALSE`.

Parameters

`stringEventMessage` Required. A string that specifies the message to send to all sprites.

`args` Optional. An argument or arguments to send with the message.

Example

This handler sends the custom message `allSpritesShouldBumpCounter` and the argument 2 to all sprites when the user clicks the mouse:

```
-- Lingo syntax
on mouseDown me
    _movie.sendAllSprites(#allspritesShouldBumpCounter, 2)
end

// JavaScript syntax
function mouseDown() {
    _movie.sendAllSprites("allspritesShouldBumpCounter", 2);
}
```

See also

`Movie`, `sendSprite()`, `stopEvent()`

sendEvent

Usage

```
member(whichCastmember).sendEvent(#eventName, arg1, arg2,...)
```

Description

3D command; sends an event and an arbitrary number of arguments to all scripts registered to receive the event. Use `registerForEvent()`, or `setCollisionCallback()` to register scripts for events.

Parameters

`eventName` Required. Specifies the name of the event to send.

arg1, arg2, ... Required. One or more arguments that are sent with the event *eventName*.

Example

The first line in this example creates an instance of a parent script named "tester". The second line sets the handler of the script instance, jumpPluto, as the handler to be called when the #jump event is sent. The third line registers a movie script handler named jumpMars as another handler to be called when the #jump event is sent. The fourth line sends the #jump event. The handlers #jumpMars in a movie script and #jumpPluto are called, along with any other handlers registered for the #jump event. A script instance value of 0 indicates that you are registering a handler of a movie script, as opposed to a handler of a behavior instance or of a child of a parent script.

```
t = new (script "tester")
member("scene").registerForEvent(#jump, #jumpPluto, t)
member("scene").registerForEvent(#jump, #jumpMars, 0)
member("scene").sendEvent(#jump)
```

See also

registerScript(), registerForEvent(), setCollisionCallback()

sendSprite()

Usage

```
-- Lingo syntax
_movie.sendSprite(spriteNameOrNum, event {, args})

// JavaScript syntax
_movie.sendSprite(spriteNameOrNum, event {, args});
```

Description

Movie method; sends a message to all scripts attached to a specified sprite.

Messages sent using sendSprite() are sent to each of the scripts attached to the sprite. The messages then follow the regular message hierarchy: cast member script, frame script, and movie script.

If the given sprite does not have an attached behavior containing the given handler, sendSprite() returns FALSE.

Parameters

spriteNameOrNum Required. A string or an integer that specifies the name or number of the sprite that will receive the event.

event Required. A symbol or string that specifies the event to send to the specified sprite.

args Optional. An argument or arguments to send with the message.

Example

This handler sends the custom message bumpCounter and the argument 2 to sprite 1 when the user clicks:

```
-- Lingo syntax
on mouseDown me
    _movie.sendSprite(1, #bumpCounter, 2)
end

// JavaScript syntax
function mouseDown() {
    _movie.sendSprite(1, "bumpCounter", 2);
```

```
}
```

See also

Movie

setAlpha()

Usage

```
imageObject.setAlpha(alphaLevel)
imageObject.setAlpha(alphaImageObject)
```

Description

Function; sets the alpha channel of an image object to a flat alphaLevel or to an existing *alphaImageObject*. The *alphaLevel* must be a number from 0–255. Lower values cause the image to appear more transparent. Higher values cause the image to appear more opaque. The value 255 has the same effect as a value of zero. In order for the *alphaLevel* to have effect, the `useAlpha()` of the image object must be set to `TRUE`.

The image object must be 32-bit. If you specify an alpha image object, it must be 8-bit. Both images must have the same dimensions. If these conditions are not met, `setAlpha()` has no effect and returns FALSE. The function returns TRUE when it is successful.

Example

The following Lingo statement makes the image of the bitmap cast member Foreground opaque and disables the alpha channel altogether. This is a good method for removing the alpha layer from an image:

```
member("Foreground").image.setAlpha(255)
member("Foreground").image.useAlpha = FALSE
```

This Lingo gets the alpha layer from the cast member Sunrise and places it into the alpha layer of the cast member Sunset:

```
tempAlpha = member("Sunrise").image.extractAlpha()
member("Sunset").image.setAlpha(tempAlpha)
```

See also

useAlpha, extractAlpha()

setaProp

Usage

```
setaProp list, listProperty, newValue
setaProp (childObject, listProperty, newValue)
list.listProperty = newValue
list[listProperty] = newValue
childObject.listProperty = newValue
```

Description

Command; replaces the value assigned to *listProperty* with the value specified by *newValue*. The `setaProp` command works with property lists and child objects. Using `setaProp` with a linear list produces a script error.

- For property lists, `setaProp` replaces a property in the list specified by *`list`*. When the property isn't already in the list, Lingo adds the new property and value.
- For child objects, `setaProp` replaces a property of the child object. When the property isn't already in the object, Lingo adds the new property and value.
- The `setaProp` command can also set `ancestor` properties.

Parameters

`listProperty` Required. A symbol (Lingo only) or a string that specifies the name of the property whose value is changing.

`newValue` Required. The new value for the *`listProperty`* property.

Example

These statements create a property list and then adds the item `#c:10` to the list:

```
newList = [#a:1, #b:5]
put newList
-- [#a:1, #b:5]
setaProp newList, #c, 10
put newList
```

Using the dot operator, you can alter the property value of a property already in a list without using `setaProp`:

```
newList = [#a:1, #b:5]
put newList
-- [#a:1, #b:5]
newList.b = 99
put newList
-- [#a:1, #b:99]
```

Note: *To use the dot operator to manipulate a property, the property must already exist in the list, child object, or behavior.*

See also

`ancestor`, `property`, `. (dot operator)`

setAt

Usage

```
setAt list, orderNumber, value
list[orderNumber] = value
```

Description

Command; replaces the item specified by *`orderNumber`* with the value specified by *`value`* in the list specified by *`list`*. When *`orderNumber`* is greater than the number of items in a property list, the `setAt` command returns a script error. When *`orderNumber`* is greater than the number of items in a linear list, Director expands the list's blank entries to provide the number of places specified by *`orderNumber`*.

Example

This handler assigns a name to the list [12, 34, 6, 7, 45], replaces the fourth item in the list with the value 10, and then displays the result in the Message window:

```
--Lingo
```

```
on enterFrame
    set vNumbers = [12, 34, 6, 7, 45]
    setAt vnumbers, 4, 10
    put vNumbers
end enterFrame

// Javascript
function enterFrame
{
    vNumbers =list(12, 34, 6, 7, 45) ;
    vNumbers.setAt(4, 10) ;
    put (vNumbers);
}
```

When the handler runs, the Message window displays the following:

```
[12, 34, 6, 10, 45]
```

You can perform this same operation may be done using bracket access to the list in the following manner:

```
--Lingo
on enterFrame
    set vNumbers = [12, 34, 6, 7, 45]
    vnumbers[4] = 10
    put vNumbers
end enterFrame

// Javascript
function enterFrame
{
    vNumbers = list(12, 34, 6, 7, 45);
    vnumbers[4] = 10 ;
    put (vNumbers);
}
```

When the handler runs, the Message window displays the following:

```
[12, 34, 6, 10, 45]
```

See also

[ ] (bracket access)

setCallback()

Usage

```
-- Lingo syntax
spriteObjRef.setCallback(actionScriptObject, ASEventName, #LingoHandlerName,
lingoScriptObject)

// JavaScript syntax
spriteObjRef.setCallback(actionScriptObject, ASEventName, #LingoHandlerName,
lingoScriptObject);
```

Description

Flash command; this command can be used as a sprite or a global method to define a Lingo callback handler for a particular event generated by the specified object. When ActionScript triggers the event in the object, that event is redirected to the given Lingo handler, including all arguments that are passed with the event.

If the ActionScript object was originally created within a Flash sprite, use the *flashSpriteReference* syntax. If the object was originally created globally, use the global syntax.

Note: *If you have not imported any Flash cast members, you must manually add the Flash Asset Xtra to your movie's Xtra list in order for global Flash commands to work correctly. You add Xtra extensions to the Xtra list by choosing Modify > Movie > Xtras. For more information about managing Xtra extensions for distributed movies, see the Using Director topics in the Director Help Panel.*

Parameters

actionScriptObject Required. Specifies the ActionScript object that contains the event *ASEventName*.

ASEventName Required. Specifies the ActionScript event that occurs.

lingoHandlerName Required. Specifies the Lingo handler that handles the event *ASEventName*.

lingoScriptObject Required. Specifies the Lingo script object that contains the handler *lingoHandlerName*.

Example

This statement sets a the Lingo handler named `myOnStatus` in the Lingo script object me to be called when an `onStatus` event is generated by the ActionScript object `tLocalConObject` in the Flash movie in sprite 3:

```
Lingo syntax
sprite(3).setCallback(tLocalConObject, "onStatus", #myOnStatus, me)

// JavaScript syntax
sprite(3).setCallback(tLocalConObject, "onStatus", symbol("myOnStatus"), me);
```

The following statements create a new global XML object and create a `callback` handler that parses XML data when it arrives. The third line loads an XML file. The `callback` handler is included as well.

```
-- Lingo syntax
gXMLCB = newObject("XML")
setCallback( gXMLCB, "onData", #dataFound, 0 )
gXMLCB.load( "myfile.xml")

-- Callback handler invoked when xml data arrives
on dataFound me, obj, source
 obj.parseXML(source)
 obj.loaded = 1
 obj.onload(TRUE)
end dataFound

// JavaScript syntax
gXMLCB = newObject("XML");
setCallback( gXMLCB, "onData", symbol("dataFound"), 0 );
gXMLCB.load( "myfile.xml" );

// Callback handler invoked when xml data arrives
function dataFound(me, obj, source) {
 obj.parseXML(source);
 obj.loaded = 1;
 obj.onload(1);
}
```

See also

`newObject()`, `clearAsObjects()`

setCollisionCallback()

Usage

```
member(whichCastmember).model(whichModel).collision.setCollisionCallback (#handlerName,
scriptInstance)
```

Description

3D collision command; registers a specified handler, in a given script instance, to be called when *whichModel* is involved in a collision.

This command works only if the model's `collision.enabled` property is `TRUE`. The default behavior is determined by the value of `collision.resolve`, you can override it using the `collision.resolveA` and/or the `collision.resolveB` commands. Do not use the `updateStage` command in the specified handler.

This command is a shorter alternative to using the `registerScript` command for collisions, but there is no difference in the overall result. This command can be considered to perform a small subset of the `registerScript` command functionality.

Parameters

handlerName Required. Specifies the handler called when a model is involved in a collision.

scriptInstance Required. Specifies the script instance that contains the handler specified by *handlerName*.

Example

This statement causes the `#bounce` handler in the cast member colScript to be called when the model named Sphere collides with another model:

```
member("3d world").model("Sphere").collision.setCollisionCallback(#bounce,
member("colScript"))
```

See also

`collisionData`, `collision (modifier)`, `resolve`, `resolveA`, `resolveB`, `registerForEvent()`, `registerScript()`, `sendEvent`

setFilterMask()

Usage

```
-- Lingo syntax
fileioObjRef.setFilterMask(stringMask)

// JavaScript syntax
fileioObjRef.setFilterMask(stringMask);
```

Description

Fileio method; Sets the filter mask for the *Files of type* field of a dialog box to specify the type of files displayed when the dialog box opens.

Parameters

stringMask Required. A string that specifies the filter mask.

Example

```
-- Lingo syntax
objFileio = new xtra("fileio")
objFileio.openFile(stringFileName, intMode)
objFileio.setFilterMask(stringMask)

 // JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile(stringFileName, intMode) ;
objFileio.setFilterMask(stringMask) ;
```

See also

Fileio

setFinderInfo()

Usage

```
-- Lingo syntax
fileioObjRef.setFinderInfo(stringAttrs)

// JavaScript syntax
fileioObjRef.setFinderInfo(stringAttrs)
```

Description

Fileio method (Mac only); Sets the finder information for an open file.

Parameters

stringAttrs Required. A string that specifies the finder information.

Example

```
-- Lingo syntax
objFileio = new xtra("fileio")
objFileio.openFile(stringFileName, intMode)
objFileio.setFinderInfo(stringAttrs)

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile(stringFileName, intMode) ;
objFileio.setFinderInfo(stringAttrs) ;
```

See also

Fileio

setFlashProperty()

Usage

```
-- Lingo syntax
spriteObjRef.setFlashProperty(targetName, #property, newValue)

// JavaScript syntax
spriteObjRef.setFlashProperty(targetName, #property, newValue);
```

Description

Function; allows Lingo to call the Flash action script function `setProperty()` on the given Flash sprite. Use the `setFlashProperty()` function to set the properties of movie clips or levels within a Flash movie. This is similar to setting sprite properties within Director.

To set a global property of the Flash sprite, pass an empty string as the *`targetName`*. You can set the global Flash properties: `#focusRect` and `#spriteSoundBufferTime`.

See the Flash documentation for descriptions of these properties.

Parameters

`targetName` Required. Specifies the name of the movie clip or level whose property you want to set within the given Flash sprite.

`property` Required. Specifies the name of the property to set. You can set the following movie clip properties: `#posX`, `#posY`, `#scaleX`, `#scaleY`, `#visible`, `#rotate`, `#alpha`, and `#name`.

`newValue` Required. Specifies the new value.

Example

This statement sets the value of the `#rotate` property of the movie clip Star in the Flash member in sprite 3 to 180:

```
-- Lingo syntax
sprite(3).setFlashProperty("Star", #rotate, 180)

// JavaScript syntax
sprite(3).setFlashProperty("Star", symbol("rotate"), 180);
```

See also

`getFlashProperty()`

setNewLineConversion()

Usage

```
-- Lingo syntax
fileioObjRef.setNewLineConversion(intOnOff)

// JavaScript syntax
fileioObjRef.setNewLineConversion(intOnOff)
```

Description

Fileio method (Mac only); Specifies whether automatic conversion of new line characters is on or off.

Parameters

`intOnOff` Required. An integer that specifies whether automatic conversion is on or off. Valid values include 0 (off) or 1 (on).

Example

```
-- Lingo syntax
objFileio = new xtra("fileio")
objFileio.openFile(stringFileName, intMode)
objFileio.setNewLineConversion(intOnOff)
```

```
 // JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile(stringFileName, intMode) ;
objFileio.setNewLineConversion(intOnOff) ;
```

See also

Fileio

setPixel()

Usage

```
-- Lingo syntax
imageObjRef.setPixel(x, y, colorObjOrIntValue)
imageObjRef.setPixel(point(x, y), colorObjOrIntValue)

// JavaScript syntax
imageObjRef.setPixel(x, y, colorObjOrIntValue);
imageObjRef.setPixel(point(x, y), colorObjOrIntValue);
```

Description

Image method. Sets the color value of the pixel at a specified point in a given image.

If setting many pixels to the color of another pixel with `getPixel()`, it is faster to set them as integers.

For best performance with color objects, use an indexed color object with 8-bit or lower images, and use an RGB color object with 16-bit or higher images.

This method returns FALSE if the specified pixel falls outside the specified image.

To see an example of this method used in a completed movie, see the Imaging movie in the Learning/Lingo folder inside the Director application folder.

Parameters

x Required if specifying a pixel using `x` and `y` coordinates. An integer that specifies the `x` coordinate of the pixel.

y Required if specifying a pixel using `x` and `y` coordinates. An integer that specifies the `y` coordinate of the pixel.

`point(`*x, y*`)` Required if specifying a pixel using a point. A point that specifies the pixel.

colorObjOrIntValue Required if setting the color to a color object or an integer value. A reference to a color object that specifies the color of the pixel, or an integer that specifies the color value of the pixel.

Example

These statements sets the color of the pixel at point (20, 20) in member Image stage to red.

```
-- Lingo
objImage = _movie.stage.image
objImage.setPixel(20, 20 , rgb(255,0,0))
put (objImage)

-- Javascript
var objImage = _movie.stage.image ;
objImage.setPixel(20, 20 , color(255,0,0)) ;
put (objImage) ;
```

See also

color(), draw(), fill(), getPixel(), image()

setPlayList()

Usage

```
-- Lingo syntax
soundChannelObjRef.setPlayList(linearListOfPropLists)

// JavaScript syntax
soundChannelObjRef.setPlayList(linearListOfPropLists);
```

Description

Sound Channel method; sets or resets the playlist of a sound channel.

This method is useful for queueing several sounds at once.

To see an example of `setPlaylist()` used in a completed movie, see the Sound Control movie in the Learning/Lingo folder inside the Director application folder.

Parameters

`linearListOfPropLists` Required. A linear list of property lists that specifies parameters of a playlist. You can specify these parameters for each sound to be queued:

Property	Description
#member	The sound cast member to queue. This property must be provided; all others are optional.
#startTime	The time within the sound at which playback begins, in milliseconds. The default is the beginning of the sound. See `startTime`.
#endTime	The time within the sound at which playback ends, in milliseconds. The default is the end of the sound. See `endTime`.
#loopCount	The number of times to play a loop defined with `#loopStartTime` and `#loopEndTime`. The default is 1. See `loopCount`.
#loopStartTime	The time within the sound to begin a loop, in milliseconds. See `loopStartTime`.
#loopEndTime	The time within the sound to end a loop, in milliseconds. See `loopEndTime`.
#preloadTime	The amount of the sound to buffer before playback, in milliseconds. See `preloadTime`.

Example

This handler queues and plays the cast member introMusic, starting at its 3-second point, with a loop repeated 5 times from the 8-second point to the 8.9-second point, and stopping at the 10-second point.

```
-- Lingo syntax
on playMusic
    sound(2).queue([#member:member("introMusic"), #startTime:3000, #endTime:10000,
#loopCount:5, #loopStartTime:8000, #loopEndTime:8900])
    sound(2).play()
end playMusic

// JavaScript syntax
function playMusic() {
```

```
    sound(2).queue(propList("member",member("introMusic"),"startTime",3000,
"endTime",10000, "loopCount",5, "loopStartTime",8000,"loopEndTime",8900));
    sound(2).play();
}
```

See also

endTime, getPlayList(), loopCount, loopEndTime, loopStartTime, Member, member, preLoadTime, queue(), Sound Channel, startTime

setPosition()

Usage

```
-- Lingo syntax
fileioObjRef.setPosition(intPosition)

// JavaScript syntax
fileioObjRef.setPosition(intPosition);
```

Description

Fileio method; Sets the position of a file.

Parameters

intPosition Required. An integer that specifies the new position of the file.

Example

```
-- Lingo syntax
objFileio = new xtra("fileio")
objFileio.openFile(stringFileName, intMode)
objFileio.setPosition(intPosition)

 // JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile(stringFileName, intMode) ;
objFileio.setPosition(intPosition) ;
```

See also

Fileio

setPref()

Usage

```
-- Lingo syntax
_player.setPref(stringPrefName, prefString)

// JavaScript syntax
_player.setPref(stringPrefName, prefString);
```

Description

Player method; writes a specified string to a specified file on the computer's local disk. The file is a standard text file.

After `setPref()` runs, if the movie is playing in a browser, a folder named Prefs is created in the Plug-In Support folder. The `setPref()` method can write only to that folder.

These are the locations where the file is created.

Windows Projector/Director ..\Documents and Settings\<username>\Application Data\Adobe\Director<version number>\Prefs\

Windows Shockwave ..\Documents and Settings\<username>\Application Data\Adobe\Shockwave Player <version number>\Prefs\

MAC Projector ~/Library/Application Support/Adobe/Director <version number>/Prefs

MAC Shockwave ~/Library/Application Support/Adobe/Shockwave Player<version number>/Prefs

Do not use this method to write to read-only media. Depending on the platform and version of the operating system, you may encounter errors or other problems.

In a browser, data written by `setPref()` is not private; any movie with Shockwave content can read this information and upload it to a server. Do not store confidential information using `setPref()`.

On Windows, `setPref()` fails if the user is a restricted user.

To see an example of `setPref()` used in a completed movie, see the Read and Write Text movie in the Learning/Lingo folder inside the Director application folder.

Parameters

`prefName` Required. A string that specifies the file to write to. The *`prefName`* parameter must be a valid filename. To make sure the filename is valid on all platforms, use no more than eight alphanumeric characters for the file name.

`prefValue` Required. A string that specifies the text to write to the file *`prefName`*.

Example

This handler saves the contents of the field cast member Text Entry in a file named DayWare settings:

```
-- Lingo syntax
on mouseUp me
    _player.setPref("DayWare", member("Text Entry").text)
end

// JavaScript syntax
function mouseUp() {
    _player.setPref("DawWare", member("Text Entry").text);
}
```

See also

`getPref()`, `Player`

setProp

Usage

```
setProp list, property, newValue
list.listProperty = newValue
```

```
list[listProperty] = newValue
```

Description

Command; in a list, replaces the value assigned to a specified property with a new value. If the list does not contain the specified property, `setProp` returns a script error.

The `setProp` command works with property lists only. Using `setProp` with a linear list produces a script error.

This command is similar to the `setaProp` command, except that `setProp` returns an error when the property is not already in the list.

Parameters

`property` Required. A symbol (Lingo only) or a string that specifies the property whose value is replaced by *`newValue`*.

`newValue` Required. The new value for the property specified by *`property`*.

Example

This statement changes the value assigned to the `age` property of property list x to 11:

```
--Lingo
setProp x, #age, 11

// Javascript
x[age] = 11
```

Using the dot operator, you can alter the property value of a property already in a list, exactly as above:

```
x.age = 11
```

See also

`setaProp`

setScriptList()

Usage

```
spriteReference.setScriptList(scriptList)
sprite(whichSprite).setScriptList(scriptList)
```

Description

This command sets the `scriptList` of the given sprite. The `scriptList` indicates which scripts are attached to the sprite and what the settings of each script property are. By setting this list, you can change which behaviors are attached to a sprite or change the behavior properties.

The list takes the form:

```
[ [ (whichBehaviorMember), " [ #property1: value, #property2: value, . . . ] ",
[(whichBehaviorMember), " [ #property1: value, #property2: value, . . . ] " ] ]
```

This command cannot be used during a score recording session. Use `setScriptList()` for sprites added during score recording after the score recording session has ended.

Parameters

`scriptList` Required. Specifies the script list for a given sprite.

See also

scriptList, value(), string()

settingsPanel()

Usage

```
-- Lingo syntax
spriteObjRef.settingsPanel({integerPanelIndex})

// JavaScript syntax
spriteObjRef.settingsPanel({integerPanelIndex});
```

Description

Flash sprite command; invokes the Flash Settings dialog box to the specified panel index. This is the same dialog box that can be opened by right-clicking (Windows) or Control-clicking (Mac) on a Flash movie playing in a browser.

The Settings dialog box will not be displayed if the Flash sprite's rectangle is not large enough to accommodate it.

If you want to emulate the Flash Player by invoking the Settings dialog box when a user right-clicks (Windows) or Control-clicks (Mac), you can use this command in a `mouseDown` handler that tests for the `rightMouseDown` property or the `controlDown` property.

In order to emulate the Flash Player by enabling the Settings dialog box in a Director movie running in a browser, you must first disable the Shockwave Player context menu that is available by right-clicking (Windows) or Control-clicking (Mac) on a movie with Shockwave content playing in a browser. For information on how to disable this menu, see the Using Director topics in the Director Help Panel.

Parameters

`integerPanelIndex` Optional. Specifies which panel to activate when the dialog box is opened. Valid values are 0, 1, 2, or 3. A value of 0 opens the dialog box showing the Privacy tab, a value of 1 opens it showing the Local Storage tab, a value of 2 opens it showing the Microphone tab, and a value of 3 opens it showing the Camera tab. The default panel index is 0.

Example

This statement opens the Flash Settings panel with the Local Storage tab active:

```
-- Lingo syntax
sprite(3).settingsPanel(1)

// JavaScript syntax
sprite(3).settingsPanel(1);
```

See also

on mouseDown (event handler), rightMouseDown, controlDown

setPref()

Usage

```
-- Lingo syntax
_player.setPref(stringPrefName, prefString)
```

```
// JavaScript syntax
_player.setPref(stringPrefName, prefString);
```

Description

Player method; writes the string specified by *prefString* in the file specified by *stringPrefName* on the computer's local disk.

The *stringPrefName* argument must be a valid filename. To make sure the filename is valid on all platforms, use no more than eight alphanumeric characters for the file name.

After the `setPref()` method runs, if the movie is playing in a browser, a folder named Prefs is created in the Plug-In Support folder. The `setPref()` method can write only to that folder.

If the movie is playing in a projector or Director, a folder is created in the same folder as the application. The folder receives the name *Prefs*.

Do not use this method to write to read-only media. Depending on the platform and version of the operating system, you may encounter errors or other problems.

This method does not perform any sophisticated manipulation of the string data or its formatting. You must perform any formatting or other manipulation in conjunction with `getPref()`; you can manipulate the data in memory and write it over the old file using `setPref()`.

In a browser, data written by `setPref()` is not private; any movie with Shockwave content can read this information and upload it to a server. Do not store confidential information using `setPref()`.

On Windows, the `setPref()` method fails if the user is a restricted user.

To see an example of `setPref()` used in a completed movie, see the Read and Write Text movie in the Learning/Lingo folder inside the Director application folder.

Parameters

stringPrefName Required. A string that specifies the name of the file to which the string *prefString* is written. The file is a standard text file.

prefString Required. The string to write to the file specified by *stringPrefName*.

Example

This handler saves the contents of the field cast member Text Entry in a file named DayWare settings:

```
-- Lingo syntax
on mouseUp me
    _player.setPref("CurPrefs", member("Text Entry").text)
end

// JavaScript syntax
function mouseUp() {
    _player.setPref("CurPrefs", member("Text Entry").text);
}
```

See also

`getPref()`, `Player`

setTrackEnabled()

Usage

```
-- Lingo syntax
spriteObjRef.setTrackEnabled(whichTrack, trueOrFalse)

// JavaScript syntax
spriteObjRef.setTrackEnabled(whichTrack, trueOrFalse);
```

Description

Command; determines whether the specified track in the digital video is enabled to play.

- When `setTrackEnabled` is `TRUE`, the specified track is enabled and playing.
- When `setTrackEnabled` is `FALSE`, the specified track is disabled and muted. For video tracks, this means they will no longer be updated on the screen.

To test whether a track is already enabled, test the `trackEnabled` sprite property.

Parameters

`whichTrack` Required. Specifies the track to test.

`trueOrFalse` Required. Specifies whether the track in the digital video is enabled (`TRUE`) or not (`FALSE`).

Example

This statement enables track 3 of the digital video assigned to sprite channel 8:

```
-- Lingo syntax
sprite(8).setTrackEnabled(3, TRUE)

// JavaScript syntax
sprite(8).setTrackEnabled(3, 1);
```

See also

`trackEnabled`

setVal()

Usage

```
<Void> Matrix.setVal(whichRow, whichColumn, valueToSet)
```

Description

Matrix method; sets the value of the specified element in the given matrix.

Parameters

`whichRow` Required. Row number of the element whose value is being set

`whichColumn` Required. Column number of the element whose value is being set

`valueToSet` Required. Float. Value to set for the Rowth row and Columnth column in the matrix

Example

The following function creates a matrix with 4 rows and 5 columns and sets the value of each element in the matrix to random values.

```
--Lingo
on randomMatrix()
  rows = 4
  cols = 5
  mat = newMatrix(rows,cols)
  repeat with i = 1 to rows
    repeat with j = 1 to cols
      mat.setVal(i,j,random(255))
    end repeat
  end repeat
  return mat
end
```

```
//Java Script
function randomMatrix()
{
  rows = 4;
  cols = 5;
  mat = newMatrix(rows,cols);
  for(i = 1 ; i <= rows;i++)
  for(j = 1 ; j <= rows;j++)
      mat.setVal(i,j,random(255));
  return mat;
}
```

See also

getVal(), numRows(), numColumns(), matrixAddition(), matrixMultiply(), matrixMultiplyScalar(), matrixTranspose(), newMatrix()

setVariable()

Usage

```
-- Lingo syntax
spriteObjRef.setVariable(variableName, newValue)

// JavaScript syntax
spriteObjRef.setVariable(variableName, newValue);
```

Description

Function; sets the value of the given variable in the given Flash sprite. Flash variables were introduced in Flash version 4.

Parameters

variableName Required. Specifies the name of the variable.

newValue Required. Specifies the new value of the variable.

Example

The following statement sets the value of the variable currentURL in the Flash cast member in sprite 3. The new value of currentURL will be "http://www.adobe.com/software/flash/".

```
-- Lingo syntax
sprite(3).setVariable("currentURL", "http://www.adobe.com/software/flash/")

// JavaScript syntax
sprite(3).setVariable("currentURL", "http://www.adobe.com/software/flash/");
```

See also

hitTest(), getVariable()

shader()

Usage

```
member(whichCastmember).shader(whichShader)
member(whichCastmember).shader[index]
member(whichCastmember).model(whichModel).shader
member(whichCastmember).modelResource(whichModelResource).face[index].shader
```

Description

3D element, model property, and face property; the object used to define the appearance of the surface of the model. The shader is the "skin" which is wrapped around the model resource used by the model.

The shader itself is not an image. The visible component of a shader is created with up to eight layers of `texture`. These eight texture layers are either created from bitmap cast members or image objects within Director or imported with models from 3D modeling programs. For more information, see `texture`.

Every model has a linear list of shaders called the `shaderlist`. The number of entries in this list equals the number of meshes in the model resource used by the model. Each mesh can be shaded by only one shader.

The 3D cast member has a default shader named `DefaultShader`, which cannot be deleted. This shader is used when no shader has been assigned to a model and when a shader being used by a model is deleted.

The syntax `member(whichCastmember).model(whichModel).shader` gives access to the first shader in the model's `shaderlist` and is equivalent to `member(whichCastmember).model(whichModel).shaderList[1]`.

Create and delete shaders with the `newShader()` and `deleteShader()` commands.

Shaders are stored in the shader palette of the 3D cast member. They can be referenced by name (*whichShader*) or palette index (*shaderIndex*). A shader can be used by any number of models. Changes to a shader will appear in all models which use that shader.

There are four types of shaders:

`#standard` shaders present their textures realistically.

`#painter`, `#engraver`, and `#newsprint` shaders stylize their textures for painting, engraving, and newsprint effects. They have special properties in addition to the `#standard` shader properties.

The shaders used by individual faces of `#mesh` primitives can be set with the syntax `member(whichCastmember).modelResource(whichModelResource).face[index].shader`. Changes to this property require a call to the `build()` command.

Example

This statement sets the `shader` property of the model named Wall to the shader named WallSurface:

```
member("Room").model("Wall").shader = member("Room").shader("WallSurface")
```

See also

shaderList, newShader, deleteShader, face[ ], texture()

showLocals()

Usage

```
-- Lingo syntax
showLocals()
```

Description

Top level function (Lingo only); displays all local variables in the Message window. This command is useful only within handlers or parent scripts that contain local variables to display. All variables used in the Message window are automatically global.

Local variables in a handler are no longer available after the handler executes. Inserting the statement `showLocals()` in a handler displays all the local variables in that handler in the Message window.

This command is useful for debugging scripts.

Parameters

None.

See also

`clearGlobals()`, `global`, `showGlobals()`

showProps()

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.showProps()

// JavaScript syntax
memberOrSpriteObjRef.showProps();
```

Description

Command; displays a list of the current property settings of a Flash movie, Vector member, or currently playing sound in the Message window. This command is useful for authoring only; it does not work in projectors or in movies with Shockwave content.

Parameters

None.

Example

This handler accepts the name of a cast as a parameter, searches that cast for Flash movie cast members, and displays the cast member name, number, and properties in the Message window:

```
-- Lingo syntax
on ShowCastProperties(whichCast)
    repeat with i = 1 to castLib(whichCast).member.count
        castType = member(i, whichCast).type
        if (castType = #flash) OR (castType = #vectorShape) then
            put castType&&"cast member" && i & ":" && member(i, whichCast).name
            put RETURN
            member(i ,whichCast).showProps()
        end if
```

```
    end repeat
end

// JavaScript syntax
function ShowCastProperties(whichCast) {
    i = 1;
    while( i < (castLib(whichCast).member.count) +1 ) {
        castType = member(i, whichCast).type;
        if ((castType = "flash") || (castType = "vectorShape")) {
            trace (castType + " cast member " + i + ": " + member(i, whichCast).name) + \n;
            member(i ,whichCast).showProps();
i++;
        }
    }
}
```

See also

queue(), setPlayList()

showGlobals()

Usage

```
-- Lingo syntax
_global.showGlobals()

// JavaScript syntax
_global.showGlobals();
```

Parameters

None.

Description

Global method; displays all global variables in the Message window.

This method is useful for debugging scripts.

Example

This statement displays all global variables in the Message window:

```
-- Lingo syntax
on mouseDown
 _global.showGlobals()
end

// JavaScript syntax
function mouseDown() {
 _global.showGlobals();
}
```

See also

Global

shutDown()

Usage

```
-- Lingo syntax
_system.shutDown()

// JavaScript syntax
_system.shutDown();
```

Description

System method; closes all open applications and turns off the computer.

Parameters

None.

Example

This statement checks whether the user has pressed Control+S (Windows) or Command+S (Mac) and, if so, shuts down the computer:

See also

System

sin()

Usage

```
sin(angle)
```

Description

Math function (Lingo only); calculates the sine of the specified angle. The angle must be expressed in radians as a floating-point number.

In JavaScript syntax, use the Math object's `sin()` function.

Parameters

angle Required. Specifies the angle.

Example

This statement calculates the sine of `pi`/2:

```
put sin (PI/2.0)
-- 1
```

See also

PI

sort

Usage

```
list.sort()
```

```
sort list
```

Description

Command; puts list items into alphanumeric order.

- When the list is a linear list, the list is sorted by values.
- When the list is a property list, the list is sorted alphabetically by properties.

After a list is sorted, it maintains its sort order even when you add new variables using the `add` command.

Parameters

None.

Example

The following statement puts the list Values, which consists of [#a: 1, #d: 2, #c: 3], into alphanumeric order. The result appears below the statement.

```
put values
-- [#a: 1, #d: 2, #c: 3]
values.sort()
put values
--[#a: 1, #c: 3, #d: 2]
```

sound()

Usage

```
-- Lingo syntax
sound(intSoundChannel)

// JavaScript syntax
sound(intSoundChannel);
```

Description

Top level function; returns a reference to a specified sound channel.

The functionality of this method is identical to the Sound object's `channel()` method.

Parameters

`intSoundChannel` Required. An integer that specifies the sound channel to reference.

Example

The following example assigns sound channel 1 to a variable `music` and plays a sound.

```
-- Lingo syntax
music = sound(1)
music.play(member("waltz1"))

// JavaScript syntax
var music = sound(1);
music.play(member("waltz1"));
```

See also

`channel() (Sound)`, `Sound Channel`

sprite()

Usage

```
-- Lingo syntax
sprite(nameOrNum)

// JavaScript syntax
sprite(nameOrNum);
```

Description

Top level function; returns a reference to a given sprite in the Score.

If the movie `scriptExecutionStyle` property is set to a value of 9, calling `sprite("foo")` where no sprite with that name exists returns a reference to sprite 1. If the movie `scriptExecutionStyle` property is set to a value of 10, calling `sprite("foo")` where no sprite with that name exists returns `VOID` if called from Lingo or `undefined` if called from JavaScript.

Parameters

`nameOrNum` Required. A string or integer that specifies the name or index position of the sprite.

Example

This statement sets the variable `thisSprite` to the sprite named Cave:

```
-- Lingo syntax
thisSprite = sprite("Cave")

// JavaScript syntax
var thisSprite = sprite("Cave");
```

See also

`Sprite Channel`

spriteSpaceToWorldSpace

Usage

```
sprite(whichSprite).camera.spriteSpaceToWorldSpace(loc)
sprite(whichSprite).camera(index).spriteSpaceToWorldSpace(loc)
```

Description

3D command; returns a world-space position that is found on the specified camera's projection plane that corresponds to a location within the referenced sprite.

The projection plane is defined by the camera's X and Y axes, and is at a distance in front of the camera such that one pixel represents one world unit of measurement. It is this projection plane that is used for the sprite display on stage.

The `camera.spriteSpaceToWorldSpace()` form of this command is a shortcut for using `camera(1).spriteSpaceToWorldSpace()`.

All cameras that are used by the referenced sprite will respond to the `spriteSpaceToWorldSpace` command as if their display rect is the same size as the sprite.

Parameters

loc Required. Specifies the location in the referenced sprite. This location should be a point relative to the sprite's upper-left corner.

Example

This statement shows that the point (50, 50) within sprite 5 is equivalent to the vector (-1993.6699, 52.0773, 2263.7446) on the projection plane of the camera of sprite 5:

```
-- Lingo
put sprite(5).camera.spriteSpaceToWorldSpace(point(50, 50))
-- vector(-1993.6699, 52.0773, 2263.7446)

// Javascript
put (sprite(5).camera.spriteSpaceToWorldSpace(point(50, 50)) );
//<vector(-1993.6699, 52.0773, 2263.7446)>
```

See also

worldSpaceToSpriteSpace, rect (camera), camera

sqrt()

Usage

```
sqrt(number)
the sqrt of number
```

Description

Math function (Lingo only); returns the square root of a specified number.

The value must be a decimal number greater than 0. Negative values return 0.

In JavaScript syntax, use the Math object's `sqrt()` function.

Parameters

number Required. Specifies the number. This number is either a floating-point number or an integer rounded to the nearest integer.

Example

This statement displays the square root of 3.0 in the Message window:

```
put sqrt(3.0)
-- 1.7321
```

This statement displays the square root of 3 in the Message window:

```
put sqrt(3)
-- 2
```

See also

floatPrecision

stageBottom

Usage

```
the stageBottom
```

Description

Function; along with `stageLeft`, `stageRight`, and `stageTop`, indicates where the Stage is positioned on the desktop. It returns the bottom vertical coordinate of the Stage relative to the upper left corner of the main screen. The height of the Stage in pixels is determined by `the stageBottom - the stageTop`.

When the movie plays back as an applet, the `stageBottom` property is the height of the applet in pixels.

This function can be tested but not set.

Parameters

None.

Example

These statements position sprite 3 a distance of 50 pixels from the bottom edge of the Stage:

```
stageHeight = the stageBottom - the stageTop
sprite(3).locV = stageHeight - 50
```

Sprite coordinates are expressed relative to the upper left corner of the Stage. For more information, see the Using Director topics in the Director Help Panel.

```
// Javascript
var stageHeight =_movie.stage.rect.Bottom -_movie.stage.rect.Top ;
sprite(3).locV = stageHeight - 50 ;
```

See also

`stageLeft`, `stageRight`, `stageTop`, `locH`, `locV`

stageLeft

Usage

```
the stageLeft
```

Description

Function; along with `stageRight`, `stageTop`, and `stageBottom`, indicates where the Stage is positioned on the desktop. It returns the left horizontal coordinate of the Stage relative to the upper left corner of the main screen. When the Stage is flush with the left side of the main screen, this coordinate is 0.

When the movie plays back as an applet, the `stageLeft` property is 0, which is the location of the left side of the applet.

This property can be tested but not set.

Sprite coordinates are expressed relative to the upper left corner of the Stage.

Parameters

None.

Example

This statement checks whether the left edge of the Stage is beyond the left edge of the screen and calls the handler `leftMonitorProcedure` if it is:

```
if the stageLeft < 0 then leftMonitorProcedure
```

See also

`stageBottom`, `stageRight`, `stageTop`, `locH`, `locV`

stageRight

Usage

```
the stageRight
```

Description

Function; along with `stageLeft`, `stageTop`, and `stageBottom`, indicates where the Stage is positioned on the desktop. It returns the right horizontal coordinate of the Stage relative to the upper left corner of the main screen's desktop. The width of the Stage in pixels is determined by `the stageRight - the stageLeft`.

When the movie plays back as an applet, the `stageRight` property is the width of the applet in pixels.

This function can be tested but not set.

Sprite coordinates are expressed relative to the upper left corner of the Stage.

Parameters

None.

Example

These two statements position sprite 3 a distance of 50 pixels from the right edge of the Stage:

```
stageWidth = the stageRight - the stageLeft
    sprite(3).locH = stageWidth - 50
```

See also

`stageLeft`, `stageBottom`, `stageTop`, `locH`, `locV`

stageToFlash()

Usage

```
-- Lingo syntax
spriteObjRef.stageToFlash(pointOnDirectorStage)

// JavaScript syntax
spriteObjRef.stageToFlash(pointOnDirectorStage);
```

Description

Function; returns the coordinate in a Flash movie sprite that corresponds to a specified coordinate on the Director Stage. The function both accepts the Director Stage coordinate and returns the Flash movie coordinate as Director point values: for example, point (300,300).

Flash movie coordinates are measured in Flash movie pixels, which are determined by the original size of the movie when it was created in Flash. Point (0,0) of a Flash movie is always at its upper left corner. (The cast member's `originPoint` property is not used to calculate movie coordinates; it is used only for rotation and scaling.)

The `stageToFlash()` function and the corresponding `flashToStage()` function are helpful for determining which Flash movie coordinate is directly over a Director Stage coordinate. For both Flash and Director, point (0,0) is the upper left corner of the Flash Stage or Director Stage. These coordinates may not match on the Director Stage if a Flash sprite is stretched, scaled, or rotated.

Parameters

`pointOnDirectorStage` Required. Specifies the point on the Director stage.

Example

The following handler checks to see if the mouse pointer (whose location is tracked in Director Stage coordinates) is over a specific coordinate (130,10) in a Flash movie sprite in channel 5. If the pointer is over that Flash movie coordinate, the script stops the Flash movie.

```
-- Lingo syntax
on checkFlashRollover
    if sprite(5).stageToFlash(point(_mouse.mouseH,_mouse.mouseV)) = point(130,10) then
        sprite(5).stop()
    end if
end

// JavaScript syntax
function checkFlashRollover() {
    var stf = sprite(5).stageToFlash(point(_mouse.mouseH,_mouse.mouseV));
    if (stf = point(130,10)) {
        sprite(5).stop();
    }
}
```

See also

`flashToStage()`

stageTop

Usage

```
the stageTop
```

Description

Function; along with `stageBottom`, `stageLeft`, and `stageRight`, indicates where the Stage is positioned on the desktop. It returns the top vertical coordinate of the Stage relative to the upper left corner of the main screen's desktop. If the Stage is in the upper left corner of the main screen, this coordinate is 0.

When the movie plays back as an applet, the `stageTop` property is always 0, which is the location of the left side of the applet.

This function can be tested but not set.

Sprite coordinates are expressed relative to the upper left corner of the Stage.

Parameters

None.

Example

This statement checks whether the top of the Stage is beyond the top of the screen and calls the handler `upperMonitorProcedure` if it is:

```
if the stageTop < 0 then upperMonitorProcedure
```

See also

stageLeft, stageRight, stageBottom, locH, locV

status()

Usage

```
-- Lingo syntax
fileioObjRef.status()

// JavaScript syntax
fileioObjRef.status();
```

Description

Fileio method; Returns the error code of the last method called.

Parameters

None.

Example

This statement displays the current status of sound channel 2 in the Message window:

```
-- Lingo syntax
put(sound(2).status)

// JavaScript syntax
put(sound(2).status);
```

See also

Fileio

stop() (DVD)

Usage

```
-- Lingo syntax
dvdObjRef.stop()

// JavaScript syntax
dvdObjRef.stop();
```

Description

DVD method; stops playback.

This method returns TRUE (1) if successful.

Parameters

None.

Example

This statement stops playback:

```
-- Lingo syntax
member(1).stop()

// JavaScript syntax
member(1).stop();
```

See also

DVD

stop() (Sound Channel)

Usage

```
-- Lingo syntax
soundChannelObjRef.stop()

// JavaScript syntax
soundChannelObjRef.stop();
```

Description

Sound Channel method; stops the currently playing sound in a sound channel.

Issuing a `play()` method begins playing the first sound of those that remain in the queue of the given sound channel.

To see an example of `stop()` used in a completed movie, see the Sound Control movie in the Learning/Lingo folder inside the Director application folder.

Parameters

None.

Example

This statement stops playback of the sound cast member currently playing in sound channel 1:

```
-- Lingo syntax
sound(1).stop()

// JavaScript syntax
sound(1).stop();
```

See also

getPlayList(), pause() (Sound Channel), play() (Sound Channel), playNext() (Sound Channel), rewind() (Sound Channel), Sound Channel

stop() (Flash)

Usage

```
-- Lingo syntax
spriteObjRef.stop()

// JavaScript syntax
spriteObjRef.stop();
```

Description

Flash command; stops a Flash movie sprite that is playing in the current frame.

Parameters

None.

Example

This frame script stops the Flash movie sprites playing in channels 5 through 10:

```
-- Lingo syntax
on enterFrame
    repeat with i = 5 to 10
        sprite(i).stop()
    end repeat
end

// JavaScript syntax
function enterFrame() {
    var i = 5;
    while (i < 11) {
        sprite(i).stop();
        i++;
    }
}
```

See also

hold()

stop() (RealMedia, SWA, Windows Media)

Usage

```
-- Lingo syntax
windowsMediaObjRef.stop()
realMediaObjRef.stop()

// JavaScript syntax
windowsMediaObjRef.stop();
realMediaObjRef.stop();
```

Description

Windows Media or RealMedia cast member or sprite method. Stops playback of a Windows Media or RealMedia cast member or sprite.

Parameters

None.

Example

The following examples stop sprite 2 and the cast member Real from playing:

```
-- Lingo syntax
sprite(2).stop()
member("Real").stop()

// JavaScript syntax
sprite(2).stop();
member("Real").stop();
```

See also

RealMedia, Windows Media

stop

Usage

```
stop(MUIObject, stopItem)
```

Description

This function stops the general purpose dialog created from an instance of the MUI Xtra. After the function is called, Lingo returns the results as the number for the stopItem parameter.

The stopItem parameter is returned from the run(MUIOject) call. Use this to pass back a parameter indicating how the dialog box was stopped. For example, this could return 1 if the user clicked OK and return 0 if the user clicked Cancel.

Note: *To close a non-modal dialog box, use the WindowOperation command with the #hide option.*

Example

This handler stops the general purpose dialog box created from MUIObject. The second parameter of the stop command is zero, which fulfills the requirement for a value but has no other purpose:

```
--Lingo syntax
on stopDialog
   global MUIObject
   if ( objectP (MUIObject)) then
      stop(MUIObject, 0)
   end if
end stopDialog
```

stopEvent()

Usage

```
-- Lingo syntax
_movie.stopEvent()

// JavaScript syntax
_movie.stopEvent();
```

Description

Movie method; prevents scripts from passing an event message to subsequent locations in the message hierarchy.

This method also applies to sprite scripts.

Use the `stopEvent()` method to stop the message in a primary event handler or a sprite script, thus making the message unavailable for subsequent sprite scripts.

By default, messages are available first to a primary event handler (if one exists) and then to any scripts attached to a sprite involved in the event. If more than one script is attached to the sprite, the message is available to each of the sprite's scripts. If no sprite script responds to the message, the message passes to a cast member script, frame script, and movie script, in that order.

The `stopEvent()` method applies only to the current event being handled. It does not affect future events. The `stopEvent()` method applies only within primary event handlers, handlers that primary event handlers call, or multiple sprite scripts. It has no effect elsewhere.

Parameters

None.

Example

This statement shows the `mouseUp` event being stopped in a behavior if the global variable `grandTotal` is equal to 500:

```
-- Lingo syntax
global grandTotal
on mouseUp me
    if (grandTotal = 500) then
        _movie.stopEvent()
    end if
end

// JavaScript syntax
_global.grandTotal;
function mouseUp() {
    if (_global.grandTotal == 500) {
        _movie.stopEvent();
    }
}
```

Neither subsequent scripts nor other behaviors on the sprite receive the event if it is stopped in this manner.

See also

Movie

stream()

Usage

```
-- Lingo syntax
memberObjRef.stream(numberOfBytes)

// JavaScript syntax
memberObjRef.stream(numberOfBytes);
```

Description

Command; manually streams a portion of a specified Flash movie cast member into memory.

The `stream` command returns the number of bytes actually streamed. Depending on a variety of conditions (such as network speed or the availability of the requested data), the number of bytes actually streamed may be less than the number of bytes requested.

You can always use the `stream` command for a cast member regardless of the cast member's `streamMode` property.

Parameters

`numberOfBytes` Optional. An integer that specifies the number of bytes to stream. If you omit the *`numberOfBytes`* parameter, Director tries to stream the number of bytes set by the cast member's `bufferSize` property.

Example

The following frame script checks to see if a linked Flash movie cast member has streamed into memory by checking its `percentStreamed` property. If the cast member is not completely loaded into memory, the script tries to stream 32,000 bytes of the movie into memory.

The script also saves the actual number of bytes streamed in a variable called `bytesReceived`. If the number of bytes actually streamed does not match the number of bytes requested, the script updates a text cast member to report the number of bytes actually received. The script keeps the playhead looping in the current frame until the cast member has finished loading into memory.

```
-- Lingo syntax
on exitFrame
    if member(10).percentStreamed < 100 then
        bytesReceived = member(10).stream(32000)
        if bytesReceived < 32000 then
            member("Message Line").text = "Received only" && bytesReceived && "of 32,000
bytes requested."
            _movie.updateStage()
        else
            member("Message Line").text = "Received all 32,000 bytes."
        end if
        _movie.go(_movie.frame)
    end if
end

// JavaScript syntax
function exitFrame() {
    var pctStm = member(10).percentStreamed;
    if (pctStm < 100) {
        var bytesReceived = member(10).stream(32000);
        if (bytesReceived < 32000) {
            member("Message Line").text = "Received only " + bytesReceived + " of32,000 bytes
requested.";
            _movie.updateStage();
        } else {
            member("Message Line").text = "Received all 32,000 bytes.";
        }
        _movie.go(_movie.frame);
    }
}
```

string()

Usage

```
string(expression)
```

Description

Function; converts an integer, floating-point number, object reference, list, symbol, or other nonstring expression to a string.

Parameters

expression Required. The expression to convert to a string.

Example

This statement adds 2.0 + 2.5 and inserts the results in the field cast member Total:

```
--Lingo
member("total").text = string(2.0 + 2.5)

// Javascript
member("total").text = (2.0 + 2.5).toString() ;
```

This statement converts the symbol `#red` to a string and inserts it in the field cast member Color:

```
--Lingo
member("Color").text = string(#red)

// Javascript
member("Color").text = symbol("red").toString();
```

See also

`value()`, `stringP()`, `float()`, `integer()`, `symbol()`

stringP()

Usage

```
stringP(expression)
```

Description

Function; determines whether an expression is a string (`TRUE`) or not (`FALSE`).

The *P* in `stringP` stands for *predicate*.

Parameters

expression Required. The expression to test.

Example

This statement checks whether *3* is a string:

```
put stringP("3")
```

The result is 1, which is the numeric equivalent of `TRUE`.

This statement checks whether the floating-point number 3.0 is a string:

```
put stringP(3.0)
```

Because 3.0 is a floating-point number and not a string, the result is 0, which is the numeric equivalent of FALSE.

See also

floatP(), ilk(), integerP(), objectP(), symbolP()

subPictureType()

Usage

```
-- Lingo syntax
dvdObjRef.subPictureType(intStream)

// JavaScript syntax
dvdObjRef.subPictureType(intStream);
```

Description

DVD method; specifies the type of a specified sub-picture stream.

This method can return the following values:

Symbol	Description
#unknown	The sub-picture type is unknown.
#Language	The sub-picture contains language-related content such as movie subtitles or other text.
#Other	The sub-picture contains non language-related content such as a bouncing ball in karaoke titles.

Parameters

intStream Required. An integer that specifies the stream to test.

Example

This statement returns the sub-picture type in stream 2:

```
-- Lingo syntax
member(12).member.subPictureType(2)

// JavaScript syntax
member(12).member.subPictureType(2);
```

See also

DVD

substituteFont

Usage

```
TextMemberRef.substituteFont(originalFont, newFont)
substituteFont(textMemberRef, originalFont, newFont)
```

Description

Text cast member command; replaces all instances of one font with another font in a text cast member.

Parameters

originalFont Required. The font to replace.

newFont Required. The new font that replaces the font specified by *originalFont*.

Example

This script checks to see if the font Bonneville is available in a text cast member, and replaces it with Arial if it is not:

```
-- Lingo syntax
property spriteNum

on beginSprite me
    currMember = sprite(spriteNum).member
    if currMember.missingFonts contains "Bonneville" then
        currMember.substituteFont("Bonneville", "Arial")
    end if
end

// JavaScript syntax
function beginSprite() {
    currMember = sprite(spriteNum).member;
    if (currMember.missingFonts contains "Bonneville") { //check syntax
        currMember.substituteFont("Bonneville", "Arial");
    }
}
```

See also

missingFonts

swing()

Usage

```
-- Lingo syntax
spriteObjRef.swing(pan, tilt, fieldOfView, speedToSwing)

// JavaScript syntax
spriteObjRef.swing(pan, tilt, fieldOfView, speedToSwing);
```

Description

QuickTime VR sprite function; swings a QuickTime 3 sprite containing a VR Pano around to the new view settings. The swing is a smooth "camera dolly" effect.

whichQTVRSprite is the sprite number of the sprite with the QuickTime VR member.

The function returns immediately, but the sprite continues to change view until it reaches the final view. The duration required to change to the final settings varies depending on machine type, size of the sprite rectangle, color depth of the screen, and other typical performance constraints.

To check if the swing has finished, check if the `pan` property of the sprite has arrived at the final value.

Parameters

pan Required. Specifies the new pan position, in degrees.

tilt Required. Specifies the new tilt, in degrees.

fieldOfView Required. Specifies the new field of view, in degrees.

speedToSwing Required. Specifies the rate at which the swing should take place. Valid values range from 1 (slow) to 10 (fast).

Example

This very gradually adjusts the view of QTVR sprite 1 to a pan position of 300°, a tilt of -15°, and a field of view of 40°:

```
-- Lingo syntax
sprite(1).swing(300, -15, 40, 1)

// JavaScript syntax
sprite(1).swing(300, -15, 40, 1);
```

See also

pan (QTVR property)

symbol()

Usage

```
-- Lingo syntax
symbol(stringValue)

// JavaScript syntax
symbol(stringValue);
```

Description

Top level function; takes a string and returns a symbol.

Parameters

stringValue Required. The string to convert to a symbol.

Example

This statement displays the symbol #hello:

```
--Lingo syntax
put(symbol("hello"))

// JavaScript syntax
put(symbol("hello"));
```

This statement displays the symbol #goodbye:

```
--Lingo syntax
x = "goodbye"
put(symbol(x))

// JavaScript syntax
var x = "goodbye";
put(symbol(x));
```

See also

value(), string()

symbolP()

Usage

```
Expression.symbolP
symbolP(expression)
```

Description

Function; determines whether a specified expression is a symbol (`TRUE`) or not (`FALSE`).

The *P* in `symbolP` stands for predicate.

Parameters

`expression` Required. Specifies the expression to test.

Example

This statement checks whether the variable `myVariable` is a symbol:

```
put myVariable.symbolP
```

See also

`ilk()`

tan()

Usage

```
tan(angle)
```

Description

Math function; yields the tangent of the specified angle expressed in radians as a floating-point number.

In JavaScript syntax, use the Math object's `tan()` function.

Parameters

`angle` Required. Specifies the angle from which a tangent is yielded.

Example

The following function yields the tangent of pi/4:

```
tan (PI/4.0) = 1
```

The p symbol cannot be used in a Lingo expression.

See also

`PI`

tellStreamStatus()

Usage

```
tellStreamStatus(onOrOffBoolean)
```

Description

Function; turns the stream status handler on (TRUE) or off (FALSE).

The form tellStreamStatus() determines the status of the handler.

When the streamStatusHandler is TRUE, Internet streaming activity causes periodic calls to the movie script, triggering streamStatusHandler. The handler is executed, with Director automatically filling in the parameters with information regarding the progress of the downloads.

Parameters

onOrOffBoolean Optional. Specifies the status of the handler.

Example

This on prepareMovie handler turns the on streamStatus handler on when the movie starts:

```
-- Lingo syntax
on prepareMovie
    tellStreamStatus(TRUE)
end

// JavaScript syntax
function prepareMovie() {
    tellStreamStatus(TRUE);
}
```

This statement determines the status of the stream status handler:

```
-- Lingo syntax
on mouseDown
    put tellStreamStatus()
end

// JavaScript syntax
function mouseDown() {
    put(tellStreamStatus());
}
```

See also

on streamStatus

tellTarget()

Usage

```
-- Lingo syntax
spriteObjRef.tellTarget(targetName)

// JavaScript syntax
spriteObjRef.tellTarget(targetName);
```

Description

Command; equivalent to the Flash beginTellTarget and endTellTarget methods. The tellTarget() command allows the user to set a target Timeline on which subsequent sprite commands will act. When the target is set to a Flash movie clip or a level containing a loaded Flash movie, certain commands act on the targeted components, rather than on the main Timeline. To switch focus back to the main Timeline, call endTellTarget().

The only valid argument for `tellTarget` is the target name. There is no valid argument for `endTellTarget`.

The Flash sprite functions that are affected by `tellTarget` are `stop`, `play`, `getProperty`, `setProperty`, `gotoFrame`, `call(frame)`, and `find(label)`. In addition, the sprite property `frame` (which returns the current frame) is affected by `tellTarget`.

Parameters

`targetName` Required. Specifies the target name.

Example

This command sets the movie clip as the target:

```
-- Lingo syntax
sprite(1).tellTarget("myMovieClip")

// JavaScript syntax
sprite(1).tellTarget("myMovieClip");
```

This command stops the movie clip:

```
-- Lingo syntax
sprite(1).stop()

// JavaScript syntax
sprite(1).stop();
```

This command causes the movie clip to play:

```
-- Lingo syntax
sprite(1).play()

// JavaScript syntax
sprite(1).play();
```

This command switches the focus back to the main Timeline:

```
-- Lingo syntax
sprite(1).endTellTarget()

// JavaScript syntax
sprite(1).endTellTarget();
```

This command stops the main movie:

```
-- Lingo syntax
sprite(1).stop()

// JavaScript syntax
sprite(1).stop();
```

texture()

Usage

```
member(whichCastmember).texture(whichTexture)
member(whichCastmember).texture[index]
member(whichCastmember).shader(whichShader).texture
member(whichCastmember).model(whichModel).shader.texture
member(whichCastmember).model(whichModel).shaderList.texture
member(whichCastmember).model(whichModel).shaderList[index].texture
```

```
member(whichCastmember).modelResource(whichParticleSystemModelResource).texture
```

Description

3D element and shader property; an image object used by a shader to define the appearance of the surface of a model. The image is wrapped onto the geometry of the model by the shader.

The visible component of a shader is created with up to eight layers of textures. These eight texture layers are either created from bitmap cast members or image objects within Director or imported with models from 3D modeling programs.

Create and delete textures with the `newTexture()` and `deleteTexture()` commands.

Textures are stored in the texture palette of the 3D cast member. They can be referenced by name (*`whichTexture`*) or palette index (*`textureIndex`*). A texture can be used by any number of shaders. Changes to a texture will appear in all shaders which use that texture.

There are three types of textures:

`#fromCastmember`; the texture is created from a bitmap cast member using the `newTexture()` command.

`#fromImageObject`; the texture is created from a lingo image object using the `newTexture()` command.

`#importedFromFile`; the texture is imported with a model from a 3D modeling program.

The texture of a particle system is a property of the model resource, whose type is `#particle`.

Example

This statement sets the `texture` property of the shader named WallSurface to the texture named BluePaint:

```
member("Room").shader("WallSurface").texture = member("Room").texture("BluePaint")
```

See also

`newTexture`, `deleteTexture`

time() (System)

Usage

```
-- Lingo syntax
_system.time()

// JavaScript syntax
_system.time();
```

Description

System method; returns the current time in the system clock as a string. The format of the time string depends on the computer's time settings.

Parameters

None.

Example

The following handler outputs the current time to a text field.

```
-- Lingo syntax
on exitFrame
```

```
    member("clock").text = _system.time()
end

// JavaScript syntax
function exitFrame() {
    member("clock").text = _system.time();
}
```

See also

date() (System), System

timeout()

Usage

```
-- Lingo syntax
timeout(timeoutObjName)

// JavaScript syntax
timeout(timeoutObjName);
```

Description

Top level function; returns a given timeout object.

Use the `new()` method to create a new timeout object and add it to the `timeoutList`.

Parameters

`timeoutObjName` Required. A string that specifies the name of the timeout object to return.

Example

This handler deletes the timeout object named Random Lightning:

```
-- Lingo syntax
on exitFrame
    timeout("Random Lightning").forget()
end

// JavaScript syntax
function exitFrame() {
    timeout("Random Lightning").forget();
}
```

See also

new(), timeoutList, timeoutHandler, time (timeout object), name (timeout), period, persistent, target

titleMenu()

Usage

```
-- Lingo syntax
dvdObjRef.titleMenu()

// JavaScript syntax
```

```
dvdObjRef.titleMenu();
```

Description

DVD method; displays the title menu.

Parameters

None.

Example

This statement displays the title menu:

```
-- Lingo syntax
member(1).titleMenu()

// JavaScript syntax
member(1).titleMenu();
```

See also

DVD

top (3D)

Usage

```
modelResourceObjectReference.top
```

Description

3D command; when used with a model resource whose type is `#box`, allows you to both get and set the `top` property of the model resource.

The `top` property determines whether the top of the box is sealed (`TRUE`) or open (`FALSE`). The default value is `TRUE`.

Parameters

None.

Example

This statement checks whether the `top` of sprite 3 is above the top of the Stage and calls the handler `offTopEdge` if it is:

```
-- Lingo syntax
if (sprite(3).top < 0) then
    offTopEdge()
end if

// JavaScript syntax
if (sprite(3).top < 0) {
    offTopEdge();
}
```

See also

back, bottom (3D), front

topCap

Usage

`modelResourceObjectReference.topCap`

Description

3D command; when used with a model resource whose type is `#cylinder`, allows you to both get and set the `topCap` property of the model resource.

The `topCap` property determines whether the top cap of the cylinder is sealed (`TRUE`) or open (`FALSE`). The default value for this property is `FALSE`.

Parameters

None.

Example

This statement sets the `topCap` property of the model resource Tube to `FALSE`, meaning the top of this cylinder will be open:

```
-- Lingo syntax
member("3D World").modelResource("Tube").topCap = FALSE

// JavaScript syntax
member("3D World").getPropRef("modelResource", 10).topCap = false;
```

topRadius

Usage

`modelResourceObjectReference.topRadius`

Description

3D command; when used with a model resource whose type is `#cylinder`, allows you to both get and set the `topRadius` property of the model resource, as a floating-point value.

The `topRadius` property determines the radius of the top cap of the cylinder. This property must always be 0.0 or greater. The default value is 25.0. Setting `topRadius` to 0.0 produces a cone.

Parameters

None.

Example

The following statement sets the `topRadius` property of the model resource Tube to 0.0. If the bottom radius has a value greater than 0, models using Tube will be conical.

```
-- Lingo syntax
member("3D World").modelResource("Tube").topRadius = 0.0

// JavaScript syntax
member("3D World").getPropRef("modelResource", 10).topRadius = 0.0;
```

trace()

Usage

```
-- Lingo syntax
trace(value)

// JavaScript syntax
trace(value);
```

Description

Top level function; evaluates an expression and displays the result in the Message window.

The functionality of this method is identical to the top level `put()` method, which is also available to both Lingo and JavaScript syntax.

This method can be used as a debugging tool by tracking the values of variables as a movie plays.

Parameters

`value` Required. The expression to evaluate.

Example

The following statement outputs the value of the variable `counter` to the Message window.

```
-- Lingo syntax
counter = (_system.milliseconds / 1000)
trace(counter)

// JavaScript syntax
var counter = (_system.milliseconds / 1000);
trace(counter);
```

See also

`put()`

transform (command)

Usage

```
transform()
```

Description

3D command; this command creates a transform object that is equal to the identity transform. The identity transform has positional and rotational components of vector(0,0,0), and it has a scale component of vector(1,1,1).

If you need to store and then rebuild transform information, store the transform properties (position, rotation and scale), then rebuild the transform by making an identity transform followed by setting the position, rotation and scale using the stored data.

Parameters

None.

Example

This statement creates an identity transform and stores it in the variable `tTransform`:

```
-- Lingo syntax
tTransform = transform()

// JavaScript syntax
tTransform = transform();
```

See also

transform (property), preRotate, preTranslate(), preScale(), rotate, translate, scale (command)

translate

Usage

```
member(whichCastmember).node(whichNode).translate(xIncrement, yIncrement, zIncrement {,
relativeTo})
member(whichCastmember).node(whichNode).translate(translateVector {, relativeTo})
transform.translate(xIncrement, yIncrement, zIncrement {, relativeTo})
transform.translate(translateVector {, relativeTo})
```

Description

3D command; applies a translation after the current positional, rotational, and scale offsets held by a referenced node's transform object or the directly referenced transform object. The translation must be specified as a set of three increments along the three corresponding axes. These increments may be specified explicitly in the form of *xIncrement*, *yIncrement*, and *zIncrement*, or by a *translateVector*, where the x component of the vector corresponds to the translation along the x axis, y about y axis, and z about z axis.

A node can be a camera, model, light or group object.

Parameters

xIncrement Required if specifying a set of three increments. Specifies the *x*-axis increment.

yIncrement Required if specifying a set of three increments. Specifies the *y*-axis increment.

zIncrement Required if specifying a set of three increments. Specifies the *z*-axis increment.

translateVector Required if specifying a vector. Specifies the vector that contains the *x*, *y*, and *z* components.

relativeTo Optional. Determines which coordinate system's axes are used to apply the desired translational changes. The *relativeTo* parameter can have any of the following values:

- `#self` applies the increments relative to the node's local coordinate system (the x, y and z axes specified for the model during authoring). This value is used as the default if you use the `translate` command with a node reference and the *relativeTo* parameter is not specified.
- `#parent` applies the increments relative to the node's parent's coordinate system. This value is used as the default if you use the `translate` command with a transform reference and the *relativeTo* parameter is not specified.
- `#world` applies the increments relative to the world coordinate system. If a model's parent is the world, than this is equivalent to using `#parent`.
- `nodeReference` allows you to specify a node to base your translation upon, the command applies the translations relative to the coordinate system of the specified node.

Example

This example constructs a transform using the transform command, then it initializes the transform's position and orientation in space before assigning the transform to the model named mars. Finally this example displays the resulting position of the model.

```
t =transform()
t.transform.identity()
t.transform.rotate(0, 90, 0)
t.transform.translate(100, 0, 0)
gbModel = member("scene").model("mars")
gbModel.transform = t
put gbModel.transform.position
-- vector(100.0000, 0.0000, 0.0000)
```

This Lingo moves the model Bip 20 units along the x axis of its parent node:

```
put member("Scene").model("Bip").position
-- vector( -38.5000, 21.2500, 2.0000)
member("Scene").model("Bip").translate(20, 10, -0.5)
put member("Scene").model("Bip").position
-- vector( -18.5000, 31.2500, 1.5000)
```

See also

transform (property), preTranslate(), scale (command), rotate

union()

Usage

```
rect(1).union(rect(2))
union (rect1, rect2)
```

Description

Function; returns the smallest rectangle that encloses two rectangles.

Parameters

rect2 Required. Specifies the second rectangle.

Example

This statement returns the rectangle that encloses the specified rectangles:

```
-- Lingo syntax
put union (rect (0, 0, 10, 10), rect (15, 15, 20, 20))
-- rect (0, 0, 20, 20)
```

or

```
put rect(0, 0, 10, 10).union(rect(15, 15, 20, 20))
--rect (0, 0, 20, 20)

// JavaScript syntax
put ( rect (0, 0, 10, 10).union( rect (15, 15, 20, 20) ) );
// <rect(0, 0, 20, 20)>
```

See also

map(), rect()

unLoad() (Member)

Usage

```
-- Lingo syntax
memberObjRef.unLoad({toMemberObjRef})

// JavaScript syntax
memberObjRef.unLoad({toMemberObjRef});
```

Description

Member method; forces Director to clear the specified cast members from memory.

Director automatically unloads the least recently used cast members to accommodate `preLoad()` methods or normal cast library loading.

- When used without a parameter, `unLoad()` clears from memory the current cast member.
- When used with the *`toMemberObjRef`* parameter, `unLoad()` clears from memory all the cast members in the range specified.

When used in a new movie with no loaded cast members, this method returns an error.

Cast members that you have modified during authoring or by setting `picture`, `pasteClipBoadInto()`, and so on, cannot be unloaded.

Parameters

`toMemberObjRef` Optional. A reference to the last cast member in the range to clear from memory.

Example

This statement clears the cast member named Ships from memory:

```
-- Lingo syntax
member("Ships").unLoad()

// JavaScript syntax
member("Ships").unLoad();
```

This statement clears from memory cast members 10 through 15:

```
-- Lingo syntax
member(10).unLoad(15)

// JavaScript syntax
member(10).unLoad(15);
```

See also

`Member unLoad() (Movie) unLoadMember() unLoadMovie()`

unLoad() (Movie)

Usage

```
-- Lingo syntax
_movie.unLoad({intFromFrameNum} {, intToFrameNum})

// JavaScript syntax
```

```
_movie.unLoad({intFromFrameNum} {, intToFrameNum});
```

Description

Movie method; removes the specified frame range of the movie from memory.

This command is useful in forcing movies to unload when memory is low.

You can use a URL as the file reference.

If the movie isn't already in RAM, the result is -1.

Parameters

intFromFrameNum Optional. An integer that specifies the number of the first frame in a range to unload from memory.

intToFrameNum Optional. An integer that specifies the number of the last frame in a range to unload from memory.

Example

The following statements unload frames 10 through 25 from memory.

```
-- Lingo syntax
_movie.unLoad(10, 25)

// JavaScript syntax
_movie.unLoad(10, 25);
```

See also

Movie unLoad() (Member) unLoadMember() unLoadMovie()

unLoadMember()

Usage

```
-- Lingo syntax
_movie.unLoadMember({memberObjRef})
_movie.unLoadMember(fromMemberNameOrNum, toMemberNameOrNum)

// JavaScript syntax
_movie.unLoadMember({memberObjRef});
_movie.unLoadMember(fromMemberNameOrNum, toMemberNameOrNum);
```

Description

Movie method; forces Director to clear a specific cast member or a range of cast members from memory. Director automatically unloads the least recently used cast members to accommodate `preLoad()` methods or normal cast library loading.

- When used without an argument, the `unLoadMember()` method clears from memory the cast members in all the frames of a movie.
- When used with one argument, *memberObjRef*, the `unLoadMember()` method clears from memory the specified cast member.
- When used with two arguments, *fromMemberNameOrNum* and *toMemberNameOrNum*, the `unLoadMember()` method unloads all cast members in the range specified. You can specify a range of cast members by member numbers or member names.

Parameters

memberObjRef Optional. A reference to the cast member to unload from memory.

fromMemberNameOrNum Required if clearing a range of cast members. A string or integer that specifies the name or number of the first cast member in a range to unload from memory.

toMemberNameOrNum Required if clearing a range of cast members. A string or integer that specifies the name or number of the last cast member in a range to unload from memory.

Example

This statement clears from memory the cast member Screen1:

```
-- Lingo syntax
_movie.unLoadMember(member("Screen1"))

// JavaScript syntax
_movie.unLoadMember(member("Screen1"));
```

This statement clears from memory all cast members from cast member 1 to cast member Big Movie:

```
-- Lingo syntax
_movie.unLoadMember(member(1), member("Big Movie"))

// JavaScript syntax
_movie.unLoadMember(member(1), member("Big Movie"));
```

See also

Movie unLoad() (Member) unLoad() (Movie) unLoadMovie()

unLoadMovie()

Usage

```
-- Lingo syntax
_movie.unLoadMovie(stringMovieName)

// JavaScript syntax
_movie.unLoadMovie(stringMovieName);
```

Description

Movie method; removes the specified preloaded movie from memory.

This command is useful in forcing movies to unload when memory is low.

You can use a URL as the file reference.

If the movie isn't already in RAM, the result is -1.

Parameters

stringMovieName Required. A string that specifies the name of the movie to unload from memory.

Example

This statement checks whether the largest contiguous block of free memory is less than 100K and unloads the movie Parsifal if it is:

```
-- Lingo syntax
if (_system.freeBlock < (100*1024)) then
```

```
    _movie.unLoadMovie("Parsifal")
end if

// JavaScript syntax
if (_system.freeBlock < (100*1024)) {
    _movie.unLoadMovie("Parsifal");
}
```

This statement unloads the movie at http://www.cbDemille.com/SunsetBlvd.dir:

```
-- Lingo syntax
_movie.unLoadMovie("http://www.cbDemille.com/SunsetBlvd.dir")

// JavaScript syntax
_movie.unLoadMovie("http://www.cbDemille.com/SunsetBlvd.dir");
```

See also

Movie unLoad() (Member) unLoad() (Movie) unLoadMember()

unregisterAllEvents

Usage

```
-- Lingo syntax
member(whichMember).unregisterAllEvents()

// JavaScript syntax
member(whichMember).unregisterAllEvents();
```

Description

3D command; unregisters the referenced cast member for all event notifications. Therefore, all handlers that were previously registered to respond to events using the `registerForEvent` command will no longer be triggered when those events occur.

Parameters

None.

Example

This statement unregisters the cast member named Scene for all event notifications:

```
-- Lingo syntax
member("Scene").unregisterAllEvents()

// JavaScript syntax
member("Scene").unregisterAllEvents();
```

See also

registerForEvent()

update

Usage

```
-- Lingo syntax
```

```
member(whichCastmember).model(whichModel).update

// JavaScript syntax
member(whichCastMember).model(whichModel).update();
```

Description

3D command; causes animations on the model to update without rendering. Use this command to find the exact position of an animating model in Lingo.

Parameters

None.

Example

```
-- Lingo syntax
member(whichCastmember).model(whichModel).update

// JavaScript syntax
member(whichCastMember).getPropRef("model",1).update();
```

updateFrame()

Usage

```
-- Lingo syntax
_movie.updateFrame()

// JavaScript syntax
_movie.updateFrame();
```

Description

Movie method; during Score generation only, enters the changes to the current frame that have been made during Score recording and moves to the next frame. Any objects that were already in the frame when the update session started remain in the frame. You must issue an `updateFrame()` method for each frame that you are updating.

Parameters

None.

Example

When used in the following handler, the `updateFrame` command enters the changes that have been made to the current frame and moves to the next frame each time Lingo reaches the end of the repeat loop. The number of frames is determined by the argument `numberOfFrames`.

```
-- Lingo syntax
on animBall(numberOfFrames)
    _movie.beginRecording()
    horizontal = 0
    vertical = 100
    repeat with i = 1 to numberOfFrames
        _movie.go(i)
        sprite(20).member = member("Ball").number
        sprite(20).locH = horizontal
        sprite(20).locV = vertical
        sprite(20).foreColor = 255
        horizontal = horizontal + 3
        vertical = vertical + 2
```

```
        _movie.updateFrame()
    end repeat
    _movie.endRecording()
end animBall

// JavaScript syntax
function animBall(numberOfFrames) {
    _movie.beginRecording();
    var horizontal = 0;
    var vertical = 100;
    for (var i = 1; i <= numberOfFrames; i++) {
        _movie.go(1);
        sprite(20).member = member("Ball");
        sprite(20).locH = horizontal;
        sprite(20).locV = vertical;
        sprite(20).foreColor = 255;
        horizontal = horizontal + 3;
        vertical = vertical + 2;
        _movie.updateFrame();
    }
    _movie.endRecording();
}
```

See also

beginRecording(), endRecording(), Movie, scriptNum, tweened

updateStage()

Usage

```
-- Lingo syntax
_movie.updateStage()

// JavaScript syntax
_movie.updateStage();
```

Description

Movie method; redraws the Stage immediately instead of only between frames.

The `updateStage()` method redraws sprites, performs transitions, plays sounds, sends a `prepareFrame` message (affecting movie and behavior scripts), and sends a `stepFrame` message (which affects `actorList`).

Parameters

None.

Example

This handler changes the sprite's horizontal and vertical locations and redraws the Stage so that the sprite appears in the new location without having to wait for the playhead to move:

```
-- Lingo syntax
on moveRight(whichSprite, howFar)
    sprite(whichSprite).locH = sprite(whichSprite).locH + howFar
    _movie.updateStage()
end moveRight

// JavaScript syntax
```

```
function moveRight(whichSprite, howFar) {
    sprite(whichSprite).locH = sprite(whichSprite).locH + howFar;
    _movie.updateStage();
}
```

See also

actorList, Movie, on prepareFrame, on stepFrame

URLEncode

Usage

```
URLEncode(proplist_or_string {, serverOSString} {, characterSet})
```

Description

Function; returns the URL-encoded string for its first argument. Allows CGI parameters to be used in other commands. The same translation is done as for `postNetText` and `getNetText()` when they are given a property list.

Parameters

`propListOrString` Required. Specifies the property list or string to be URL-encoded.

`serverOSString` Optional. Encodes any return characters in *`propListOrString`*. The value defaults to `"Unix"` but may be set to `"Win"` or `"Mac"` and translates any carriage returns in *`propListOrString`* into those used on the server. For most applications, this setting is unnecessary because line breaks are usually not used in form responses.

`characterSet` Optional. Applies only if the user is running on a Shift-JIS (Japanese) system. Its possible settings are `"JIS"`, `"EUC"`, `"ASCII"`, and `"AUTO"`. Retrieved data is converted from Shift-JIS to the named character set. Returned data is handled exactly as by `getNetText()` (converted from the named character set to Shift-JIS). If you use `"AUTO"`, the posted data from the local character set is not translated; the results sent back by the server are translated as they are for `getNetText()`. `"ASCII"` is the default if *`characterSet`* is omitted. `"ASCII"` provides no translation for posting or results.

Example

In the following example, `URLEncode` supplies the URL-encoded string to a CGI query at the specified location.

```
URL = "http://aserver/cgi-bin/echoquery.cgi"
gotonetpage URL & "?" & URLEncode( [#name: "Ken", #hobby: "What?"] )
```

See also

getNetText(), postNetText

value()

Usage

```
value(stringExpression)
```

Description

Function; returns the value of a string. When `value()` is called, Lingo parses through the *`stringExpression`* provided and returns its logical value.

Any Lingo expression that can be `put` in the Message window or set as the value of a variable can also be used with `value()`.

These two Lingo statements are equivalent:

```
put sprite(2).member.duration * 5
put value("sprite(2).member.duration * 5")
```

These two Lingo statements are also equivalent:

```
x = (the mouseH - 10) / (the mouseV + 10)
x = value("(the mouseH - 10) / (the mouseV + 10)")
```

Expressions that Lingo cannot parse will produce unexpected results, but will not produce Lingo errors. The result is the value of the initial portion of the expression up to the first syntax error found in the string.

The `value()` function can be useful for parsing expressions input into text fields by end-users, string expressions passed to Lingo by Xtra extensions, or any other expression you need to convert from a string to a Lingo value.

Keep in mind that there may be some situations where using `value()` with user input can be dangerous, such as when the user enters the name of a custom handler into the field. This will cause the handler to be executed when it is passed to `value()`.

Do not confuse the actions of the value function with the `integer()` and `float()` functions.

Parameters

`stringExpression` Required. Specifies the string from which a value is returned. The string can be any expression that Lingo can understand.

Example

This statement displays the numerical value of the string `"the sqrt of" && "2.0"`:

```
put value("the sqrt of" && "2.0")
```

The result is 1.4142.

This statement displays the numerical value of the string "penny":

```
put value("penny")
```

The resulting display in the Message window is `VOID`, because the word *penny* has no numerical value.

You can convert a string that is formatted as a list into a true list by using this syntax:

```
myString = "[" & QUOTE & "cat" & QUOTE & ", " & QUOTE & "dog" & QUOTE & "]"
myList = value(myString)
put myList
-- ["cat", "dog"]
```

This allows a list to be placed in a field or text cast member and then extracted and easily reformatted as a list.

This statement parses the string "3 5" and returns the value of the portion of the string that Lingo understands:

```
put value("3 5")
-- 3
```

See also

`string()`, `integer()`, `float()`

vector()

Usage

```
-- Lingo syntax
vector()
vector(intX, intY, intZ)

// JavaScript syntax
vector();
vector(intX, intY, intZ);
```

Description

Top level function and data type. Describes a point in 3D space according to three parameters, which are the specific distances from the reference point along the *x*-axis, *y*-axis, and *z*-axis, respectively.

If the vector is in world space, the reference point is the world origin, `vector(0, 0, 0)`. If the vector is in object space, the reference point is the object's position and orientation.

This method returns a vector object.

Vector values can be operated upon by the `+`, `-`, `*` and `/` operators. See their individual definitions for more information.

Parameters

`intX` Optional. An integer that specifies the *x*-axis point.

`intY` Optional. An integer that specifies the *y*-axis point.

`intZ` Optional. An integer that specifies the *z*-axis point.

Example

This statement creates a vector and assigns it to the variable `myVector`:

```
-- Lingo syntax
myVector = vector(10.0, -5.0, 0.0)

// JavaScript syntax
var myVector = vector(10.0, -5.0, 0.0);
```

In Lingo only, this statement adds two vectors and assigns the resulting value to the variable `thisVector`:

```
-- Lingo syntax
thisVector = vector(1.0, 0.0, 0.0) + vector(0.0, -12.5, 2.0)
```

version()

Usage

```
-- Lingo syntax
fileioObjRef.version()
```

```
// JavaScript syntax
fileioObjRef.version();
```

Description

Fileio method; Displays the Fileio version and build information in the Message window.

Parameters

None.

Example

This statement displays the version of Director in the Message window:

```
-- Lingo
put(_player.productVersion)

// Javascript
trace(_player.productVersion)
```

See also

Fileio

voiceCount()

Usage

```
voiceCount()
```

Description

Function: returns the number of installed voices available to the text-to-speech engine. The return value is an integer. This number of voices can be used with `voiceSet()` and `voiceGet()` to specify a particular voice to be active.

Parameters

None.

Example

This statement sets the variable `numVoices` to the number of available text-to-speech voices:

```
-- Lingo
numVoices = voiceCount()

// Javascript
Var numVoices = voiceCount();
```

See also

voiceInitialize(), voiceSet(), voiceGet()

voiceGet()

Usage

```
voiceGet()
```

Description

Function; returns a property list describing the current voice being used for text-to-speech. The list contains the following properties:

- `#name` indicates the name of the installed voice.
- `#age` indicates the age of the voice. The value is a string. Possible values include "Teen", "Adult", "Toddler", and "Senior", as well as numeric values such as "35". Actual values depend on the operating system, speech software version, and voices installed.
- `#gender` indicates whether the voice is male or female. The value is a string.
- `#index` indicates the position of the voice in the list of installed voices. You can refer to a voice by its index when using the `voiceSet()` command.

Use `voiceCount()` to determine the number of available voices.

Parameters

None.

Example

This statement sets the variable `oldVoice` to the property list describing the current text-to-speech voice:

```
-- Lingo
oldVoice = voiceGet()

// Javascript
Var oldVoice = voiceGet();
```

This statement displays the property list of the current text-to-speech voice:

```
-- Lingo
put voiceGet()
-- [#name: "Mary", #age: "teen", #gender: "female", #index: 5]

// Javascript
trace(voiceget())
// <[#name: "Mary", #age: "teen", #gender: "female", #index: 5]>
```

See also

`voiceInitialize()`, `voiceCount()`, `voiceSet()`, `voiceGet()`

voiceGetAll()

Usage

```
voiceGetAll()
```

Description

Function; returns a list of the available voices installed on the computer. The list is composed of property lists, one for each available voice.

Each property list contains the following properties:

- `#name` indicates the name of the installed voice.

- `#age` indicates the age of the voice. The value is a string. Possible values include "Teen", "Adult", "Toddler", and "Senior", as well as numeric values such as "35". Actual values depend on the operating system, speech software version, and voices installed.
- `#gender` indicates wether the voice is male or female.
- `#index` indicates the position of the voice in the list of installed voices. You can refer to a voice by its index when using the `voiceSet()` command.

You can also use `voiceCount()` to determine the number of available voices.

Parameters

None.

Example

This statement sets the variable `currentVoices` to the list of voices installed on the user's computer:

```
-- Lingo
currentVoices = voiceGetAll()

// Javascript
Var currentVoices = voiceGetAll();
```

This statement displays the property list describing each of the currently installed text-to-speech voices:

```
-- Lingo
put voiceGetAll()
-- [[#name: "Mary", #age: "teen", #gender: "female", #index: 1], [#name: "Joe", #age:
"adult", #gender: "male", #index: 2]]

// Javascript
trace(voiceGetAll());
// <[[#name: "Mary", #age: "teen", #gender: "female", #index: 1], [#name: "Joe", #age:
"adult", #gender: "male", #index: 2]]>
```

See also

`voiceInitialize()`, `voiceCount()`, `voiceSet()`, `voiceGet()`

voiceGetPitch()

Usage

```
voiceGetPitch()
```

Description

Function; returns the current pitch for the current voice as an integer. The valid range of values depends on the operating system platform and text-to-speech software.

Parameters

None.

Example

These statements check whether the pitch of the current voice is above 10 and set it to 10 if it is:

```
-- Lingo syntax
if voiceGetPitch() > 10 then
```

```
    voiceSetPitch(10)
end if

// JavaScript syntax
if (voiceGetPitch() > 10) {
    voiceSetPitch(10);
}
```

See also

voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceGetRate(), voiceSetRate(), voiceSetPitch(), voiceGetVolume(), voiceSetVolume(), voiceState(), voiceWordPos()

voiceGetRate()

Usage

```
voiceGetRate()
```

Description

Function; returns the current playback rate of the text-to-speech engine. The return value is an integer. The valid range of values depends on the text-to-speech software and operating system platform. In general, values between -10 and 10 can be expected.

Parameters

None.

Example

These statements check whether the rate of speech synthesis is below 50 and set it to 50 if it is:

```
-- Lingo syntax
if voiceGetRate() < 50 then
    voiceSetRate(50)
end if

// JavaScript syntax
if (voiceGetRate() < 50) {
    voiceSetRate(50);
}
```

See also

voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceSetRate(), voiceGetPitch(), voiceSetPitch(), voiceGetVolume(), voiceSetVolume(), voiceState(), voiceWordPos()

voiceGetVolume()

Usage

```
voiceGetVolume()
```

Description

Function: returns the current volume of the text-to-speech synthesis. The value returned is an integer. The valid range of values depends on the operating system platform.

Parameters

None.

Example

These statements check whether the text-to-speech volume is at least 55 and set it to 55 if is lower:

```
-- Lingo syntax
if voiceGetVolume() < 55 then
    voiceSetVolume(55)
end if

// JavaScript syntax
if (voiceGetVolume() < 55) {
    voiceSetVolume(55);
}
```

See also

voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceGetRate(), voiceSetRate(), voiceGetPitch(), voiceSetPitch(), voiceSetVolume(), voiceState(), voiceWordPos()

voiceInitialize()

Usage

```
voiceInitialize()
```

Description

Command; loads the computer's text-to-speech engine. If the `voiceInitialize()` command returns 0, text-to-speech software is not present or failed to load.

The command returns 1 if successful, 0 otherwise.

Parameters

None.

Example

These statements load the computer's text-to-speech engine and then test for whether the text-to-speech engine has completed loading before using the `voiceSpeak()` command to speak the phrase "Welcome to Shockwave.":

```
-- Lingo syntax
err = voiceInitialize()
if err = 1 then
    voiceSpeak("Welcome to Shockwave")
else
    alert "Text-to-speech software failed to load."
end if

// JavaScript syntax
err = voiceInitialize();
if (err == 1) {
    voiceSpeak("Welcome to Shockwave");
} else {
    alert("Text-to-speech software failed to load.");
}
```

See also

voiceCount(), voiceSet(), voiceGet()

voicePause()

Usage

```
voicePause()
```

Description

Command; pauses the speech output to the text-to-speech engine. The command returns a value of 1 if it is successful, or 0 if it is not.

Parameters

None.

Example

These statements cause the text-to-speech engine to pause when the user clicks the mouse:

```
-- Lingo syntax
on mouseUp
    voicePause()
end mouseUp

// JavaScript syntax
function mouseUp() {
    voicePause();
}
```

See also

voiceSpeak(), voiceResume(), voiceStop(), voiceGetRate(), voiceSetRate(), voiceGetPitch(), voiceSetPitch(), voiceGetVolume(), voiceSetVolume(), voiceState(), voiceWordPos()

voiceResume()

Usage

```
voiceResume()
```

Description

Command; resumes the speech output to the text-to-speech engine. The command returns a value of 1 if it is successful, or 0 if it is not.

Parameters

None.

Example

These statements resume the speech when the playhead moves to the next frame in the Score:

```
-- Lingo syntax
on exitFrame
    voiceResume()
```

```
end exitFrame

// JavaScript syntax
function exitFrame() {
    voiceResume();
}
```

See also

voiceSpeak(), voicePause(), voiceStop(), voiceGetRate(), voiceSetRate(), voiceGetPitch(), voiceSetPitch(), voiceGetVolume(), voiceSetVolume(), voiceState(), voiceWordPos()

voiceSet()

Usage

```
voiceSet(integer)
```

Description

Command: Sets the current voice of the text-to-speech synthesis. If successful, the command returns the new value that was set. Use `voiceCount()` to determine the number of available voices.

Parameters

integer Required. An integer that specifies the number of the text-to-speech voice to use. The valid range of values depends on the number of voices installed on the user's computer. If an out-of-range value is specified, the voice is set to the nearest valid value.

Example

This statement sets the current text-to-speech voice to the third voice installed on the user's computer:

```
voiceSet(3)
```

See also

voiceInitialize(), voiceCount(), voiceGet()

voiceSetPitch()

Usage

```
voiceSetPitch(integer)
```

Description

Command; sets the pitch for the current voice of the text-to-speech engine to the specified value. The return value is the new pitch value that has been set.

Parameters

integer Required. An integer that specifies the pitch for the text-to-speech voice. The valid range of values depends on the operating system platform and text-to-speech software.

Example

This statement sets the pitch for the current voice to 75:

```
-- Lingo
voiceSetPitch(75)

// Javascript
voiceSetPitch(75);
```

See also

voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceGetRate(), voiceSetRate(), voiceGetPitch(), voiceGetVolume(), voiceSetVolume(), voiceState(), voiceWordPos()

voiceSetRate()

Usage

```
voiceSetRate(integer)
```

Description

Command; sets the playback rate of the text-to-speech engine to the specified integer value. The command returns the new value that has been set.

Parameters

integer Required. An integer that specifies the playback rate that the text-to-speech engine uses. The valid range of values depends on the operating system platform. In general, values between -10 and 10 are appropriate for most text-to-speech software. If an out-of-range value is specified, the rate will be set to the nearest valid value.

Example

This statement sets the playback rate of the text-to-speech engine to 7:

```
-- Lingo
voiceSetRate(7)

// Javascript
voiceSetRate(7);
```

See also

voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceGetRate(), voiceGetPitch(), voiceSetPitch(), voiceGetVolume(), voiceSetVolume(), voiceState(), voiceWordPos()

voiceSetVolume()

Usage

```
voiceSetVolume(integer)
```

Description

Command; sets the volume of the text-to-speech synthesis.

Parameters

integer Required. An integer that specifies the volume of text-to-speech synthesis. The range of valid values depends on the operating system platform. If successful, the command returns the new value that was set. If an invalid value is specified, the volume is set to the nearest valid value.

Example

This statement sets the volume of text-to-speech synthesis to 55:

```
-- Lingo
voiceSetVolume(55)

// Javascript
voiceSetVolume(55);
```

See also

voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceGetRate(), voiceSetRate(), voiceGetPitch(), voiceSetPitch(), voiceGetVolume(), voiceState(), voiceWordPos()

voiceSpeak()

Usage

```
-- Lingo syntax
voiceSpeak("string")

// JavaScript syntax
voiceSpeak("string"); // documentation n/a
```

Description

Command; causes the specified string to be spoken by the text-to-speech engine. When this command is used, any speech currently in progress is interrupted by the new string.

Parameters

string Required. The string to be spoken by the text-to-speech engine.

Example

This statement causes the text-to-speech engine to speak the string "Welcome to Shockwave":

```
voiceSpeak("Welcome to Shockwave")
```

See also

voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceGetRate(), voiceSetRate(), voiceGetPitch(), voiceSetPitch(), voiceGetVolume(), voiceSetVolume(), voiceState(), voiceWordPos()

voiceState()

Usage

```
-- Lingo syntax
voiceState()

// JavaScript syntax
voiceState(); // documentation n/a
```

Description

Function; returns the current status of the voice as a symbol. The possible return values are `#playing`, `#paused`, and `#stopped`.

Parameters

None.

Example

These statements check whether the text-to-speech engine is actively speaking and set the voice to 1 if it is not:

```
--Lingo syntax
if voiceState() <> #playing then
    voiceSet(1)
end if

// JavaScript syntax
if (voiceState() != symbol("playing")) {
    voiceSet(1);
}
```

See also

`voiceSpeak()`, `voicePause()`, `voiceResume()`, `voiceStop()`, `voiceGetRate()`, `voiceSetRate()`, `voiceGetPitch()`, `voiceSetPitch()`, `voiceGetVolume()`, `voiceSetVolume()`, `voiceWordPos()`, `voiceSpeak()`

voiceStop()

Usage

```
-- Lingo syntax
voiceStop()

// JavaScript syntax
voiceStop(); // documentation n/a
```

Description

Command; stops the speech output to the text-to-speech engine and empties the text-to-speech buffer. The command returns a value of 1 if it is successful, or 0 if it is not.

Parameters

None.

Example

These statements stop the speech when the playhead moves to the next frame in the Score:

```
-- Lingo syntax
on exitFrame
    voiceStop()
end exitFrame

// JavaScript syntax
function exitFrame() {
    voiceStop();
}
```

See also

voiceSpeak(), voicePause(), voiceResume(), voiceGetRate(), voiceSetRate(), voiceGetPitch(), voiceSetPitch(), voiceGetVolume(), voiceSetVolume(), voiceState(), voiceWordPos(), voiceSpeak()

voiceWordPos()

Usage

```
-- Lingo syntax
voiceWordPos()

// JavaScript syntax
voiceWordPos(); // documentation n/a
```

Description

Function; returns an integer indicating the position of the word that is currently being spoken within the entire string that contains it. For example, if a cast member containing 15 words is being spoken and the fifth word of the cast member is being spoken when the function is used, the return value is 5.

Parameters

None.

Example

The following statements cause the sentence "Hello, how are you?" to be spoken and display the current word position in the Message window. Since the `voiceWordPos()` function is called immediately after the `voiceSpeak()` command is used, the return value will be 1.

```
-- Lingo syntax
voiceSpeak("Hello, how are you?")
put voiceWordPos()
-- 1

// JavaScript syntax
voiceSpeak("Hello, how are you?");
put(voiceWordPos());
// 1
```

See also

voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceGetRate(), voiceSetRate(), voiceGetPitch(), voiceSetPitch(), voiceGetVolume(), voiceSetVolume(), voiceState(), voiceSpeak()

voidP()

Usage

```
-- Lingo syntax
voidP(variableName)

// JavaScript syntax
variableName == null
```

Description

Function; determines whether a specified variable has any value. If the variable has no value or is VOID, this function returns TRUE. If the variable has a value other than VOID, this function returns FALSE.

Parameters

variableName Required. Specifies the variable to test.

Example

This statement checks whether the variable answer has an initial value:

```
-- Lingo syntax
put voidP(answer)

// JavaScript syntax
put(answer == null));
```

See also

ilk(), VOID

window()

Usage

```
-- Lingo syntax
window(stringWindowName)

// JavaScript syntax
window(stringWindowName);
```

Description

Top level function; returns a reference to a specified window.

The specified window must contain a Director movie.

Windows that play movies are useful for creating floating palettes, separate control panels, and windows of different shapes. Using windows that play movies, you can have several movies open at once and allow them to interact.

Parameters

stringWindowName Required. A string that specifies the name of the window to reference.

Example

This statement sets the variable myWindow to the window named Collections:

```
-- Lingo syntax
myWindow = window("Collections")

// JavaScript syntax
var myWindow = window("Collections");
```

See also

Window

WindowOperation

Usage

```
WindowOperation(MUIObject, operation)
```

Description

This command controls the window for a general purpose dialog box.

Replace the operation parameter with a value that determines what the window does. Possible values and their result are:

Possible values	Result
#show	Displays a non-modal dialog box only. (To open a modal dialog box, use the Run command.)
#hide	Hides a non-modal dialog box. (To close a modal dialog box, use the Stop command.)
#center	Centers the window on the monitor screen
#zoom	Sends a message that the user clicked the zoom box on the window. The callback handler must resize the dialog box, if you want the window to resize after the user clicks the zoom box.
#tipsOn	Turns tool tips on. (This is reserved for future versions of the MUI Xtra.)
#tipsOff	Turns tool tips off. (This is reserved for future versions of the MUI Xtra.)

Example

This handler checks whether MUIObject exists and displays the dialog box if the does:

```
--Lingo syntax
on showDialog
   global MUIObject
   if objectP( MUIObject ) then
      WindowOperation( MUIObject, #show )
   end if
end showDialog
```

This statement hides the dialog box created from MUIObject:

```
WindowOperation(MUIObject, #hide)
```

windowPresent()

Usage

```
-- Lingo syntax
_player.windowPresent(stringWindowName)

// JavaScript syntax
_player.windowPresent(stringWindowName);
```

Description

Player method; indicates whether the object specified by *`stringWindowName`* is running as a movie in a window (`TRUE`) or not (`FALSE`).

If a window had been opened, `windowPresent()` remains `TRUE` for the window until the window has been removed from the `windowList` property.

The *stringWindowName* argument must be the window's name as it appears in the windowList property.

Parameters

stringWindowName Required. A string that specifies the name of the window to test.

Example

This statement tests whether the object myWindow is a movie in a window (MIAW) and then displays the result in the Message window:

```
-- Lingo syntax
put(_player.windowPresent(myWindow))

// JavaScript syntax
put(_player.windowPresent(myWindow));
```

See also

Player, windowList

worldSpaceToSpriteSpace

Usage

```
-- Lingo syntax
member(whichCastmember).camera(whichCamera).worldSpaceToSpriteSpace(vector)

// JavaScript syntax
member(whichCastmember).camera(whichCamera).worldSpaceToSpriteSpace(vector);
```

Description

3D command; returns the point within the camera's rect at which a specified world-relative position would appear. The position returned by this command is relative to the upper left corner of the camera's rect.

If the position specified is out of view of the camera, this command returns void.

Parameters

vector Required. Specifies the world-relative position that would appear.

Example

This statement shows that the world origin, specified by vector (0, 0, 0), appears at point (250,281) within the camera's rect:

```
-- Lingo syntax
put sprite(5).camera.worldSpaceToSpriteSpace(vector(0, 0, 0))
-- point(250, 281)

// JavaScript syntax
put(sprite(5).camera.worldSpaceToSpriteSpace(vector(0,0,0)));
```

See also

spriteSpaceToWorldSpace, rect (camera)

writeChar()

Usage

```
-- Lingo syntax
fileioObjRef.writeChar(stringChar)

// JavaScript syntax
fileioObjRef.writeChar(stringChar)
```

Description

Fileio method; Writes a single specified ASCII character to a file.

You must first open a file by calling `openFile()` before using `writeChar()` to write a character.

Parameters

`stringChar` Required. Specifies the ASCII character to write to the file.

Example

The following statement opens a file c:\xtra.txt with Read/Write permission and writes the character “d” in the file and closes the file.

```
-- Lingo
objFileio = new xtra("fileio")
objFileio.openFile(“c:\xtra.txt”,0)
objFileio.writeChar(“d”)
objFileio.closeFile()

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile(“c:\xtra.txt”,0);
objFileio.writeChar(“d”);
objFileio.closeFile();
```

See also

Fileio

writeReturn()

Usage

```
-- Lingo syntax
fileioObjRef.writeReturn(symbol(""))

// JavaScript syntax
fileioObjRef.writeReturn(symbol(""));
```

Description

Fileio method; Inserts a line return in a file.

Parameters

None.

Example

The following statement opens a file c:\xtra.txt with Read/Write permission and writes the line “First Line” in the file followed by a line break and the second line “Second Line” and closes the file.

```
-- Lingo
objFileio = new xtra("fileio")
objFileio.openFile(“c:\xtra.txt”,0)
objFileio.writeString(“First Line”)
objFileio.writeReturn(#windows)
objFileio.writeString(“Second Line”)
objFileio.closeFile()

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile(“c:\xtra.txt”,0);
objFileio.writeString(“First Line”);
objFileio.writeReturn(Symbol(“windows”));
objFileio.writeString(“Second Line”);
objFileio.closeFile();
```

See also

Fileio

writeString()

Usage

```
-- Lingo syntax
fileioObjRef.writeString(string)

// JavaScript syntax
fileioObjRef.writeString(string)
```

Description

Fileio method; Writes a null-terminated string to a file.

Parameters

string Required. The string to write to a file.

Example

The following statement opens a file c:\xtra.txt with Read/Write permission and writes the line “First Line” and the second line “Second Line” and closes the file.

```
-- Lingo
objFileio = new xtra("fileio")
objFileio.openFile(“c:\xtra.txt”,0)
objFileio.writeString(“First Line”)
objFileio.writeString(“Second Line”)
objFileio.closeFile()

// JavaScript syntax
var objFileio = new xtra("fileio");
objFileio.openFile(“c:\xtra.txt”,0);
objFileio.writeString(“First Line”);
objFileio.writeString(“Second Line”);
objFileio.closeFile();
```

See also

Fileio

xtra()

Usage

```
-- Lingo syntax
xtra(xtraNameOrNum)

// JavaScript syntax
xtra(xtraNameOrNum);
```

Description

Top level function; returns an instance of a specified Xtra.

A reference to an empty object is returned if the specified Xtra is not found.

To see an example of `xtra` used in a completed movie, see the Read and Write Text movie in the Learning/Lingo folder inside the Director application folder.

Parameters

`xtraNameOrNum` Required. A string that specifies the name of the Xtra to return, or an integer that specifies the index position of the Xtra to return. String names are not case sensitive.

Example

This statement sets the variable `myNetLingo` to the NetLingo Xtra extension:

```
-- Lingo syntax
myNetLingo = xtra("netlingo")

// JavaScript syntax
var myNetLingo = xtra("netlingo");
```

zoomBox

Usage

```
-- Lingo syntax
zoomBox startSprite, endSprite {,delayTicks}

// JavaScript syntax
zoomBox(startSprite, endSprite {,delayTicks}); // not yet documented
```

Description

Command; creates a zooming effect, like the expanding windows in the Mac Finder. The zoom effect starts at a bounding rectangle of a specified starting sprite and finishes at the bounding rectangle of a specified ending sprite. The `zoomBox` command uses the following logic when executing:

1. Look for *`endSprite`* in the current frame: otherwise,
2. Look for *`endSprite`* in the next frame.

Note, however, that the `zoomBox` command does not work for *`endSprite`* if it is in the same channel as *`startSprite`*.

Parameters

`startSprite` Required. Specifies the starting sprite.

`endSprite` Required. Specifies the ending sprite.

`delayTicks` Optional. Specifies the delay in ticks between each movement of the zoom rectangles. If *`delayTicks`* is not specified, the delay is 1.

Example

This statement creates a zoom effect between sprites 7 and 3:

```
-- Lingo syntax
zoomBox 7, 3

// JavaScript syntax
zoomBox(7, 3); // not yet documented
```

Chapter 13: Operators

This section provides an alphabetical list of all the operators available in Director®.

The majority of these operators apply only to Lingo. JavaScript syntax does contain some operators that are either similar or identical to the Lingo operators listed here; therefore, where appropriate, JavaScript syntax usage and examples are provided to help you map the functionality of Lingo operators with their closest counterparts in JavaScript syntax. For more information about JavaScript syntax operators, see "Director Scripting Essentials" on page 4.

(symbol)

Usage

```
--Lingo syntax
#symbolName

// JavaScript syntax
symbol("symbolName");
```

Description

Symbol operator; defines a symbol, a self-contained unit that can be used to represent a condition or flag. The value *symbolName* begins with an alphabetical character and may be followed by any number of alphabetical or numerical characters.

A symbol can do the following:

- Assign a value to a variable.
- Compare strings, integers, rectangles, and points.
- Pass a parameter to a handler or method.
- Return a value from a handler or method.

A symbol takes up less space than a string and can be manipulated, but unlike a string it does not consist of individual characters. You can convert a symbol to a string for display purposes by using the `string` function.

The following are some important points about symbol syntax:

- Symbols are not case-sensitive.
- Symbols can't start with a number.
- Spaces may not be used, but you can use underscore characters to simulate them.
- Symbols use the 128 ASCII characters, and letters with diacritical or accent marks are treated as their base letter.
- Periods may not be used in symbols.

All symbols, global variables, and names of parameters passed to global variables are stored in a common lookup table.

Example

This statement sets the state variable to the symbol `#Playing`:

```
-- Lingo syntax
```

```
state = #Playing

// JavaScript syntax
var state = symbol("Playing");
```

See also

ilk(), string(), symbol(), symbolP()

. (dot operator)

Usage

```
-- Lingo syntax
objectReference.objectProperty
textExpression.objectProperty
object.commandOrFunction()

// JavaScript syntax
objectReference.objectProperty;
textExpression.objectProperty;
object.commandOrFunction();
```

Description

Operator; used to test or set properties of objects, or to issue a command or execute a function of the object. The object may be a cast member, a sprite, a property list, a child object of a parent script, or a behavior.

Example

This statement displays the current member contained by the sprite in channel 10:

```
-- Lingo syntax
put(sprite(10).member)

// JavaScript syntax
put(sprite(10).member);
```

To use the alternate syntax and call a function, you can use this form:

```
-- Lingo syntax
myColorObject = color(124, 22, 233)
put(myColorObject.ilk())
-- #color

// JavaScript syntax
var myColorObject = color(124, 22, 233);
put(myColorObject.ilk());
// #color
```

- (minus)

Usage

```
-- Lingo syntax
(Negation): -expression
```

```
(Subtraction): expression1 - expression2

// JavaScript syntax
(Negation): -expression
(Subtraction): expression1 - expression2
```

Description

Math operator; when used for negation, - (minus) reverses the sign of the value of *expression*; when used for subtraction, - (minus) performs an arithmetic subtraction on two numerical expressions, subtracting *expression2* from *expression1*.

When used for negation, - (minus) is an arithmetic operator with a precedence level of 5.

When used for subtraction, both expressions are integers, the difference is an integer. When either or both expressions are floating-point numbers, the difference is a floating-point number. The - (minus) operator is an arithmetic operator with a precedence level of 3.

Example

(Negation): This statement reverses the sign of the expression 2 + 3:

```
-- Lingo syntax
put(-(2 + 3))

// JavaScript syntax
put(-(2 + 3));
```

The result is -5.

(Subtraction): This statement subtracts the integer 2 from the integer 5 and displays the result in the Message window:

```
-- Lingo syntax
put(5 - 2)

// JavaScript syntax
put(5 - 2);
```

The result is 3, which is an integer.

(Subtraction): This statement subtracts the floating-point number 1.5 from the floating-point number 3.25 and displays the result in the Message window:

```
-- Lingo syntax
put(3.25 - 1.5)

// JavaScript syntax
put(3.25 - 1.5);
```

The result is 1.75, which is a floating-point number.

-- (comment)

Usage

```
-- Lingo syntax
-- comment

// JavaScript syntax
```

```
// comment
```

Description

Comment delimiter; indicates the beginning of a script comment. On any line, anything that appears between the comment delimiter (double hyphen) and the end-of-line return character is interpreted as a comment rather than a Lingo statement.

Example

This handler uses a double hyphen to make the second, fourth, and sixth lines comments:

```
-- Lingo syntax
on resetColors
    -- This handler resets the sprite's colors.
    sprite(1).forecolor = 35
    -- bright red
    sprite(1).backcolor = 36
    -- light blue
end

// JavaScript syntax
function resetColors() {
    // this handler resets the sprite's colors
    sprite(1).forecolor = 35;
    // bright red
    sprite(1).backcolor = 36;
    // light blue
}
```

&, + (concatenation operator)

Usage

```
-- Lingo syntax
expression1 & expression2

// JavaScript syntax
expression1 + expression2
```

Description

String operator; performs a string concatenation of two expressions. If either *expression1* or *expression2* is a number, it is first converted to a string. The resulting expression is a string.

This is a string operator with a precedence level of 2.

Lingo allows you to use some commands and functions that take only one argument without parentheses surrounding the argument. When an argument phrase includes an operator, Lingo interprets only the first argument as part of the function, which may confuse Lingo.

Avoid this problem by placing parentheses around the entire phrase that includes an operator. The parentheses clear up Lingo's confusion by changing the precedence by which Lingo deals with the operator, causing Lingo to treat the two parts of the argument as one complete argument.

Example

This statement concatenates the strings "abra" and "cadabra" and displays the resulting string in the Message window:

```
-- Lingo syntax
put("abra" & "cadabra")

// JavaScript syntax
put("abra" + "cadabra");
```

The result is the string "abracadabra".

This statement concatenates the strings "$" and the content of the `price` variable and then assigns the concatenated string to the Price field cast member:

```
-- Lingo syntax
member("Price").text = "$" & price

// JavaScript syntax
member("Price").text = "$" + price;
```

&&, + (concatenation operator)

Usage

```
-- Lingo syntax
expression1 && expression2

// JavaScript syntax
expression1 + expression2
```

Description

String operator; concatenates two expressions, inserting a space character between the original string expressions. If either *`expression1`* or *`expression2`* is a number, it is first converted to a string. The resulting expression is a string.

This is a string operator with a precedence level of 2.

Example

This statement concatenates the strings "abra" and "cadabra" and inserts a space between the two:

```
-- Lingo syntax
put("abra" && "cadabra")

// JavaScript syntax
put("abra " + "cadabra");
```

The result is the string "abra cadabra".

This statement concatenates the strings "Today is" and today's date in the long format and inserts a space between the two:

```
-- Lingo syntax
put("Today is" && date())

// JavaScript syntax
put("Today is " + Date());
```

() (parentheses)

Usage

```
-- Lingo syntax
(expression)

// JavaScript syntax
(expression)
```

Description

Grouping operator; performs a grouping operation on an expression to control the order of execution of the operators in an expression. This operator overrides the automatic precedence order so that the expression within the parentheses is evaluated first. When parentheses are nested, the contents of the inner parentheses are evaluated before the contents of the outer ones.

This is a grouping operator with a precedence level of 5.

Be aware that Lingo allows you to use some commands and functions that take only one argument without parentheses surrounding the argument. When an argument phrase includes an operator, Lingo interprets only the first argument as part of the function, which may confuse Lingo.

For example, the `open window` command allows one argument that specifies which window to open. If you use the `&` operator to define a pathname and filename, Director interprets only the string before the `&` operator as the filename. For example, Lingo interprets the statement `open window the applicationPath & "theMovie"` as `(open window the applicationPath) & ("theMovie")`. Avoid this problem by placing parentheses around the entire phrase that includes an operator, as follows:

```
-- Lingo syntax
open window (the applicationPath & "theMovie")

// JavaScript syntax
window(the applicationPath + "theMovie").open();
```

Example

These statements use the grouping operator to change the order in which operations occur (the result appears below each statement):

```
-- Lingo syntax
put((2 + 3) * (4 + 5))
-- 45
put(2 + (3 * (4 + 5)))
-- 29
put(2 + 3 * 4 + 5)
-- 19

// JavaScript syntax
put((2 + 3) * (4 + 5));
// 45
put(2 + (3 * (4 + 5)));
// 29
put(2 + 3 * 4 + 5);
// 19
```

* (multiplication)

Usage

```
-- Lingo syntax
expression1 * expression2

// JavaScript syntax
expression1 * expression2
```

Description

Math operator; performs an arithmetic multiplication on two numerical expressions. If both expressions are integers, the product is an integer. If either or both expressions are floating-point numbers, the product is a floating-point number.

This is an arithmetic operator with a precedence level of 4.

Example

This statement multiplies the integers 2 and 3 and displays the result in the Message window:

```
-- Lingo syntax
put(2 * 3)

// JavaScript syntax
put(2 * 3);
```

The result is 6, which is an integer.

This statement multiplies the floating-point numbers 2.0 and 3.1414 and displays the result in the Message window:

```
-- Lingo syntax
put(2.0 * 3.1416)

// JavaScript syntax
put(2.0 * 3.1416);
```

The result is 6.2832, which is a floating-point number.

+ (addition)

Usage

```
-- Lingo syntax
expression1 + expression2

// JavaScript syntax
expression1 + expression2
```

Description

Math operator; performs an arithmetic sum on two numerical expressions. If both expressions are integers, the sum is an integer. If either or both expressions are floating-point numbers, the sum is a floating-point number.

This is an arithmetic operator with a precedence level of 4.

Example

This statement adds the integers 2 and 3 and then displays the result, 5, an integer, in the Message window:

```
-- Lingo syntax
put(2 + 3)

// JavaScript syntax
put(2 + 3);
```

This statement adds the floating-point numbers 2.5 and 3.25 and displays the result, 5.7500, a floating-point number, in the Message window:

```
-- Lingo syntax
put(2.5 + 3.25)

// JavaScript syntax
put(2.5 + 3.25);
```

+ (addition) (3D)

Usage

```
-- Lingo syntax
vector1 + vector2
vector + scalar

// JavaScript syntax
vector1 + vector2
vector + scalar
```

Description

3D vector operator; adds the components of two vectors, or adds the scalar value to each component of the vector and returns a new vector.

vector1 + *vector2* adds the components of *vector1* to the corresponding to components of *vector2* and returns a new vector.

vector + *scalar* adds the scalar value to each of the components of the vector and returns a new vector.

- (subtraction)

Usage

```
-- Lingo syntax
vector1 - vector2
vector - scalar

// JavaScript syntax
vector1 - vector2
vector - scalar
```

Description

3D vector operator; subtracts the components of *vector2* from the corresponding components of *vector1*, or subtracts the scalar value from each of the components and returns a new vector.

vector1 - *vector2* subtracts the values of *vector2* from the corresponding components in *vector1* and returns a new vector.

vector - *scalar* subtracts the value of the scalar from each of the components in the vector and returns a new vector.

* (multiplication)

Usage

```
-- Lingo syntax
vector1 * vector2
vector * scalar
transform * vector

// JavaScript syntax
vector1 * vector2
vector * scalar
transform * vector
```

Description

3D vector operator; multiplies the components of *vector1* by the corresponding components in *vector2*, and returns the dot product, or multiplies each of the components the vector by the scalar value and returns a new vector.

vector1 * *vector2* returns the dot product of the two vectors, which is not a new vector. This operation is the same as *vector1*.dotproduct.*vector2*.

vector * *scalar* multiplies each of the components in the vector by the scalar value and returns a new vector.

transform * *vector* multiplies the *transform* by the *vector* and returns a new vector. The new vector is the result of applying the positional and rotational changes defined by *transform* to the *vector*. Note that *vector* * *transform* is not supported.

See also

dotProduct()

/ (division)

Usage

```
-- Lingo syntax
expression1 / expression2

// JavaScript syntax
expression1 / expression2
```

Description

Math operator; performs an arithmetic division on two numerical expressions, dividing *expression1* by *expression2*. If both expressions are integers, the quotient is an integer. If either or both expressions are floating-point numbers, the quotient is a floating-point number.

This is an arithmetic operator with a precedence level of 4.

Example

This statement divides the integer 22 by 7 and then displays the result in the Message window:

```
-- Lingo syntax
put(22 / 7)

// JavaScript syntax
put(22 / 7);
```

The result is 3. Because both numbers in the division are integers, Lingo rounds the answer down to the nearest integer.

This statement divides the floating-point number 22.0 by 7.0 and then displays the result in the Message window:

```
-- Lingo syntax
put(22.0 / 7.0)

// JavaScript syntax
put(22.0 / 7.0);
```

The result is 3.1429, which is a floating-point number.

/ (division) (3D)

Usage

```
-- Lingo syntax
vector / scalar

// JavaScript syntax
vector / scalar
```

Description

3D vector operator; divides each of the vector components by the scalar value and returns a new vector.

< (less than)

Usage

```
-- Lingo syntax
expression1 < expression2

// JavaScript syntax
expression1 < expression2
```

Description

Comparison operator; compares two expressions and determines whether *expression1* is less than *expression2* (TRUE), or whether *expression1* is greater than or equal to *expression2* (FALSE).

This operator can compare strings, integers, floating-point numbers, rects, and points. Be aware that comparisons performed on rects or points are handled as if the terms were lists, with each element of the first list compared to the corresponding element of the second list.

This is a comparison operator with a precedence level of 1.

<= (less than or equal to)

Usage

```
-- Lingo syntax
expression1 <= expression2

// JavaScript syntax
expression1 <= expression2
```

Description

Comparison operator; compares two expressions and determines whether *expression1* is less than or equal to *expression2* (TRUE), or whether *expression1* is greater than *expression2* (FALSE).

This operator can compare strings, integers, floating-point numbers, rects, and points. Be aware that comparisons performed on rects or points are handled as if the terms were lists, with each element of the first list compared to the corresponding element of the second list.

This is a comparison operator with a precedence level of 1.

<> (not equal)

Usage

```
-- Lingo syntax
expression1 <> expression2

// JavaScript syntax
expression1 != expression2
```

Description

Comparison operator; compares two expressions, symbols, or operators and determines whether *expression1* is not equal to *expression2* (TRUE), or whether *expression1* is equal to *expression2* (FALSE).

This operator can compare strings, integers, floating-point numbers, rects, and points. Be aware that comparisons performed on rects or points are handled as if the terms were lists, with each element of the first list compared to the corresponding element of the second list.

This is a comparison operator with a precedence level of 1.

= (equals)

Usage

```
-- Lingo syntax
expression1 = expression2
```

```
// JavaScript syntax
expression1 = expression2
```

Description

Comparison operator; compares two expressions, symbols, or objects and determines whether *expression1* is equal to *expression2* (TRUE), or whether *expression1* is not equal to *expression2* (FALSE).

This operator can compare strings, integers, floating-point numbers, rects, lists, and points.

Lists are compared based on the number of elements in the list. The list with more elements is considered larger than the than the list with fewer elements.

This is a comparison operator with a precedence level of 1.

> (greater than)

Usage

```
-- Lingo syntax
expression1 > expression2

// JavaScript syntax
expression1 > expression2
```

Description

Comparison operator; compares two expressions and determines whether *expression1* is greater than *expression2* (TRUE), or whether *expression1* is less than or equal to *expression2* (FALSE).

This operator can compare strings, integers, floating-point numbers, rects, and points. Be aware that comparisons performed on rects or points are handled as if the terms were lists, with each element of the first list compared to the corresponding element of the second list.

This is a comparison operator with a precedence level of 1.

>= (greater than or equal to)

Usage

```
-- Lingo syntax
expression1 >= expression2

// JavaScript syntax
expression1 >= expression2
```

Description

Comparison operator; compares two expressions and determines whether *expression1* is greater than or equal to *expression2* (TRUE), or whether *expression1* is less than *expression2* (FALSE).

This operator can compare strings, integers, floating-point numbers, rects, and points. Be aware that comparisons performed on rectangles or points are handled as if the terms were lists, with each element of the first list compared to the corresponding element of the second list.

This is a comparison operator with a precedence level of 1.

[ ] (bracket access)

Usage

```
-- Lingo syntax
textExpression[chunkNumberBeingAddressed]
textExpression[firstChunk..lastChunk]
```

Description

Operator; allows a chunk expression to be addressed by number. Useful for finding the *n*th chunk in the expression. The chunk can be a word, line, character, paragraph, or other Text cast member chunk.

Example

This outputs the first word of the third line in the text cast member First Names:

```
-- Lingo syntax
put(member("First Names").text.line[3].word[1])

// JavaScript syntax
put(member("First Names").getPropRef("line", 1).getProp("word", 1));
```

[ ] (list)

Usage

```
[entry1, entry2, entry3, ...]
```

Description

List operator; specifies that the entries within the brackets are one of four types of lists:

- Unsorted linear lists
- Sorted linear lists
- Unsorted property lists
- Sorted property lists

Each entry in a linear list is a single value that has no other property associated with it. Each entry in a property list consists of a property and a value. The property appears before the value and is separated from the value by a colon. You cannot store a property in a linear list. When using strings as entries in a list, enclose the string in quotation marks.

For example, [6, 3, 8] is a linear list. The numbers have no properties associated with them. However, [#gears:6, #balls:3, #ramps:8] is a property list. Each number has a property—in this case, a type of machinery—associated with it. This property list could be useful for tracking the number of each type of machinery currently on the Stage in a mechanical simulation. Properties can appear more than once in a property list.

Lists can be sorted in alphanumeric order. A sorted linear list is ordered by the values in the list. A sorted property list is ordered by the properties in the list. You sort a list by using the appropriate command for a linear list or property list.

- In linear lists, symbols and strings are case sensitive.
- In property lists, symbols aren't case-sensitive, but strings are case-sensitive.

A linear list or property list can contain no values at all. An empty list consists of two square brackets ([ ]). To create or clear a linear list, set the list to `[ ]`. To create or clear a property list, set the list to [:].

You can modify, test, or read items in a list.

Lingo treats an instance of a list as a reference to the list. This means each instance is the same piece of data, and changing it will change the original. Use the `duplicate` command to create copies of lists.

Lists are automatically disposed when they are no longer referred to by any variable. When a list is held within a global variable, it persists from movie to movie.

You can initialize a list in the `on prepareMovie` handler or write the list as a field cast member, assign the list to a variable, and then handle the list by handling the variable.

Not all PC keyboards have square brackets. If square brackets aren't available, use the `list` function to create a linear list.

For a property list, create the list pieces as a string before converting them into a useful list.

```
myListString = numToChar(91) & ":" & numToChar(93)
put myListString
-- "[:]"
myList = myListString.value
put myList
-- [:]
put myList.listP
-- 1
myList[#name] = "Brynn"
put myList
-- [#name: "Brynn"]
```

Example

This statement defines a list by making the `machinery` variable equal to the list:

```
-- Lingo syntax
machinery = [#gears:6, #balls:3, #ramps:8]

// JavaScript syntax
var machinery = propList("gears",6, "balls",3, "ramps",8);
```

This handler sorts the list aList and then displays the result in the Message window:

```
-- Lingo syntax
on sortList aList
    alist.sort()
    put(aList)
end sortList

// JavaScript syntax
function sortList(aList) {
    aList.sort();
    put(aList);
}
```

If the movie issues the statement `sortList machinery`, where `machinery` is the list in the preceding example, the result is [#balls:3, #gears:6, #ramps:8].

The following statements create an empty linear list:

```
-- Lingo syntax
x = [ ]
x = list()
```

```
// JavaScript syntax
var x = list();
```

The following statements create an empty property list:

```
-- Lingo syntax
x = [:]
x = propList()

// JavaScript syntax
var x = propList();
```

See also

add, addVertex(), append, count(), deleteAt, duplicate() (list function), findPos, findPosNear, getProp(), getAt, getLast(), getPos(), ilk(), list(), max(), min, setAt, setaProp, sort

@ (pathname)

Usage

```
@pathReference
```

Description

Pathname operator; defines the path to the current movie's folder and is valid on both Windows® and Mac® computers.

Identify the current movie's folder by using the @ symbol followed by one of these pathname separators:

- / (forward slash)
- \ (backslash)
- : (colon)

When a movie is queried to determine its location, the string returned will include the @ symbol.

Be sure to use only the @ symbol when navigating between Director movies or changing the source of a linked media cast member. The @ symbol does not work when the Fileio Xtra extension or other functions are used outside those available within Director.

You can build on this pathname to specify folders that are one or more levels above or below the current movie's folder. Keep in mind that the @ portion represents the current movie's location, not necessarily the location of the projector.

- Add an additional pathname separator immediately after the @ symbol to specify a folder one level up in the hierarchy.
- Add folder names and filenames (separated by /, \, or :) after the current folder name to specify subfolders and files within folders.

You can use relative pathnames in Lingo to indicate the location of a linked file in a folder different than the movie's folder.

Example

These are equivalent expressions that specify the subfolder bigFolder, which is in the current movie's folder:

```
@/bigFolder
@:bigFolder
@\bigFolder
```

These are equivalent expressions that specify the file linkedFile, in the subfolder bigFolder, which is in the current movie's folder:

```
@:bigFolder:linkedFile
@\bigFolder\linkedFile
@/bigFolder/linkedFile
```

This expression specifies the file linkedFile, which is located one level up from the current movie's folder:

```
@//linkedFile
```

This expression specifies the file linkedFile, which is located two levels up from the current movie's folder:

```
@:::linkedFile
```

These are equivalent expressions that specify the file linkedFile, which is in the folder otherFolder. The otherFolder folder is in the folder one level up from the current movie's folder.

```
@::otherFolder:linkedFile
@\\otherFolder\linkedFile
@//otherFolder/linkedFile
```

See also

searchPathList, fileName (Cast), fileName (Member), fileName (Window)

and

Usage

```
-- Lingo syntax
logicalExpression1 and logicalExpression2

// JavaScript syntax
logicalExpression1 && logicalExpression2
```

Description

Logical operator; determines whether both *logicalExpression1* and *logicalExpression2* are TRUE (1), or whether either or both expressions are FALSE (0).

The precedence level of this logical operator is 4.

Example

This statement determines whether both logical expressions are TRUE and displays the result in the Message window:

```
-- Lingo syntax
put(1 < 2 and 2 < 3)

// JavaScript syntax
put((1 < 2) && (2 < 3));
```

The result is 1, which is the numerical equivalent of TRUE.

The first logical expression in the following statement is TRUE; and the second logical expression is FALSE. Because both logical expressions are not TRUE, the logical operator displays the result 0, which is the numerical equivalent of FALSE.

```
-- Lingo syntax
put(1 < 2 and 2 < 1)
-- 0

// JavaScript syntax
put((1 < 2) && (2 < 1));
// 0
```

See also

not, or

contains

Usage

```
-- Lingo syntax
stringExpression1 contains stringExpression2

// JavaScript syntax
stringExpression1.indexOf(stringExpression2);
```

Description

Operator; compares two strings and determines whether *stringExpression1* contains *stringExpression2* (TRUE) or not (FALSE).

The contains comparison operator has a precedence level of 1.

The contains comparison operator is useful for checking whether the user types a specific character or string of characters. You can also use the contains operator to search one or more fields for specific strings of characters.

Example

This example determines whether a character passed to it is a digit:

```
-- Lingo syntax
on isNumber aLetter
    digits = "1234567890"
    if digits contains aLetter then
        return TRUE
    else
        return FALSE
    end if
end

// JavaScript syntax
function isNumber(aLetter) {
    var digits = "1234567890"
    if (digits.indexOf(aLetter) >= 0) {
        return true;
    } else {
        return false;
    }
}
```

Note: *The string comparison is not sensitive to case or diacritical marks; "a" and Å are treated the same.*

See also

offset() (string function), starts

mod

Usage

```
-- Lingo syntax
integerExpression1 mod integerExpression2

// JavaScript syntax
integerExpression1 % integerExpression2
```

Description

Math operator; performs the arithmetic modulus operation on two integer expressions. In this operation, *integerExpression1* is divided by *integerExpression2*.

The resulting value of the entire expression is the integer remainder of the division. It always has the sign of *integerExpression1*.

This is an arithmetic operator with a precedence level of 4.

Example

This statement divides 7 by 4 and then displays the remainder in the Message window:

```
-- Lingo syntax
put(7 mod 4)

// JavaScript syntax
put(7 % 4);
```

The result is 3.

The following handler sets the ink effect of all odd-numbered sprites to `copy`, which is the ink effect specified by the number 0. First the handler checks whether the sprite in the variable `mySprite` is an odd-numbered sprite by dividing the sprite number by 2 and then checking whether the remainder is 1. If the remainder is 1, the result for an odd-numbered number, the handler sets the ink effect to `copy`.

```
-- Lingo syntax
on setInk
    repeat with mySprite = 1 to _movie.lastChannel
        if (mySprite mod 2) = 1 then
            sprite(mySprite).ink = 0
        else
            sprite(mySprite).ink = 8
        end if
    end repeat
end setInk

// JavaScript syntax
function setInk() {
    for (mySprite=1; mySprite<=_movie.lastChannel; mySprite++) {
        if ((mySprite % 2) == 1) {
            sprite(mySprite).ink = 0;
        } else {
            sprite(mySprite).ink = 8;
        }
    }
}
```

This handler regularly cycles a sprite's cast member among a number of bitmaps:

```
-- Lingo syntax
```

```
on exitFrame
    global gCounter
    -- These are sample values for bitmap cast member numbers
    theBitmaps = [2,3,4,5,6,7]
    -- Specify which sprite channel is affected
    theChannel = 1
    -- This cycles through the list
    gCounter = 1 + (gCounter mod theBitmaps.count)
    sprite(theChannel).memberNum = theBitmaps[gCounter]
    _movie.go(_movie.frame)
end

// JavaScript syntax
function exitFrame() {
    // these are sample values for bitmap cast member numbers
    theBitmaps = new Array(2,3,4,5,6,7);
    // specify which sprite channel is affected
    theChannel = 1;
    // this cycles through the list
    _global.gCounter = 1 + (_global.gCounter % theBitmaps.length);
    sprite(theChannel).memberNum = theBitmaps[_global.gCounter];
    _movie.go(_movie.frame);
}
```

not

Usage

```
-- Lingo syntax
not logicalExpression

// JavaScript syntax
! logicalExpression
```

Description

Operator; performs a logical negation on a logical expression. This is the equivalent of making a `TRUE` value `FALSE`, and making a `FALSE` value `TRUE`. It is useful when testing to see if a certain known condition is not the case.

This logical operator has a precedence level of 5.

Example

This statement determines whether 1 is not less than 2:

```
-- Lingo syntax
put(not (1 < 2))

// JavaScript syntax
put(!(1 < 2));
```

Because 1 is less than 2, the result is 0, which indicates that the expression is `FALSE`.

This statement determines whether 1 is not greater than 2:

```
-- Lingo syntax
put(not (1 > 2))

// JavaScript syntax
put(!(1 > 2));
```

Because 1 is not greater than 2, the result is 1, which indicates that the expression is TRUE.

This handler sets the checkMark menu item property for Bold in the Style menu to the opposite of its current setting:

```
-- Lingo syntax
on resetMenuItem
    menu("Style").menuItem("Bold").checkMark =
    not (menu("Style").menuItem("Bold").checkMark)
end resetMenuItem

// JavaScript syntax
function resetMenuItem() {
    menu("Style").menuItem("Bold").checkMark =
    !(menu("Style").menuItem("Bold").checkMark)
}
```

See also

and, or

or

Usage

```
-- Lingo syntax
logicalExpression1 or logicalExpression2

// JavaScript syntax
logicalExpression1 || logicalExpression2
```

Description

Operator; performs a logical OR operation on two or more logical expressions to determine whether any expression is TRUE.

This is a logical operator with a precedence level of 4.

Example

This statement indicates in the Message window whether at least one of the expressions 1 < 2 and 1 > 2 is TRUE:

```
-- Lingo syntax
put((1 < 2) or (1 > 2))

// JavaScript syntax
put((1 < 2) || (1 > 2));
```

Because the first expression is TRUE, the result is 1, which is the numerical equivalent of TRUE.

This Lingo checks whether the content of the field cast member named State is either AK or HI and displays an alert if it is:

```
-- Lingo syntax
if member("State").text = "AK" or member("State").text = "HI" then
    _player.alert("You're off the map!")
end if

// JavaScript syntax
if (member("State").text == "AK" || member("State").text == "HI") {
    _player.alert("You're off the map!");
```

```
}
```

See also

and, not

starts

Usage

```
-- Lingo syntax
string1 starts string2

// JavaScript syntax
string1.indexOf(string2) == 0;
```

Description

Operator; compares to determines whether *string1* starts with *string2* (TRUE or 1) or not (FALSE or 0).

The string comparison is not sensitive to case or diacritical marks; *a* and *Å* are considered to be the same.

This is a comparison operator with a precedence level of 1.

Example

This statement reports in the Message window whether the word *Macrostuff* starts with the string "Macro":

```
-- Lingo syntax
put("Macrostuff" starts "Macro")

// JavaScript syntax
var string1 = "Macrostuff";
put(string1.indexOf("Macro") == 0);
```

The result is 1, which is the numerical equivalent of TRUE.

See also

contains

Chapter 14: Properties

This section provides an alphabetical list of all the properties available in Director®.

_global

Usage

```
-- Lingo syntax
_global

// JavaScript syntax
_global;
```

Description

Top-level property; provides a reference to the Global object, which stores all global variables. Read-only.

All global variables are accessible to both Lingo and JavaScript syntax.

Example

This statement sets the variable `objGlobal` to the `_global` property:

```
-- Lingo syntax
objGlobal = _global

// JavaScript syntax
var objGlobal = _global;
```

This statement uses the `_global` property directly to clear all global variables:

```
-- Lingo syntax
_global.clearGlobals()

// JavaScript syntax
_global.clearGlobals();
```

See also

Global

_key

Usage

```
-- Lingo syntax
_key

// JavaScript syntax
_key;
```

Description

Top-level property; provides a reference to the Key object, which is used to monitor a user's keyboard activity. Read-only.

Example

This statement sets the variable `objKey` to the `_key` property:

```
-- Lingo syntax
objKey = _key

// JavaScript syntax
var objKey = _key;
```

This statement uses the `_key` property directly to access the value of the `key` property:

```
-- Lingo syntax
theKey = _key.key

// JavaScript syntax
var theKey = _key.key;
```

See also

Key

_mouse

Usage

```
-- Lingo syntax
_mouse

// JavaScript syntax
_mouse;
```

Description

Top-level property; provides a reference to the Mouse object, which provides access to a user's mouse activity, including mouse movement and mouse clicks. Read-only.

Example

This statement sets the variable `objMouse` to the `_mouse` property:

```
-- Lingo syntax
objMouse = _mouse

// JavaScript syntax
var objMouse = _mouse;
```

This statement uses the `_mouse` property directly to access the value of the `mouseH` property:

```
-- Lingo syntax
theMouseH = _mouse.mouseH

// JavaScript syntax
var theMouseH = _mouse.mouseH;
```

See also

Mouse

_movie

Usage

```
-- Lingo syntax
_movie

// JavaScript syntax
_movie;
```

Description

Top-level property; provides a reference to the Movie object, which represents the currently active movie within the Director player, and provides access to properties and methods that are available on a movie level. Read-only.

Example

This statement sets the variable `objMovie` to the `_movie` property:

```
-- Lingo syntax
objMovie = _movie

// JavaScript syntax
var objMovie = _movie;
```

This statement uses the `_movie` property directly to access the value of the `displayTemplate` property:

```
-- Lingo syntax
theTemplate = _movie.displayTemplate

// JavaScript syntax
var theTemplate = _movie.displayTemplate;
```

See also

Movie

_player

Usage

```
-- Lingo syntax
_player

// JavaScript syntax
_player;
```

Description

Top-level property; provides a reference to the Player object, which manages and executes all movies, including movies in a window (MIAWs). Read-only.

Example

This statement sets the variable `objPlayer` to the `_player` property:

```
-- Lingo syntax
objPlayer = _player

// JavaScript syntax
var objPlayer = _player;
```

This statement uses the _player property directly to access the value of the xtraList property:

```
-- Lingo syntax
theXtras = _player.xtraList

// JavaScript syntax
var theXtras = _player.xtraList;
```

See also

Player

_sound

Usage

```
-- Lingo syntax
_sound

// JavaScript syntax
_sound;
```

Description

Top-level property; provides a reference to the Sound object, which controls audio playback in all eight available sound channels. Read-only.

Example

This statement sets the variable objSound to the _sound property:

```
-- Lingo syntax
objSound = _sound

// JavaScript syntax
var objSound = _sound;
```

This statement uses the _sound property directly to access the soundLevel property:

```
-- Lingo syntax
theLevel = _sound.soundLevel

// JavaScript syntax
var theLevel = _sound.soundLevel;
```

See also

Sound

_system

Usage

```
-- Lingo syntax
_system

// JavaScript syntax
_system;
```

Description

Top-level property; provides a reference to the System object, which provides access to system and environment information, including system level methods. Read-only.

Example

This statement sets the variable `objSystem` to the `_system` property:

```
-- Lingo syntax
objSystem = _system

// JavaScript syntax
var objSystem = _system;
```

This statement uses the `_system` property directly to access the `freeBytes` property:

```
-- Lingo syntax
theBytes = _system.freeBytes

// JavaScript syntax
var theBytes = _system.freeBytes;
```

See also

System

aboutInfo

Usage

```
-- Lingo syntax
_movie.aboutInfo

// JavaScript syntax
_movie.aboutInfo;
```

Description

Movie property; a string entered during authoring in the Movie Properties dialog box. Read-only.

Example

These statements display movie information in the Message window.

```
-- Lingo syntax
trace(_movie.aboutInfo)

// JavaScript syntax
trace(_movie.aboutInfo);
```

See also

copyrightInfo (Movie), Movie

actionsEnabled

Usage

```
-- Lingo syntax
```

```
memberOrSpriteObjRef.actionsEnabled

// JavaScript syntax
memberOrSpriteObjRef.actionsEnabled;
```

Description

Cast member property and sprite property; controls whether the actions in Adobe® Flash® content are enabled (`TRUE`, default) or disabled (`FALSE`).

This property can be tested and set.

Example

This handler accepts a sprite reference as a parameter, and then toggles the sprite's `actionsEnabled` property on or off.

```
-- Lingo syntax
on ToggleActions(whichSprite)
     sprite(whichSprite).actionsEnabled = not(sprite(whichSprite).actionsEnabled)
end

// JavaScript syntax
function ToggleActions(whichSprite) {
     switch(sprite(whichSprite).actionsEnabled){
        case 0:
             sprite(whichSprite).actionsEnabled = 1;
            break;
        case 1:
            sprite(whichSprite).actionsEnabled = 0;
            break;
    }
}
```

active3dRenderer

Usage

```
-- Lingo syntax
_movie.active3dRenderer

// JavaScript syntax
_movie.active3dRenderer;
```

Description

Movie property; Indicates the renderer currently in use by the movie for drawing 3D sprites. This property is equivalent to the `getRendererServices().renderer` property. Read-only.

The possible values of the `active3dRenderer` property are `#openGL`, `#directX7_0, #directx9`, `#directX5_2`, and `#software`. The values `#openGL`, `#directX7_0`, `#directx9,` and `#directX5_2`, which are video card drivers, will lead to much faster performance than `#software`, a software renderer used when none of the first three options are available.

Use `getRendererServices().renderer` to set this property.

Example

These examples show the two ways to determine which renderer is currently in use.

```
-- Lingo syntax
put(_movie.active3dRenderer)
put(getRendererServices().renderer)

// JavaScript syntax
put(_movie.active3dRenderer);
put(getRendererServices().renderer);
```

See also

Movie, renderer

activeCastLib

Usage

```
-- Lingo syntax
_movie.activeCastLib

// JavaScript syntax
_movie.activeCastLib;
```

Description

Movie property; indicates which cast library was most recently activated. Read-only.

The `activeCastLib` property's value is the cast library's number.

The `activeCastLib` property is useful when working with the Cast object's `selection` property. Use it to determine which cast library the selection refers to.

Example

These statements assign the selected cast members in the most recently selected cast to the variable `selectedMembers`:

```
-- Lingo syntax
castLibOfInterest = _movie.activeCastLib
selectedMembers = castLib(castLibOfInterest).selection

// JavaScript syntax
var castLibOfInterest = _movie.activeCastLib;
var selectedMembers = castLib(castLibOfInterest).selection;
```

See also

Player, selection

activeWindow

Usage

```
-- Lingo syntax
_player.activeWindow

// JavaScript syntax
_player.activeWindow;
```

Description

Player property; indicates which movie window is currently active. Read-only.

For the main movie, `activeWindow` is the Stage. For a movie in a window (MIAW), `activeWindow` is the movie in the window.

Example

This example places the word Active in the title bar of the clicked window and places the word Inactive in the title bar of all other open windows:

```
-- Lingo syntax
on activateWindow
    clickedWindow = _player.windowList.getPos(_player.activeWindow)
    windowCount = _player.windowList.count
    repeat with x = 1 to windowCount
        if (x = clickedWindow) then
            _player.window[clickedWindow].title = "Active"
        else
            _player.windowList[x].title = "Inactive"
        end if
    end repeat
end activateWindow

// JavaScript syntax
function activateWindow() {
    var clickedWindow = _player.windowList.getPos(_player.activeWindow);
    var windowCount = _player.windowList.count;
    for (var x = 1; x <= windowCount; x++) {
        if (x == clickedWindow) {
            _player.window[clickedWindow].title = "Active"
        }
        else {
            _player.windowList[x].title = "Inactive"
        }
    }
}
```

See also

Player

actorList

Usage

```
-- Lingo syntax
_movie.actorList

// JavaScript syntax
_movie.actorList;
```

Description

Movie property; a list of child objects that have been explicitly added to this list. Read/write.

Objects in `actorList` receive a `stepFrame` message each time the playhead enters a frame.

To add an object to `the actorList`, use `_movie.actorList.append(`*`newScriptObjRef`*`)`. The object's `stepFrame` handler in its parent or ancestor script will then be called automatically at each frame advance.

To clear objects from `the actorList`, set `actorList` to `[ ]`, which is an empty list.

Director doesn't clear the contents of `actorList` when branching to another movie, which can cause unpredictable behavior in the new movie. To prevent child objects in the current movie from being carried over to the new movie, insert the statement `actorList = [ ]` in the `prepareMovie` handler of the new movie.

Example

This statement adds a child object created from the parent script Moving Ball. All three values are parameters that the script requires.

This statement displays the contents of `actorList` in the Message window:

```
-- Lingo syntax
put(_movie.actorList)

// JavaScript syntax
put(_movie.actorList);
```

This statement clears objects from `actorList`.

```
-- Lingo syntax
_movie.actorList = [] -- using brackets
_movie.actorList = list() -- using list()

// JavaScript syntax
_movie.actorList = list();
```

See also

Movie, on prepareMovie, on stepFrame

alertHook

Usage

```
-- Lingo syntax
_player.alertHook

// JavaScript syntax
_player.alertHook;
```

Description

Player property; specifies a parent script that contains the `alertHook` handler. Read/write.

Use `alertHook` to control the display of alerts about file errors or script errors. When an error occurs and a parent script is assigned to `alertHook`, Director runs the `alertHook` handler in the parent script.

Although it is possible to place `alertHook` handlers in movie scripts, it is strongly recommended that you place an `alertHook` handler in a behavior or parent script to avoid unintentionally calling the handler from a wide variety of locations and creating confusion about where the error occurred.

Because the `alertHook` handler runs when an error occurs, avoid using the `alertHook` handler for script that isn't involved in handling an error. For example, the `alertHook` handler is a bad location for a `go()` statement.

The `alertHook` handler is passed an instance argument, two string arguments that describe the error, and an optional argument specifying an additional event that invokes the handler.

The fourth argument can have 1 of these 4 values:

- `#alert`—causes the handler to be triggered by the `alert()` method.
- `#movie`—causes the handler to be triggered by a file not found error while performing a `go()` command.
- `#script`—causes the handler to be triggered by a script error.
- `#safeplayer`—causes the handler to be triggered by a check of the `safePlayer` property.

Depending on the script within it, the `alertHook` handler can ignore the error or report it in another way.

Example

The following statement specifies that the parent script Alert is the script that determines whether to display alerts when an error occurs. If an error occurs, the script assigns the error and message strings to the field cast member Output and returns the value 1.

```
-- Lingo syntax
on prepareMovie
    _player.alertHook = script("Alert")
end

-- "Alert" script
on alertHook me, err, msg
    member("Output").text = err && msg
    return 1
end

// JavaScript syntax
function prepareMovie() {
     _player.alertHook = "alert("Error type", "Error message");"
}

// alert handler
function alert(err, msg) {
    member("Output").text = err + " " + msg; return 1;
}
```

See also

alertHook, Player, safePlayer

alignment

Usage

```
-- Lingo syntax
memberObjRef.alignment

// JavaScript syntax
memberObjRef.alignment;
```

Description

Cast member property; determines the alignment used to display characters within the specified cast member. This property appears only to field and text cast members containing characters, if only a space.

For field cast members, the value of the property is a string consisting of one of the following: left, center, or right.

For text cast members, the value of the property is a symbol consisting of one of the following: #left, #center, #right, or #full.

The parameter *whichCastMember* can be either a cast name or a cast number.

This property can be tested and set. For text cast members, the property can be set on a per-paragraph basis.

Example

This statement sets the variable named `characterAlign` to the current `alignment` setting for the field cast member Rokujo Speaks:

```
--Lingo syntax
characterAlign = member("Rokujo Speaks").alignment

// JavaScript syntax
var characterAlign = member("Rokujo Speaks").alignment;
```

See also

text, font, lineHeight, fontSize, fontStyle, &, + (concatenation operator), &&, + (concatenation operator)

allowCustomCaching

Usage

```
-- Lingo syntax
_movie.allowCustomCaching

// JavaScript syntax
_movie.allowCustomCaching;
```

Description

Movie property; will contain information regarding a private cache in future versions of Director. Read/write.

This property defaults to `TRUE`.

See also

allowGraphicMenu, allowSaveLocal, allowTransportControl, allowVolumeControl, allowZooming, Movie

allowGraphicMenu

Usage

```
-- Lingo syntax
_movie.allowGraphicMenu

// JavaScript syntax
_movie.allowGraphicMenu;
```

Description

Movie property; sets the availability of the graphic controls in the context menu when playing the movie in a Adobe Shockwave® environment. Read/write.

Set this property to `FALSE` if you would rather have a text menu displayed than the graphic context menu.

This property defaults to TRUE.

See also

allowCustomCaching, allowSaveLocal, allowTransportControl, allowVolumeControl, allowZooming, Movie

allowSaveLocal

Usage

```
-- Lingo syntax
_movie.allowSaveLocal

// JavaScript syntax
_movie.allowSaveLocal;
```

Description

Movie property; sets the availability of the Save control in the context menu when playing the movie in a Shockwave® Player environment. Read/write.

This property is provided to allow for enhancements in future versions of Shockwave Player.

This property defaults to TRUE.

See also

allowCustomCaching, allowGraphicMenu, allowTransportControl, allowVolumeControl, allowZooming, Movie

allowTransportControl

Usage

```
-- Lingo syntax
_movie.allowTransportControl

// JavaScript syntax
_movie.allowTransportControl;
```

Description

Movie property; this property is provided to allow for enhancements in future versions of Shockwave Player. Read/write.

This property defaults to TRUE.

See also

allowCustomCaching, allowGraphicMenu, allowSaveLocal, allowVolumeControl, allowZooming, Movie

allowVolumeControl

Usage

```
-- Lingo syntax
_movie.allowVolumeControl

// JavaScript syntax
_movie.allowVolumeControl;
```

Description

Movie property; sets the availability of the volume control in the context menu when playing the movie in a Shockwave Player environment. Read/write.

When set to TRUE one or the other volume control is active, and is disabled when the property is set to FALSE.

This property defaults to TRUE.

See also

allowCustomCaching, allowGraphicMenu, allowSaveLocal, allowTransportControl, allowZooming, Movie

allowZooming

Usage

```
-- Lingo syntax
_movie.allowZooming

// JavaScript syntax
_movie.allowZooming;
```

Description

Movie property; determines whether the movie may be stretched or zoomed by the user when playing back in Shockwave Player and ShockMachine. Read/write.

Set this property to FALSE to prevent users from changing the size of the movie in browsers and ShockMachine.

The property defaults to TRUE.

See also

allowCustomCaching, allowGraphicMenu, allowSaveLocal, allowTransportControl, allowVolumeControl, Movie

alphaThreshold

Usage

```
-- Lingo syntax
memberObjRef.alphaThreshold

// JavaScript syntax
memberObjRef.alphaThreshold;
```

Description

Bitmap cast member property; governs how the bitmap's alpha channel affects hit detection. This property is a value from 0 to 255, that exactly matches alpha values in the alpha channel for a 32-bit bitmap image.

For a given `alphaThreshold` setting, Director detects a mouse click if the pixel value of the alpha map at that point is equal to or greater than the threshold. Setting the `alphaThreshold` to 0 makes all pixels opaque to hit detection regardless of the contents of the alpha channel.

See also

`useAlpha`

ambient

Usage

```
member(whichCastmember).shader(whichShader).ambient
member(whichCastmember).model(whichModel).shader.ambient
member(whichCastmember).model(whichModel).shaderList{[index]}.ambient
```

Description

3D `#standard` shader property; indicates how much of each color component of the ambient light in the cast member is reflected by the shader.

For example, if the color of the ambient light is `rgb(255, 255, 255)` and the value of the `ambient` property of the shader is `rgb(255, 0, 0)`, the shader will reflect all of the red component of the light that the shader's colors can reflect. However, it will reflect none of the blue and green components of the light, regardless of the colors of the shader. In this case, if there are no other lights in the scene, the blue and green colors of the shader will reflect no light, and will appear black.

The default value of this property is `rgb(63,63,63)`.

Example

This statement sets the `ambient` property of the model named Chair to `rgb(255, 255, 0)`. Chair will fully reflect the red and green components of the ambient light in the scene and completely ignore its blue component.

```
-- Lingo syntax
member("Room").model("Chair").shader.ambient = rgb(255, 0, 0)

// JavaScript syntax
member("Room").getPropRef("model",1).shader.ambient = color(255,0,0);
```

See also

`ambientColor`, `newLight`, `type (light)`, `diffuse`, `specular (shader)`

ambientColor

Usage

```
member(whichCastmember).ambientColor
```

Description

3D cast member property; indicates the RGB color of the default ambient light of the cast member.

The default value for this property is rgb(0, 0, 0). This adds no light to the scene.

Example

This statement sets the `ambientColor` property of the cast member named Room to rgb(255, 0, 0). The default ambient light of the cast member will be red. This property can also be set in the Property inspector.

```
-- Lingo syntax
member("Room").ambientColor = rgb(255, 0, 0)

// JavaScript syntax
member("Room").ambientColor = color(255,0,0);
```

See also

`directionalColor`, `directionalPreset`, `ambient`

ancestor

Usage

```
property {optionalProperties} ancestor
```

Description

Object property; allows child objects and behaviors to use handlers that are not contained within the parent script or behavior.

The `ancestor` property is typically used with two or more parent scripts. You can use this property when you want child objects and behaviors to share certain behaviors that are inherited from an ancestor, while differing in other behaviors that are inherited from the parents.

For child objects, the `ancestor` property is usually assigned in the `on new` handler within the parent script. Sending a message to a child object that does not have a defined handler forwards that message to the script defined by the `ancestor` property.

If a behavior has an ancestor, the ancestor receives mouse events such as `mouseDown` and `mouseWithin`.

The `ancestor` property lets you change behaviors and properties for a large group of objects with a single command.

The ancestor script can contain independent property variables that can be obtained by child objects. To refer to property variables within the ancestor script, you must use this syntax:

`me.`*`propertyVariable`* `=` *`value`*

For example, this statement changes the property variable `legCount` within an ancestor script to 4:

```
me.legCount = 4
```

Use the syntax `the` *`variableName`* `of` *`scriptName`* to access property variables that are not contained within the current object. This statement allows the variable `myLegCount` within the child object to access the property variable `legCount` within the ancestor script:

```
set myLegCount to the legCount of me
```

Example

Each of the following scripts is a cast member. The ancestor script Animal and the parent scripts Dog and Man interact with one another to define objects.

The first script, Dog, sets the property variable `breed` to `Mutt`, sets `the ancestor of Dog` to the Animal script, and sets the `legCount` variable that is stored in the ancestor script to 4:

```
property breed, ancestor

on new me
    set breed = "Mutt"
    set the ancestor of me to new(script "Animal")
    set the legCount of me to 4
    return me
end
```

The second script, Man, sets the property variable `race` to Caucasian, sets the `ancestor of Man` to the Animal script, and sets the `legCount` variable that is stored in the ancestor script to 2:

```
property race, ancestor
on new me
    set race to "Caucasian"
    set the ancestor of me to new(script "Animal")
    set the legCount of me to 2
    return me
end
```

See also

new(), menu, property

angle (3D)

Usage

```
member(whichCastmember).modelResource(whichModelResource).emitter.angle
```

Description

3D emitter property; describes the area into which the particles of a particle system are emitted. A particle system is a model resource whose type is `#particle`.

The primary direction of particle emission is the vector set by the emitter's `direction` property. However, the direction of emission of a given particle will deviate from that vector by a random angle between 0 and the value of the emitter's `angle` property.

The effective range of this property is 0.0 to 180.0. The default value is 180.0.

Example

This statement sets the angle of emission of the model resource named mrFount to 1, which causes the emitted particles to form a thin line.

```
member("fountain").modelResource("mrFount").emitter.angle = 1
```

See also

emitter, direction

angle (DVD)

Usage

```
-- Lingo syntax
dvdObjRef.angle

// JavaScript syntax
dvdObjRef.angle;
```

Description

DVD property; returns the number of the current camera angle. Read/write.

The returned number is an integer.

Example

This statement returns the number of the current camera angles:

```
-- Lingo syntax
put(member(1).angle)-- 1

// JavaScript syntax
put(member(1).angle);// 1
```

See also

DVD

angleCount

Usage

```
-- Lingo syntax
dvdObjRef.angleCount

// JavaScript syntax
dvdObjRef.angleCount;
```

Description

DVD property; returns the number of available camera angles in the current title. Read-only.

The returned value is an integer that can range from 1 to 9.

Example

This statement returns the number of available camera angles:

```
-- Lingo syntax
put(member(12).angleCount)-- 2

// JavaScript syntax
put(member(12).angleCount);// 2
```

See also

DVD

animationEnabled

Usage

```
member(whichCastmember).animationEnabled
```

Description

3D cast member property; indicates whether motions will be executed (TRUE) or ignored (FALSE). This property can also be set in the Property inspector.

The default value for this property is TRUE.

Example

This statement disables animation for the cast member named Scene.

```
member("Scene").animationEnabled = FALSE
```

antiAlias

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.antiAlias
```

```
// JavaScript syntax
memberOrSpriteObjRef.antiAlias;
```

Description

Cast member property; controls whether a Vector shape, or Flash® cast member is rendered using anti-aliasing to produce high-quality rendering, but possibly slower playback of the movie. The antiAlias property is TRUE by default.

For vector shapes, TRUE is the equivalent of the #high quality setting for a Flash asset, and FALSE is the equivalent of #low.

The antiAlias property may also be used as a sprite property only for Vector shape sprites.

This property can be tested and set.

Note: *The antiAlias property is not supported for text members in Director 11. To antialias text members, use the antiAliasType property.*

Example

This behavior checks the color depth of the computer on which the movie is playing. If the color depth is set to 8 bits or less (256 colors), the script sets the antiAlias property of the sprite to FALSE.

```
--Lingo syntax
property spriteNum

on beginsprite me
    if _system.colorDepth <= 8 then
        sprite(spriteNum).antiAlias = FALSE
    end if
end

// JavaScript syntax
```

```
function beginsprite() {
    var cd = _system.colorDepth;
    if (cd <= 8 ) {
        sprite(this.spriteNum).antiAlias = 0;
    }
}
```

See also

antiAliasThreshold, quality

antiAliasingEnabled

Usage

```
sprite(whichSprite).antiAliasingEnabled
```

Description

3D sprite property; indicates whether the 3D world in the sprite *whichSprite* is anti-aliased. It can be tested and set. The default value is FALSE, indicating that anti-aliasing is off. If the antiAliasingEnabled property is set to TRUE and the 3D renderer changes to a renderer that does not support anti-aliasing, the property is set to FALSE. The value of this property is not saved when the movie is saved.

Anti-aliased sprites use more processor power and memory than sprites that are not anti-aliased. Temporarily turning off anti-aliasing can improve the performance of animations and user interaction.

Example

This Lingo checks whether the currently running 3D renderer for sprite 2 supports anti-aliasing with the antiAliasingSupported property. If anti-aliasing is supported, the second statement turns on anti-aliasing for the sprite with the antiAliasingEnabled property.

```
if sprite(2).antiAliasingSupported = TRUE then
    sprite(2).antiAliasingEnabled = TRUE
end if
```

See also

antiAliasingSupported, renderer, rendererDeviceList

antiAliasingSupported

Usage

```
sprite(whichSprite).antiAliasingSupported
```

Description

3D sprite property; indicates whether anti-aliasing is supported by the current 3D renderer. This property can be tested but not set. This property returns either TRUE or FALSE.

Example

This Lingo checks whether the currently running 3D renderer for sprite 3 supports anti-aliasing. If anti-aliasing is supported, the second statement turns on anti-aliasing for the sprite with the `antiAliasingEnabled` property.

```
if sprite(3).antiAliasingSupported = TRUE then
    sprite(3).antiAliasingEnabled = TRUE
end if
```

See also

antiAliasingEnabled, renderer, rendererDeviceList

antiAliasThreshold

Usage

```
-- Lingo syntax
memberObjRef.antiAliasThreshold

// JavaScript syntax
memberObjRef.antiAliasThreshold;
```

Description

Text cast member property; this setting controls the point size at which automatic anti-aliasing takes place in a text cast member. This has an effect only when the `antiAlias` property of the text cast member is set to `TRUE`.

The setting itself is an integer indicating the font point size at which the anti-alias takes place.

This property defaults to 14 points.

See also

antiAliasType

antiAliasType

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.antiAliasType

// JavaScript syntax
memberOrSpriteObjRef.antiAliasType;
```

Description

Cast member property; controls whether a text cast member is rendered using anti-aliasing to produce high-quality rendering. The `antiAliasType` property is `#autoAlias` by default.

If you are upgrading from a previous version of Director, the auto option is mapped to Grayscale Larger Than with the `antiAliasThreshold` set to zero. The Larger Than option is mapped to Grayscale Larger Than with the corresponding threshold.

Use the following symbols to set the corresponding antialiasing option:

```
Auto – #AutoAlias (Uses the font file information for anti-aliasing.)
```

Grayscale All - #GrayScaleAllAlias (Enables grayscale anti-aliasing for all text members.)

Subpixel All - #SubpixelAllAlias (Enables sub-pixel anti-aliasing for all text members.)

Grayscale Larger Than - #GrayscaleLargerThanAlias (Enables grayscale anti-aliasing for font sizes greater than the specified threshold.)

None - #NoneAlias (turns off anti-aliasing for the current cast member.)

This property can be tested and set.

Example

This behavior checks the color depth of the computer on which the movie is playing. If the color depth is set to 8 bits or less (256 colors), the script sets the `antiAlias` property of the sprite to `FALSE`.

```
--Lingo syntax
property spriteNum

on beginsprite me
    if _system.colorDepth <= 8 then
        sprite(spriteNum).antiAliasType = #noneAlias
    end if
end

// JavaScript syntax
function beginsprite() {
    var cd = _system.colorDepth;
    if (cd <= 8 ) {
        sprite(this.spriteNum).antiAliasType = symbol("noneAlias");
    }
}
```

See also
antiAliasThreshold

appearanceOptions

Usage

```
-- Lingo syntax
windowObjRef.appearanceOptions

// JavaScript syntax
windowObjRef.appearanceOptions;
```

Description

Window property; specifies a list of properties that stores the appearance options of a window. Read/write.

The property list contains the following properties.

Property	Description
#mask	Specifies the 1-bit cast member to use as a mask for the window.
#border	Specifies the type of border for the window. This property can be one of three values: • #none. Specifies no border around the window. • #line. Specifies a 1-pixel black border around the window. The #none and #line properties are only effective if the titlebarOptions.visible property is set to FALSE.
#metal	(Mac only) Specifies whether the window should have a metal look (TRUE). If FALSE, the window will have an ice look.
#dragRegionMask	Specifies the 1-bit cast member to use as a mask for a region of the window.
#shadow	(Mac only) Specifies whether the window should have a shadow. Mac windows typically have a shadow.
#liveresize	(Mac only) Specifies whether the window should have live resizing. If TRUE, live resizing is enabled. If FALSE, live resizing is disabled.

These properties can also be accessed by using the Movie object's displayTemplate property.

Example

This statement displays in the Message window all current appearance options for the window named Control Panel:

```
-- Lingo syntax
put(window("Control Panel").appearanceOptions)

// JavaScript syntax
put(window("Control Panel").appearanceOptions);
```

This statement sets the border property to display a 1-pixel border around the window named Control Panel:

```
-- Lingo syntax
window("Control Panel").appearanceOptions.border = #line

// JavaScript syntax
window("Control Panel").appearanceOptions.border = symbol("line");
```

See also

displayTemplate, titlebarOptions, visible, Window

applicationName

Usage

```
-- Lingo syntax
_player.applicationName

// JavaScript syntax
_player.applicationName;
```

Description

Player property; specifies the name of the running copy of the Director application during authoring, or the name of a projector file during runtime. Read-only.

The property value is a string.

Shockwave Player does not support this property.

Example

This statement displays the name of the Director application, Director.exe.

```
-- Lingo syntax
put(_player.applicationName)

// JavaScript syntax
put(_player.applicationName);
```

See also

applicationPath, Player

applicationPath

Usage

```
-- Lingo syntax
_player.applicationPath

// JavaScript syntax
_player.applicationPath;
```

Description

Player property; determines the path or location of the folder containing the running copy of the Director application during authoring, or the folder containing the projector during runtime. Read-only.

The property value is a string.

If you use `applicationPath` followed by `&` and a path to a subfolder, enclose the entire expression in parentheses so that script parses the expression as one phrase.

Shockwave Player does not support this property.

Example

This statement displays the pathname for the folder that contains the Director application.

```
-- Lingo syntax
put(_player.applicationPath)

// JavaScript syntax
put(_player.applicationPath);
```

This statement opens the movie Sunset Boulevard in a window (on a Windows machine):

```
-- Lingo syntax
window(_player.applicationPath & "Film Noir\Sunset Boulevard").open()

// JavaScript syntax
window(_player.applicationPath + "Film Noir\Sunset Boulevard").open();
```

See also

applicationName, Player

aspectRatio

Usage

```
-- Lingo syntax
dvdObjRef.aspectRatio

// JavaScript syntax
dvdObjRef.aspectRatio;
```

Description

DVD property. Returns a property list that specifies the width and height of the DVD cast member. Read-only.

Both the width and height are returned as integers.

Example

This statement returns the aspectRatio of member 1:

```
-- Lingo syntax
trace(member(1).aspectRatio) -- [#width: 16, #height:9]

// JavaScript syntax
trace(member(1).aspectRatio); // ["width": 16, "height":9];
```

See also

DVD

attenuation

Usage

```
member(whichCastMember).light(whichLight).attenuation
```

Description

3D light property; indicates the constant, linear, and quadratic attenuation factors for spotlights and point lights.

The default value for this property is vector(1.0, 0.0, 0.0).

Example

This statement sets the `attenuation` property of the light named HouseLight to the vector (.5, 0, 0), darkening it slightly.

```
-- Lingo syntax
member("3d world").light("HouseLight").attenuation = vector(.5, 0, 0)

// JavaScript syntax
member("3d world").getProp("light",1).attenuation = vector(.5, 0, 0);
```

See also

color (light)

attributeName

Usage

```
XMLnode.attributeName[ attributeNumber ]
```

Description

XML property; returns the name of the specified child node of a parsed XML document.

Example

Beginning with the following XML:

```
<?xml version="1.0"?>
    <e1>
        <tagName attr1="val1" attr2="val2"/>
        <e2> element 2</e2>
        <e3>element 3</e3>
        here is some text
    </e1>
```

This Lingo returns the name of the first attribute of the tag called `tagName`:

```
put gParserObject.child[1].child[1].attributeName[1]
-- "attr1"
```

See also

attributeValue

attributeValue

Usage

```
XMLnode.attributeValue[ attributeNameOrNumber ]
```

Description

XML property; returns the value of the specified child node of a parsed XML document.

Example

Beginning with the following XML:

```
<?xml version="1.0"?>
    <e1>
        <tagName attr1="val1" attr2="val2">
        <e2> element 2</e2>
        <e3>element 3</e3>
        here is some text
    </e1>
```

This Lingo returns the value of the first attribute of the tag called `tagName`:

```
put gParserObject.child[1].child[1].attributeValue[1]
-- "val1"
```

See also

attributeName

audio (DVD)

Usage

```
-- Lingo syntax
dvdObjRef.audio

// JavaScript syntax
dvdObjRef.audio;
```

Description

DVD property. Determines whether audio is enabled (TRUE, default) or not (FALSE). Read/write.

Example

This statement disables audio:

```
-- Lingo syntax
member(14).audio = 0

// JavaScript syntax
member(14).audio = 0;
```

See also

DVD

audio (RealMedia)

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.audio

// JavaScript syntax
memberOrSpriteObjRef.audio;
```

Description

RealMedia sprite or cast member property; allows you to play (TRUE) or mute (FALSE) the audio in the RealMedia stream. The default setting for this property is TRUE (1). Integer values other than 1 or 0 are treated as TRUE (1). Setting this property has no effect if the realPlayerNativeAudio() method is set to TRUE.

If the audio property is set to FALSE when a RealMedia cast member starts playing, a sound channel is still allocated, which allows you to toggle the sound on and off during playback.

There may be some latency involved in setting this property, which means there may be a slight delay before the sound toggles on or off.

Example

The following examples show that the audio properties for sprite 2 and the cast member Real is set to TRUE, which means that the audio portion of the RealMedia stream will be played.

```
-- Lingo syntax
put(sprite(2).audio) -- 1
put(member("Real").audio) -- 1

// JavaScript syntax
```

```
put(sprite(2).audio); // 1
put(member("Real").audio); // 1
```

The following Lingo sets the `audio` property for sprite 2 and the cast member Real to `FALSE`, which means that the audio portion of the RealMedia stream will not be played when the movie is played.

```
-- Lingo syntax
sprite(2).audio = FALSE
member("Real").audio = FALSE

// JavaScript syntax
sprite(2).audio = 0;
member("Real").audio = 0;
```

See also

soundChannel (RealMedia), video (RealMedia, Windows Media), sound (Player)

audio (Windows Media)

Usage

```
-- Lingo syntax
windowsMediaObjRef.audio

// JavaScript syntax
windowsMediaObjRef.audio;
```

Description

Windows Media property. Specifies whether audio is enabled (`TRUE`, default) or not (`FALSE`) during playback. Read/write.

Example

This statement displays in the Message window whether audio is enabled for cast member 5:

```
-- Lingo syntax
trace(member(5).audio)

// JavaScript syntax
trace(member(5).audio);
```

See also

Windows Media

audioChannelCount

Usage

```
-- Lingo syntax
dvdObjRef.audioChannelCount

// JavaScript syntax
dvdObjRef.audioChannelCount;
```

Description

DVD property; returns the number of audio channels. Read-only.

Example

This statement returns the number of audio channels:

```
-- Lingo syntax
member(1).audioChannelCount

// JavaScript syntax
member(1).audioChannelCount;
```

See also

DVD

audioExtension

Usage

```
-- Lingo syntax
dvdObjRef.audioExtension

// JavaScript syntax
dvdObjRef.audioExtension;
```

Description

DVD property. Returns a symbol that indicates the audio extensions, if any, of an audio stream. Read-only.

Possible returned values are as follows:

Symbol	Description
#caption	The audio stream contains captions.
#lowvision	The audio stream contains content for people with low vision.
#directorcomments1	The audio stream contains "director comments 1."
#directorcomments2	The audio stream contains "director comments 2."
#none	The DVD does not specify an audio extension for this audio stream, or it could not be determined.

See also

DVD

audioFormat

Usage

```
-- Lingo syntax
dvdObjRef.audioFormat

// JavaScript syntax
dvdObjRef.audioFormat;
```

Description

DVD property. Returns a symbol that indicates the format (encoding mode) of an audio stream. Read-only.

Possible returned values are as follows:

Symbol	Description
#AC3	The audio format is Dolby AC-3.
#MPEG1	The audio format is MPEG-1.
#MPEG1DRC	The audio format is MPEG-1 with dynamic range control.
#MPEG2	The audio format is MPEG-2.
#MPEG2DRC	The audio format is MPEG-2 with dynamic range control
#LPCM	The audio format is Linear Pulse Code Modulated (LPCM).
#DTS	The audio format is Digital Theater Systems (DTS).
#SDDS	The audio format is Sony Dynamic Digital Sound (SDDS).

See also

DVD

audioSampleRate

Usage

```
-- Lingo syntax
dvdObjRef.audioSampleRate

// JavaScript syntax
dvdObjRef.audioSampleRate;
```

Description

DVD property; returns the frequency, in hertz, of an audio stream. Read-only.

See also

DVD

audioStream

Usage

```
-- Lingo syntax
dvdObjRef.audioStream

// JavaScript syntax
dvdObjRef.audioStream;
```

Description

DVD property. Returns the currently active audio stream. Read/write.

Valid values range from 1 to 8.

See also

DVD

audioStreamCount

Usage

```
-- Lingo syntax
dvdObjRef.audioStreamCount

// JavaScript syntax
dvdObjRef.audioStreamCount;
```

Description

DVD property; returns the number of available audio streams in the current title. Read-only.

The number of available audio streams ranges from 1 to 8.

See also

DVD

auto

Usage

```
member(whichCastmember).model(whichModel).lod.auto
```

Description

3D `lod` modifier property; allows the modifier to manage the reduction of detail in the model as the distance between the model and the camera changes.

The setting of the modifier's `bias` property determines how aggressively the modifier removes detail from the model when the `auto` property is set to `TRUE`.

The modifier updates its `level` property as it adjusts the model's level of detail. Setting the `level` property has no effect unless the `auto` property is set to `FALSE`.

The `#lod` modifier can only be added to models created outside of Director in 3D modeling programs. The value of the `type` property of the model resources used by these models is `#fromFile`. The modifier cannot be added to primitives created within Director.

Example

This statement sets the `auto` property of the lod modifier of the model named Spaceship to `TRUE`. The modifier will automatically set the model's level of detail.

```
-- Lingo syntax
member("3D World").model("Spaceship").lod.auto = TRUE

// Java Script
member("3D World").getPropRef("model", 1).getPropRef("lod", 1).auto = true;
```

See also

lod (modifier), bias, level

autoblend

Usage

```
member(whichCastmember).model(whichModel).keyframePlayer.autoblend
member(whichCastmember).model(whichModel).bonesPlayer.autoblend
```

Description

3D keyframePlayer and bonesPlayer modifier property; indicates whether the modifier creates a linear transition to the currently playing motion from the motion that preceded it (TRUE) or not (FALSE). If autoBlend is TRUE, the length of the transition is set by the blendTime property of the modifier. If autoBlend is FALSE, the transition is controlled by the blendFactor property of the modifier and blendTime is ignored.

Motion blending is completely disabled when blendTime is set to 0 and autoBlend is set to TRUE.

The default value of this property is TRUE.

Example

This statement turns autoblend off for the model named Alien3. The model's blendFactor setting will be used for blending successive motions in the playlist.

```
-- Lingo syntax
member("newaliens").model("Alien3").addModifier(#keyframeplayer)
member("newaliens").model("Alien3").keyframePlayer.autoblend = FALSE

// JavaScript syntax
member("newaliens").getPropRef("model",1).addModifier(symbol("keyframeplayer"));
member("newaliens").getPropRef("model",1).getPropRef("keyframeplayer",1).autoBlend =
false;
```

See also

blendFactor, blendTime

autoCameraPosition

Usage

```
member(whichTextCastmember).autoCameraPosition
```

Description

3D camera property; indicates whether the camera of the 3D text cast member is automatically positioned to show all of the text (TRUE) or not (FALSE). This is useful when changing the text, font, fontsize, and other properties of the cast member.

This property is not valid with other types of 3D cast members.

Example

This statement sets the `autoCameraPosition` property of the cast member named Headline to `FALSE`. When the cast member is displayed in 3D mode, the camera will not be positioned automatically.

```
-- Lingo syntax
member("Headline").autoCameraPosition = FALSE

// JavaScript syntax
member("Headline").autoCameraPosition = false;
```

See also

displayMode

autoMask

Usage

```
member(whichCursorCastMember).autoMask
the autoMask of member whichCastMember
```

Description

Cast member property; specifies whether the white pixels in the animated color cursor cast member *whichCursorCastMember* are transparent, allowing the background to show through (`TRUE`, default), or opaque (`FALSE`).

Example

In this script, when the custom animated cursor stored in cast member 5 enters the sprite, the automask is turned on so that the background of the sprite will show through the white pixels. When the cursor leaves the sprite, the automask is turned off.

```
-- Lingo syntax
on mouseEnter
    member 5.autoMask = TRUE
end

on mouseLeave
    member 5.autoMask = FALSE
end
```

Using traditional Lingo syntax, the script is written as:

```
on mouseEnter
    set the autoMask of member 5 = TRUE
end

on mouseLeave
    set the autoMask of member 5 = FALSE
end
```

autoTab

Usage

```
-- Lingo syntax
```

```
memberObjRef.autoTab

// JavaScript syntax
memberObjRef.autoTab;
```

Description

Cast member property; determines the effect that pressing the Tab key has on the editable field or text cast member specified by *whichCastMember*. The property can be made active (TRUE) or inactive (FALSE). Tabbing order depends on sprite number order, not position on the Stage.

Example

This statement causes the cast member Comments to automatically advance the insertion point to the next editable field or text sprite after the user presses Tab.

```
--Lingo syntax
member ("Comments").autotab = TRUE

// JavaScript syntax
member ("Comments").autotab = true;
```

axisAngle

Usage

```
member(whichCastmember).model(whichModel).transform.axisAngle
member(whichCastmember).camera(whichCamera).transform.axisAngle
member(whichCastmember).light(whichLight).transform.axisAngle
member(whichCastmember).group(whichGroup).transform.axisAngle
transformReference.axisAngle
```

Description

3D transform property; describes the transform's rotation as an axis/angle pair.

The `axisAngle` property is a linear list containing a vector (the axis) and a float (the angle). The vector is the axis around which the transform is rotated. The float is the amount, in degrees, of rotation.

The default value of this property is `[vector( 1.0000, 0.0000, 0.0000 ), 0.0000]`.

Example

This statement shows the rotation of the model named Mailbox as an axisAngle. The model is rotated 145.5 degrees counterclockwise about the y axis.

```
-- Lingo syntax
put member("Yard").model("Mailbox").transform.axisAngle
-- [vector( 0.0000, 1.0000, 0.0000 ), -145.5000]

// JavaScript syntax
put(member("Yard").getProp("model",1).transform.axisAngle);
// <[vector( 0.0000, 1.0000, 0.0000 ), -145.5000]>
```

See also

rotation (transform)

back

Usage

```
member(whichCastmember).modelResource(whichModelResource).back
```

Description

3D `#box` model resource property; indicates whether the side of the box intersected by its +Z axis is sealed (`TRUE`) or open (`FALSE`).

The default value for this property is `TRUE`.

Example

This statement sets the `back` property of the model resource named Crate to `FALSE`, meaning the back of this box will be open.

```
-- Lingo syntax
nmr = member("3D World").newModelResource("Crate", symbol("box"))
member("3D World").modelResource("Crate").back = FALSE

// JavaScript syntax
nmr = member("3D World").newModelResource("Crate", symbol("box"));
member("3D World").getProp("modelresource",10).back = false;
```

See also

`bottom (3D)`, `front`, `top (3D)`, `left (3D)`, `right (3D)`

backColor

Usage

```
-- Lingo syntax
spriteObjRef.backColor

// JavaScript syntax
spriteObjRef.backColor;
```

Description

Sprite property; sets the background color of a specified sprite according to the color value assigned. Read/write.

Setting `backColor` of a sprite is the same as choosing the background color from the Tool palette when the sprite is selected on the Stage. For the value that a script sets to last beyond the current sprite, the sprite must be a scripted sprite. The background color applies to all bitmap cast members, in addition to field, button, check box, and radio cast members.

The `backColor` value ranges from 0 to 255 for 8-bit color and from 0 to 15 for 4-bit color. The numbers correspond to the index number of the background color in the current palette. (A color's index number appears in the color palette's lower left corner when you click the color.)

If this property is set on bitmap cast members that are deeper than 1-bit, the `backColor` may not be seen if the background of the bitmap is not visible.

If the blend of a sprite is less than 100 but greater than 0, the `backColor` will mix with the transparent colors.

Note: *It is recommended that the newer `bgColor` property be used instead of the `backColor` property.*

Example

The following statement sets the variable `oldColor` to the background color of sprite 5:

```
-- Lingo syntax
oldColor = sprite(5).backColor

// JavaScript syntax
var oldColor = sprite(5).backColor;
```

The following statement randomly changes the background color of a random sprite between sprites 11 and 13 to color number 36:

```
-- Lingo syntax
sprite(10 + random(3)).backColor = 36

// JavaScript syntax
sprite(10 + random(3)).backColor = 36;
```

See also

Sprite

backdrop

Usage

```
sprite(whichSprite).camera{(index)}.backdrop[index].loc
member(whichCastmember).camera(whichCamera).backdrop[index].loc
sprite(whichSprite).camera{(index)}.backdrop[index].source
member(whichCastmember).camera(whichCamera).backdrop[index].source
sprite(whichSprite).camera{(index)}.backdrop[index].scale
member(whichCastmember).camera(whichCamera).backdrop[index].scale
sprite(whichSprite).camera{(index)}.backdrop[index].rotation
member(whichCastmember).camera(whichCamera).backdrop[index].rotation
sprite(whichSprite).camera{(index)}.backdrop[index].regPoint
member(whichCastmember).camera(whichCamera).backdrop[index].regPoint
sprite(whichSprite).camera{(index)}.backdrop[index].blend
member(whichCastmember).camera(whichCamera).backdrop[index].blend
sprite(whichSprite).camera{(index)}.backdrop.count
member(whichCastmember).camera(whichCamera).backdrop.count
```

Description

3D camera property; a 2D image that is rendered on the camera's projection plane. All models in the camera's view appear in front of the backdrop.

Backdrops have the following properties:

Note: *These properties can also be used to get, set, and manipulate overlays. For detailed information, see the individual property entries.*

loc (backdrop and overlay) indicates the 2D location of the backdrop, as measured from the upper left corner of the sprite.

source indicates the texture used by the backdrop.

scale (backdrop and overlay) is the number by which the height and width of the texture are multiplied to determine the dimensions of the backdrop.

rotation (backdrop and overlay) is the amount by which the backdrop is rotated about its regPoint.

regPoint (3D) indicates the registration point of the backdrop.

blend (3D) indicates the opacity of the backdrop.

count (3D) indicates the number of items in the camera's list of backdrops.

Use the following commands to create and remove backdrops:

addBackdrop creates a backdrop from a texture and adds it to the end of the camera's list of backdrops.

insertBackdrop creates a backdrop from a texture and adds it to the camera's list of backdrops at a specific index position.

removeBackdrop deletes the backdrop.

See also

overlay

backgroundColor

Usage

```
-- Lingo syntax
memberObjRef.backgroundColor

// JavaScript syntax
memberObjRef.backgroundColor;
```

Description

Vector shape cast member property; sets the background color of the specified cast member or sprite to the RGB color value assigned.

This property can be both tested and set.

Example

```
-- Lingo syntax
member("Archie").backgroundColor= color(255,255,255)

// JavaScript syntax
member("Archie").backgroundColor= color(255,255,255);
```

See also

bgColor (Window)

beepOn

Usage

```
-- Lingo syntax
_movie.beepOn

// JavaScript syntax
_movie.beepOn;
```

Description

Movie property; determines whether the computer automatically beeps when the user clicks on anything except an active sprite (TRUE), or not (FALSE, default). Read/write.

Scripts that set beepOn should be placed in frame or movie scripts.

Example

This statement sets beepOn to TRUE:

```
-- Lingo syntax
_movie.beepOn = TRUE

// JavaScript syntax
_movie.beepOn = true;
```

This statement sets beepOn to the opposite of its current setting:

```
-- Lingo syntax
_movie.beepOn = not(_movie.beepOn)

// JavaScript syntax
_movie.beepOn = !(_movie.beepOn);
```

See also

Movie

bevelDepth

Usage

```
member(whichTextCastmember).bevelDepth
member(which3DCastmember).modelResource(whichModelResource).bevelDepth
```

Description

3D text property; indicates the degree of beveling on the 3D text.

For text cast members, this property has no effect unless the member's displayMode property is set to #mode3D and its bevelType property is set to #miter or #round.

For extruded text in a 3D cast member, this property has no effect unless the model resource's bevelType property is set to #miter or #round.

The range of this property is 0.0 to 10.0, and the default setting is 10.0.

Example

In this example, the cast member named Logo is a text cast member. This statement sets the bevelDepth of logo to 5.5. When logo is displayed in 3D mode, if its bevelType property is set to #miter or #round, the edges of its letters will exhibit dramatic beveling.

```
-- Lingo syntax
member("Logo").bevelDepth = 5.5

// JavaScript syntax
member("Logo").bevelDepth = 5.5;
```

In this example, the model resource of the model named Slogan is extruded text. This statement sets the bevelDepth of Slogan's model resource to 5. If the `bevelType` property of Slogan is set to `#miter` or `#round`, the edges of its letters will exhibit dramatic beveling.

```
-- Lingo syntax
member("scene").model("Slogan").resource.bevelDepth = 5
```

See also

`bevelType`, `extrude3D`, `displayMode`

bevelType

Usage

```
member(whichTextCastmember).bevelType
member(which3DCastmember).modelResource(whichModelResource).bevelType
```

Description

3D text property; indicates the style of beveling applied to the 3D text.

For text cast members, this is a member property. For extruded text in a 3D cast member, this is a model resource property.

The `bevelType` property has the following possible values:

- `#none`
- `#miter` (the default)
- `#round`

Example

In this example, the cast member named Logo is a text cast member. This statement sets the `bevelType` of Logo to `#round`.

```
member("logo").beveltype = #round
```

In this example, the model resource of the model named Slogan is extruded text. This statement sets the `bevelType` of Slogan's model resource to `#miter`.

```
member("scene").model("Slogan").resource.bevelType = #miter
```

See also

`bevelDepth`, `extrude3D`, `displayMode`

bgColor (Window)

Usage

```
-- Lingo syntax
windowObjRef.bgColor

// JavaScript syntax
windowObjRef.bgColor;
```

Description

Window property; determines the background color of a window. Read/write.

Setting the `bgColor` property is equivalent to setting the color in the Movie Properties dialog box.

Example

This example sets the color of the window named Animals to an RGB value.

```
-- Lingo syntax
window("Animals").bgColor = color(255, 153, 0)

// JavaScript syntax
window("Animals").bgColor = color(255, 153, 0);
```

See also

Window

bgColor (Sprite, 3D Member)

Usage

```
sprite(whichSpriteNumber).bgColor
the bgColor of sprite whichSpriteNumber
the bgColor of the stage
(the stage).bgColor
member(which3dMember).bgcolor
```

Description

Sprite property, system property, and 3D cast member property; determines the background color of the sprite specified by *whichSprite*, the color of the Stage, or the background color of the 3D cast member. Setting the `bgColor` sprite property is equivalent to choosing the background color from the Tools window when the sprite is selected on the Stage. Setting the `bgColor` property for the Stage is equivalent to setting the color in the Movie Properties dialog box.

The sprite property has the equivalent functionality of the `backColor` sprite property, but the color value returned is a color object of whatever type has been set for that sprite.

This property can be tested and set.

Example

This example sets the color of the Stage to an RGB value

Dot syntax:

```
(the stage).bgColor = rgb(255, 153, 0)
```

Verbose Lingo syntax:

```
set the bgColor of the stage = rgb(255, 153, 0)
```

See also

color(), backColor, backgroundColor

bias

Usage

```
member(whichCastmember).model(whichModel).lod.bias
```

Description

3D `lod` modifier property; indicates how aggressively the modifier removes detail from the model when its `auto` property is set to `TRUE`. This property has no effect when the modifier's `auto` property is set to `FALSE`.

The range for this property is from 0.0 (removes all polygons) to +100.0 (removes no polygons). The default setting is 100.0.

The `#lod` modifier can only be added to models created outside of Director in 3D modeling programs. The value of the `type` property of the model resources used by these models is `#fromFile`. The modifier cannot be added to primitives created within Director.

Example

This statement sets the `bias` property of the lod modifier of the model named Spaceship to 10. If the lod modifier's `auto` property is set to `TRUE`, the modifier will very aggressively lower the level of detail of Spaceship as it moves away from the camera.

```
-- Lingo syntax
member("3D World").model("Spaceship").lod.bias = 10

// Java Script
member("3D World").getPropRef("model", 1).getPropRef("lod", 1).bias = 10;
```

See also

`lod (modifier)`, `auto`, `level`

bitmapSizes

Usage

```
-- Lingo syntax
memberObjRef.bitmapSizes

// JavaScript syntax
memberObjRef.bitmapSizes;
```

Description

Font cast member property; returns a list of the bitmap point sizes that were included when the font cast member was created.

Example

This statement displays the bitmap point sizes that were included when cast member 11 was created:

```
-- Lingo syntax
put(member(11).bitmapSizes)

// JavaScript syntax
put(member(11).bitmapSizes);
```

See also

recordFont, characterSet, originalFont

bitRate

Usage

```
-- Lingo syntax
memberObjRef.bitRate

// JavaScript syntax
memberObjRef.bitRate;
```

Description

Shockwave Audio (SWA) cast member property; returns the bit rate, in kilobits per second (Kbps), of the specified SWA cast member that has been preloaded from the server.

The `bitRate` member property returns 0 until streaming begins.

Example

This behavior outputs the bit rate of an SWA cast member when the sprite is first encountered.

```
-- Lingo syntax
property spriteNum

on beginSprite (me)
    memName = sprite(spriteNum).member.name
    put("The bitRate of member"&&memName&&"is"&&member(memName).bitRate)
end

// JavaScript syntax
function beginSprite() {
    var memName = sprite(spriteNum).member.name;
    put("The bitRate of member " + memName +" is " + member(memName).bitRate);
}
```

bitsPerSample

Usage

```
-- Lingo syntax
memberObjRef.bitsPerSample

// JavaScript syntax
memberObjRef.bitsPerSample;
```

Description

Shockwave Audio (SWA) cast member property; indicates the bit depth of the original file that has been encoded for Shockwave Audio (SWA). This property is available only after the SWA sound begins playing or after the file has been preloaded using the `preLoadBuffer` command.

This property can be tested but not set.

Example

This statement assigns the original bit rate of the file used in SWA streaming cast member Paul Robeson to the field cast member How Deep.

```
-- Lingo syntax
member("How Deep").text = member("Paul Robeson").bitsPerSample

// JavaScript syntax
member("How Deep").text = member("Paul Robeson").bitsPerSample;
```

blend (3D)

Usage

```
sprite(whichSprite).camera{(index)}.backdrop[index].blend
member(whichCastmember).camera(whichCamera).backdrop[index].blend
sprite(whichSprite).camera{(index)}.overlay[index].blend
member(whichCastmember).camera(whichCamera).overlay[index].blend
member(whichCastmember).shader(whichShader).blend
member(whichCastmember).model(whichModel).shader.blend
member(whichCastmember).model(whichModel).shaderList{[index]}.blend
```

Description

3D backdrop, overlay, and `#standard` shader property; indicates the opacity of the backdrop, overlay, or shader.

Setting the `blend` property of a shader will have no effect unless the shader's `transparent` property is set to `TRUE`.

The range of this property is 0 to 100, and the default value is 100.

Example

This statement sets the `blend` property of the shader for the model named Window to 80. If the `transparent` property of Window's shader is set to TRUE, the model will be slightly transparent.

```
-- Lingo syntax
member("House").model("Window").shader.blend = 80

// Java Script
member("House").getPropRef("model", 1).shaderList[1].blend = 80;
```

See also

bevelDepth, overlay, shadowPercentage, transparent

blend (Sprite)

Usage

```
-- Lingo syntax
spriteObjRef.blend

// JavaScript syntax
spriteObjRef.blend;
```

Description

Sprite property; returns or sets a sprite's blend value, from 0 to 100, corresponding to the blend values in the Sprite Properties dialog box. Read/write.

The possible colors depend on the colors available in the palette, regardless of the monitor's color depth.

For best results, use the blend ink with images that have a color depth greater than 8-bit.

Example

The following statement sets the blend value of sprite 3 to 40 percent.

```
-- Lingo syntax
sprite(3).blend = 40

// JavaScript syntax
sprite(3).blend = 40;
```

This statement displays the blend value of sprite 3 in the Message window:

```
-- Lingo syntax
put(sprite(3).blend)

// JavaScript syntax
put(sprite(3).blend);
```

See also

`blendLevel`, `Sprite`

blendConstant

Usage

```
member(whichCastmember).shader(whichShader).blendConstant
member(whichCastmember).model(whichModel).shader.blendConstant
member(whichCastmember).model(whichModel).shaderList{[index]}.blendConstant
```

Description

3D `#standard` shader property; indicates the blending ratio used for the first texture layer of the shader.

If the shader's `useDiffuseWithTexture` property is set to `TRUE`, the texture blends with the color set by the shader's `diffuse` property. If `useDiffuseWithTexture` is `FALSE`, white is used for blending.

Each of the other texture layers blends with the texture layer below it. Use the `blendConstantList` property to control blending in those texture layers.

The `blendConstant` property works only when the shader's `blendSource` property is set to `#constant`. For more information, see `blendSource` and `blendSourceList`.

The range of this property is 0 to 100; the default is 50.

Example

In this example, the shader list of the model named MysteryBox contains six shaders. This statement sets the `blendConstant` property of the second shader to 20. This property is affected by the settings of the `blendFunction`, `blendFunctionList`, `blendSource`, and `blendSourceList` properties.

```
member("Level2").model("MysteryBox").shaderList[2].blendConstant = 20
```

See also

blendConstantList, blendFunction, blendFunctionList, blendSource, blendSourceList, useDiffuseWithTexture, diffuse, diffuseColor

blendConstantList

Usage

```
member(whichCastmember).shader(whichShader).blendConstantList
member(whichCastmember).model(whichModel).shader.blendConstantList{[index]}
member(whichCastmember).model(whichModel).shaderList{[index]}.blendConstantList{[index]}
```

Description

3D `#standard` shader property; indicates the ratio used for blending a texture layer of the shader with the texture layer below it.

The shader's texture list and the blend constant list both have eight index positions. Each index position in the blend constant list controls blending for the texture at the corresponding index position in the texture list. You can set all index positions of the list to the same value at one time by not specifying the optional *index* parameter. Use the *index* parameter to set the list one index position at a time.

The `blendConstantList` property works only when the `blendSource` property of the corresponding texture layer is set to `#constant`.

The range of this property is 0 to 100; the default is 50.

Example

In this example, the shader list of the model named MysteryBox contains six shaders. This statement shows the `blendConstant` property of each of the textures used by the second shader. This property is affected by the settings of the `blendFunction`, `blendFunctionList`, `blendSource`, and `blendSourceList` properties.

```
-- Lingo syntax
put member("Level2").model("MysteryBox").shaderList[2].blendConstantList
-- [20.0000, 50.0000, 50.0000, 50.0000, 20.0000, 50.0000, 50.0000, 50.0000]

// JavaScript syntax
put(member("Level2").getPropRef("model",3).shaderList[1].blendConstantList);
// <[20.0000, 50.0000, 50.0000, 50.0000, 20.0000, 50.0000, 50.0000, 50.0000]>
put member("Level2").model("MysteryBox").shaderList[2].blendConstantList
-- [20.0000, 50.0000, 50.0000, 50.0000, 20.0000, 50.0000, 50.0000, 50.0000]
```

See also

blendConstant, blendFunction, blendFunctionList, blendSource, blendSourceList, useDiffuseWithTexture, diffuse, diffuseColor

blendFactor

Usage

```
member(whichCastmember).model(whichModel).keyframePlayer.blendFactor
member(whichCastmember).model(whichModel).bonesPlayer.blendFactor
```

Description

3D `keyframePlayer` and `bonesPlayer` modifier property; indicates the amount by which a motion is combined with the motion that preceded it.

The range of this property is 0 to 100, and the default value is 0.

`BlendFactor` is used only when the `autoblend` property of the modifier is set to `FALSE`. If the value of the `blendFactor` property is 100, the current motion will have none of the characteristics of the motion that preceded it. If the value of `blendFactor` is 0, the current motion will have all of the characteristics of the motion that preceded it and none of its own. If the value of `blendFactor` is 50, the current motion will be a synthesis equally composed of its own characteristics and those of the motion that preceded it. The value `blendFactor` can be varied over time to create transitions unlike the linear transition created when the modifier's `autoblend` property is set to `TRUE`.

Example

This statement sets the `blendFactor` property of model Alien3 to 50. If the modifier's `autoblend` property is `FALSE`, each motion in the playlist of the `keyframePlayer` for Alien3 will be an even mixture of itself and the motion that preceded it.

```
member("newaliens").model("Alien3").keyframePlayer.blendFactor = 50
```

See also

`autoblend`, `keyframePlayer (modifier)`

blendFunction

Usage

```
member(whichCastmember).shader(whichShader).blendFunction
member(whichCastmember).model(whichModel).shader.blendFunction
member(whichCastmember).model(whichModel).shaderList{[index]}.blendFunction
```

Description

3D `#standard` shader property; indicates the type of blending used by the first texture layer of the shader.

If the shader's `useDiffuseWithTexture` property is set to `TRUE`, the texture blends with the color set by the shader's `diffuse` property. If `useDiffuseWithTexture` is `FALSE`, white is used for blending.

Each of the other texture layers blends with the texture layer below it. Use the `blendFunctionList` property to control blending in those texture layers.

The `blendFunction` property can have the following values:

`#multiply` multiplies the RGB values of the texture layer by the color being used for blending (see above).

`#add` adds the RGB values of the texture layer to the color being used for blending, and then clamps to 255.

`#replace` prevents the texture from being blended with the color set by the shader's `diffuse` property.

`#blend` combines the colors of the texture layer with the color being used for blending in the ratio set by the `blendConstant` property.

The default value of this property is `#multiply`.

Example

In this example, the shader list of the model named MysteryBox contains six shaders. This statement sets the `blendFunction` property of the second shader to `#blend`. This enables the settings of the `blendSource`, `blendSourceList`, `blendConstant`, and `blendConstantList` properties.

```
member("Level2").model("MysteryBox").shaderList[2].blendFunction = #blend
```

See also

`blendConstant`, `blendConstantList`, `blendFunctionList`, `blendSource`, `blendSourceList`, `useDiffuseWithTexture`, `diffuse`, `diffuseColor`

blendFunctionList

Usage

```
member(whichCastmember).shader(whichShader).blendFunctionList{[index]}
member(whichCastmember).model(whichModel).shader.blendFunctionList{[index]}
member(whichCastmember).model(whichModel).shaderList{[index]}.blendFunctionList{[index]}
```

Description

3D `#standard` shader property; a linear list that indicates the manner in which each texture layer blends with the texture layer below it.

The shader's texture list and blend function list both have eight index positions. Each index position in the blend function list controls blending for the texture at the corresponding index position in the texture list. You can set all index positions of the list to the same value at one time by not specifying the optional *index* parameter. Use the *index* parameter to set the list one index position at a time.

Each index position of the blend function list can have one of the following values:

`#multiply` multiplies the RGB values of the texture layer by the RGB values of the texture layer below it.

`#add` adds the RGB values of the texture layer to the RGB values of the texture layer below it, and then clamps to 255.

`#replace` causes the texture to cover the texture layer below it. No blending occurs.

`#blend` causes blending to be controlled by the value of the `blendSource` property, which allows alpha blending.

The default value of this property is `#multiply`.

Example

In this example, the `shaderList` property of the model named MysteryBox contains six shaders. This statement shows that the value of the fourth index position of the `blendFunctionList` property of the second shader is set to `#blend`. Blending of the fourth texture layer of the second shader of the model will be controlled by the settings of the `blendSource`, `blendSourceList`, `blendConstant`, `blendConstantList`, `diffuse`, `diffuseColor`, and `useDiffuseWithTexture` properties.

```
put member("Level2").model("MysteryBox").shaderList[2].blendFunctionList[4]
-- #blend
```

See also

`blendConstant`, `blendConstantList`, `blendFunction`, `blendSource`, `blendSourceList`, `diffuse`, `diffuseColor`, `useDiffuseWithTexture`

blendLevel

Usage

```
sprite(whichSpriteNumber).blendLevel
the blendLevel of sprite whichSpriteNumber
```

Description

Sprite property; allows the current blending value of a sprite to be set or accessed. The possible range of values is from 0 to 255. This differs from the Sprite Inspector, which shows values in the range 0 to 100. The results are the same, the scales simply differ.

This property is the equivalent of the `blend` sprite property.

Example

```
sprite(3).blendlevel = 99
```

See also

`blend (Sprite)`

blendRange

Usage

```
member(whichCastmember).modelResource(whichModelResource).blendRange.start
modelResourceObjectReference.blendRange.end
member(whichCastmember).modelResource(whichModelResource).blendRange.start
modelResourceObjectReference.blendRange.end
```

Description

3D property; when used with a model resource whose type is `#particle`, allows you to get or set the start and end of the model resource's blend range.

The opacity of particles in the system is interpolated linearly between `blendRange.start` and `blendRange.end` over the lifetime of each particle.

This property's value must be greater than or equal to 0.0 and less than or equal to 100.0. The default value for this property is 100.0.

Example

This statement sets the blendRange properties of model resource ThermoSystem, which is of the type #particle.

The first line sets the start value to 100, and the second line sets the end value to 0. The effect of this statement is that the particles of ThermoSystem are fully opaque when they first appear, and then gradually fade to transparent during their lifetime.

```
member("Heater").modelResource("ThermoSystem").blendRange.start = 100.0
member("Heater").modelResource("ThermoSystem").blendRange.end = 0.0
```

blendSource

Usage

```
member(whichCastmember).shader(whichShader).blendSource
member(whichCastmember).model(whichModel).shader.blendSource
member(whichCastmember).model(whichModel).shaderList{[index]}.blendSource
```

Description

3D `#standard` shader property; indicates whether blending of the first texture layer in the shader's texture list is based on the texture's alpha information or a constant ratio.

If the shader's `useDiffuseWithTexture` property is set to `TRUE`, the texture blends with the color set by the shader's `diffuse` property. If `useDiffuseWithTexture` is `FALSE`, white is used for blending.

Each of the other texture layers blends with the texture layer below it. Use the `blendSourceList` property to control blending in those texture layers.

The `blendSource` property works only when the shader's `blendFunction` property is set to `#blend`.

The possible values of this property are as follows:

`#alpha` causes the alpha information in the texture to determine the blend ratio of each pixel of the texture with the color being used for blending (see above).

`#constant` causes the value of the shader's `blendConstant` property to be used as the blend ratio for all the pixels of the texture.

The default value of this property is `#constant`.

Example

In this example, the shader list of the model named MysteryBox contains six shaders. This statement sets the `blendSource` property of the first texture used by the second shader to `#constant`. This enables the settings of the `blendConstant` and `blendConstantList` properties.

```
-- Lingo syntax
member("Level2").model("MysteryBox").shaderList[2].blendSource = #constant

// JavaScript syntax
member("Level2").getPropRef("model",3).shaderList[2].blendSource = symbol("constant");
```

See also

`blendSourceList`, `blendFunction`, `blendFunctionList`, `blendConstant`, `blendConstantList`, `useDiffuseWithTexture`, `diffuse`, `diffuseColor`

blendSourceList

Usage

```
member(whichCastmember).shader(whichShader).blendSourceList[index]
member(whichCastmember).model(whichModel).shader.blendSourceList{[index]}
member(whichCastmember).model(whichModel).shaderList{[index]}.blendSourceList{[index]}
```

Description

3D `#standard` shader property; indicates whether blending of a texture layer with the texture layers below it is based on the texture's alpha information or a constant ratio.

The shader's texture list and the blend source list both have eight index positions. Each index position in the blend source list controls blending for the texture at the corresponding index position in the texture list. You can set all index positions of the list to the same value at one time by not specifying the optional *index* parameter. Use the *index* parameter to set the list one index position at a time.

The `blendSourceList` property only works when the `blendFunction` property of the corresponding texture layer is set to `#blend`. See `blendFunction` and `blendFunctionList` for more information.

The possible values of this property are as follows:

`#alpha` causes the alpha information in the texture to determine the blend ratio of each pixel of the texture layer with the layer below it.

`#constant` causes the value of the `blendConstant` property of the corresponding texture layer to be used as the blend ratio for all of the pixels of the texture layer. See `blendConstant` and `blendConstantList` for more information.

The default value of this property is `#constant`.

Example

In this example, the shader list of the model MysteryBox contains six shaders. Each shader has a texture list that contains up to eight textures. This statement shows that the `blendSource` property of the fourth texture used by the second shader is set to `#constant`. This enables the settings of the `blendConstant`, `blendConstantList`, and `useDiffuseWithTexture` properties

```
member("Level2").model("MysteryBox").shaderList[2].blendSourceList[4] = #constant
```

See also

`blendSource`, `blendFunction`, `blendFunctionList`, `blendConstant`, `blendConstantList`, `useDiffuseWithTexture`, `diffuse`, `diffuseColor`

blendTime

Usage

```
member(whichCastmember).model(whichModel).keyframePlayer.blendTime
member(whichCastmember).model(whichModel).bonesPlayer.blendTime
```

Description

3D `keyframePlayer` and `bonesPlayer` modifier property; determines the duration, in milliseconds, of the transition between motions in the playlist of the modifier for the model.

The `blendTime` property works in conjunction with the modifier's `autoBlend` property. When `autoBlend` is set to `TRUE`, the modifier creates a linear transition to the model's currently playing motion from the motion that preceded it. The value of the `blendTime` property is the length of that transition. The `blendTime` property is ignored if `autoBlend` is set to `FALSE`.

The default setting of this property is 500.

Example

This statement sets the length of the transition between motions in the playlist of the modifier for the model named Alien5 to 1200 milliseconds.

```
member("newaliens").model("Alien5").keyframePlayer.blendTime = 1200
```

See also

autoblend, blendFactor

bone

Usage

```
-- Lingo Usage
member(whichCastmember).modelResource(whichModelResource).bone.count
member(whichCastmember).model(whichModel).bonesPlayer.bone[index].transform
member(whichCastmember).model(whichModel).bonesPlayer.bone[index].worldTransform

// JavaScript Usage
member(whichCastmember).getProp("model",
whichModelIndex).getPropRef("bonesplayer").getPropRef("bone", whichBoneIndex).transform
member(whichCastmember).getProp("model",
whichModelIndex).getPropRef("bonesplayer").getPropRef("bone",
whichBoneIndex).worldTransform
```

Description

3D element; a bone is structural element of a model resource authored in a 3D modeling program. Bones cannot be created, deleted, or rearranged in Director.

Bones (`#bones`) motions, which also must be scripted in a 3D modeling program, act upon the bone structure of a model resource, and are managed in Director by the `bonesPlayer` modifier.

See also

count (3D), bonesPlayer (modifier), transform (property), worldTransform

bonesPlayer (modifier)

Usage

```
member(whichCastmember).model(whichModel).bonesPlayer.whichBonesPlayerProperty
```

Description

3D modifier; manages the use of motions by models. The motions managed by the `bonesPlayer` modifier animate segments, called bones, of the model.

Motions and the models that use them must be created in a 3D modeling program, exported as W3D files, and then imported into a movie. Motions cannot be applied to model primitives created within Director.

Adding the `bonesPlayer` modifier to a model by using the `addModifier` command allows access to the following `bonesPlayer` modifier properties:

playing (3D) indicates whether a model is executing a motion.

playlist is a linear list of property lists containing the playback parameters of the motions that are queued for a model.

currentTime (3D) indicates the local time, in milliseconds, of the currently playing or paused motion.

playRate (3D) is a number that is multiplied by the *scale* parameter of the play() or queue() command to determine the playback speed of the motion.

playlist.count (3D) returns the number of motions currently queued in the playlist.

rootLock indicates whether the translational component of the motion is used or ignored.

currentLoopState indicates whether the motion plays once or repeats continuously.

blendTime indicates the length of the transition created by the modifier between motions when the modifier's autoblend property is set to TRUE.

autoblend indicates whether the modifier creates a linear transition to the currently playing motion from the motion that preceded it.

blendFactor indicates the degree of blending between motions when the modifier's autoBlend property is set to FALSE.

bone[boneId].transform indicates the transform of the bone relative to the parent bone. You can find the boneId value by testing the getBoneID property of the model resource. When you set the transform of a bone, it is no longer controlled by the current motion, and cannot be returned to the control of the motion. Manual control ends when the current motion ends.

bone[boneId].getWorldTransform returns the world-relative transform of the bone.

lockTranslation indicates whether the model can be displaced from the specified planes.

positionReset indicates whether the model returns to its starting position after the end of a motion or each iteration of a loop.

rotationReset indicates the rotational element of a transition from one motion to the next, or the looping of a single motion.

Note: *For more detailed information about these properties, see the individual property entries.*

The bonesPlayer modifier uses the following commands:

pause() (3D) halts the motion currently being executed by the model.

play() (3D) initiates or unpauses the execution of a motion.

playNext() (3D) initiates playback of the next motion in the playlist.

queue() (3D) adds a motion to the end of the playlist.

The bonesPlayer modifier generates the following events, which are used by handlers declared in the registerForEvent() and registerScript() commands. The call to the declared handler includes three arguments: the event type (either #animationStarted or #animationEnded), the name of the motion, and the current time of the motion. For detailed information about notification events, see registerForEvent().

#animationStarted is sent when a motion begins playing. If blending is used between motions, the event is sent when the transition begins.

`#animationEnded` is sent when a motion ends. If blending is used between motions, the event is sent when the transition ends.

See also

keyframePlayer (modifier), addModifier, modifiers, modifier

border

Usage

```
-- Lingo syntax
memberObjRef.border

// JavaScript syntax
memberObjRef.border;
```

Description

Field cast member property; indicates the width, in pixels, of the border around the specified field cast member.

Example

This statement makes the border around the field cast member Title 10 pixels wide.

```
--Lingo syntax
member("Title").border = 10

// JavaScript syntax
member("Title").border = 10;
```

bottom

Usage

```
-- Lingo syntax
spriteObjRef.bottom

// JavaScript syntax
spriteObjRef.bottom;
```

Description

Sprite property; specifies the bottom vertical coordinate of the bounding rectangle of a sprite. Read/write.

Example

This statement assigns the vertical coordinate of the bottom of the sprite numbered (i + 1) to the variable named `lowest`.

```
-- Lingo syntax
lowest = sprite(i + 1).bottom

// JavaScript syntax
var lowest = sprite(i + 1).bottom;
```

See also

Sprite

bottom (3D)

Usage

```
member(whichCastmember).modelResource(whichModelResource).bottom
```

Description

3D `#box` model resource property; indicates whether the side of the box intersected by its -Y axis is sealed (`TRUE`) or open (`FALSE`).

The default value for this property is `TRUE`.

Example

This statement sets the `bottom` property of the model resource named GiftBox to `TRUE`, meaning the bottom of this box will be closed.

```
-- Lingo syntax
nmr = member("3D World").newModelresource("GiftBox", #box)
member("3D World").modelResource("GiftBox").bottom = TRUE

// JavaScript syntax
nmr = member("3D World").newModelresource("GiftBox",symbol("box"));
member("3D World").getProp("modelresource",10).bottom = true;
```

See also

back, front, top (3D), left (3D), right (3D), bottomCap

bottomCap

Usage

```
member(whichCastmember).modelResource(whichModelResource).bottomCap
```

Description

3D `#cylinder` model resource property; indicates whether the end of the cylinder intersected by its -Y axis is sealed (`TRUE`) or open (`FALSE`).

The default value for this property is `TRUE`.

Example

This statement sets the `bottomCap` property of the model resource named Cylinder11 to FALSE, meaning the bottom of this cylinder will be open.

```
-- Lingo syntax
nmr = member("3D World").newModelresource("Cylinder11", #cylinder);
member("3D World").modelResource("Cylinder11").bottomCap = FALSE

// JavaScript syntax
nmr = member("3D World").newModelresource("Cylinder11", symbol("cylinder"));
member("3D World").getPropRef("modelResource", 11).bottomCap = false;
```

See also

topCap, bottomRadius, bottom (3D)

bottomRadius

Usage

```
member(whichCastmember).modelResource(whichModelResource).bottomRadius
```

Description

3D `#cylinder` model resource property; indicates the radius, in world units, of the end of the cylinder that is intersected by its -Y axis.

The default value for this property is 25.

Example

This statement sets the `bottomRadius` property of the model resource named Tube to 38.5.

```
member("3D World").modelResource("Tube").bottomRadius = 38.5
```

See also

`topRadius`, `bottomCap`

bottomSpacing

Usage

```
-- Lingo syntax
chunkExpression.bottomSpacing

// JavaScript syntax
chunkExpression.bottomSpacing;
```

Description

Text cast member property; enables you to specify additional spacing applied to the bottom of each paragraph in the *`chunkExpression`* portion of the text cast member.

The value itself is an integer, where less than 0 indicates less spacing between paragraphs and greater than 0 indicates more spacing between paragraphs.

The default value is 0, which results in default spacing between paragraphs.

Note: *This property, like all text cast member properties, supports only dot syntax.*

Example

This example adds spacing after the first paragraph in cast member News Items.

```
--Lingo syntax
member("News Items").paragraph[1].bottomSpacing=20

// JavaScript syntax
member("News Items").getPropRef("paragraph", 1).bottomSpacing=20;
```

See also

`top (3D)`

boundary

Usage

```
member(whichCastmember).model(whichModel).inker.boundary
member(whichCastmember).model(whichModel).toon.boundary
```

Description

3D `inker` and `toon` modifier property; allows you to set whether a line is drawn at the edges of a model.

The default setting for this property is `TRUE`.

Example

This statement sets the `boundary` property of the inker modifier applied to the model named Box to `TRUE`. Lines will be drawn at the edges of the surface of the model.

```
-- Lingo syntax
member("shapes").model("Box").addModifier(#inker)
member("shapes").model("Box").inker.boundary = TRUE

// JavaScript syntax
member("shapes").getProp("model",1).addModifier(symbol("inker"));
member("shapes").getProp("model",1).getPropRef("inker").boundary = true;
```

See also

lineColor, lineOffset, silhouettes, creases

boundingSphere

Usage

```
member(whichCastmember).model(whichModel).boundingSphere
member(whichCastmember).group(whichGroup).boundingSphere
member(whichCastmember).light(whichLight).boundingSphere
member(whichCastmember).camera(whichCamera).boundingSphere
```

Description

3D model, group, light, and camera property; describes a sphere that contains the model, group, light, or camera and its children.

The value of this property is a list containing the vector position of the center of the sphere and the floating-point length of the sphere's radius.

This property can be tested but not set.

Example

This example displays the bounding sphere of a light in the message window.

```
-- Lingo syntax
put member("newAlien").light[5].boundingSphere
-- [vector(166.8667, -549.6362, 699.5773), 1111.0039]

// JavaScript syntax
put(member("newAlien").getProp("light",5).boundingSphere);
// <[vector(166.8667, -549.6362, 699.5773), 1111.0039]>
```

See also

debug

boxDropShadow

Usage

```
-- Lingo syntax
memberObjRef.boxDropShadow

// JavaScript syntax
memberObjRef.boxDropShadow;
```

Description

Cast member property; determines the size, in pixels, of the drop shadow for the box of the field cast member specified by *whichCastMember*.

Example

This statement makes the drop shadow of field cast member Title 10 pixels wide.

```
--Lingo syntax
member("Title").boxDropShadow = 10

// JavaScript syntax
member("Title").boxDropShadow = 10;
```

boxType

Usage

```
-- Lingo syntax
memberObjRef.boxType

// JavaScript syntax
memberObjRef.boxType;
```

Description

Cast member property; determines the type of text box used for the specified cast member. The possible values are `#adjust`, `#scroll`, `#fixed`, and `#limit`.

Example

This statement makes the box for field cast member Editorial a scrolling field.

```
--Lingo syntax
member("Editorial").boxType = #scroll

// JavaScript syntax
member("Editorial").boxType = symbol("scroll");
```

brightness

Usage

```
member(whichCastmember).shader(whichShader).brightness
member(whichCastmember).model(whichModel).shader.brightness
member(whichCastmember).model(whichModel).shaderList{[index]}.brightness
```

Description

3D `#newsprint` and `#engraver` shader property; indicates the amount of white blended into the shader.

The range of this property is 1 to 100; the default value is 0.

Example

This statement sets the brightness of the shader used by the model named gbCyl2 to half of its maximum value.

```
-- Lingo syntax
member("scene").model("gbCyl2").shader.brightness = 50

// JavaScript syntax
member("scene").getProp("shader",1).brightness = 50;
```

See also

newShader

broadcastProps

Usage

```
-- Lingo syntax
memberObjRef.broadcastProps

// JavaScript syntax
memberObjRef.broadcastProps;
```

Description

Cast member property; controls whether changes made to a Flash or Vector shape cast member are immediately broadcast to all of its sprites currently on the Stage (`TRUE`) or not (`FALSE`).

When this property is set to `FALSE`, changes made to the cast member are used only as defaults for new sprites and don't affect sprites on the Stage.

The default value for this property is `TRUE`, and it can be both tested and set.

Example

This frame script assumes that a Flash movie cast member named Navigation Movie has been set up with its `broadcastProps` property set to `FALSE`. The script momentarily allows changes to a Flash movie cast member to be broadcast to its sprites currently on the Stage. It then sets the `viewScale` property of the Flash movie cast member, and that change is broadcast to its sprite. The script then prevents the Flash movie from broadcasting changes to its sprites.

```
-- Lingo syntax
on enterFrame
    member("Navigation Movie").broadcastProps = TRUE
    member("Navigation Movie").viewScale = 200
```

```
    member("Navigation Movie").broadcastProps = FALSE
end

// JavaScript syntax
function enterFrame() {
    member("Navigation Movie").broadcastProps = 1;
    member("Navigation Movie").viewScale = 200;
    member("Navigation Movie").broadcastProps = 0;
}
```

bufferSize

Usage

```
-- Lingo syntax
memberObjRef.bufferSize

// JavaScript syntax
memberObjRef.bufferSize;
```

Description

Flash cast member property; controls how many bytes of a linked Flash movie are streamed into memory at one time. The `bufferSize` member property can have only integer values. This property has an effect only when the cast member's `preload` property is set to `FALSE`.

This property can be tested and set. The default value is 32,768 bytes.

Example

This `startMovie` handler sets up a Flash movie cast member for streaming and then sets its `bufferSize` property.

```
-- Lingo syntax
on startMovie
    member.("Flash Demo").preload = FALSE
    member.("Flash Demo").bufferSize = 65536
end

// JavaScript syntax
function startMovie() {
    member.("Flash Demo").preload = 0;
    member.("Flash Demo").bufferSize = 65536;
}
```

See also

`bytesStreamed`, `preLoadRAM`, `stream()`, `streamMode`

buttonCount

Usage

```
-- Lingo syntax
dvdObjRef.buttonCount

// JavaScript syntax
dvdObjRef.buttonCount;
```

Description

DVD property; returns the number of available buttons on the current DVD menu. Read-only.

Currently unsupported on Mac.

See also

DVD

buttonsEnabled

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.buttonsEnabled

// JavaScript syntax
memberOrSpriteObjRef.buttonsEnabled;
```

Description

Flash cast member property and sprite property; controls whether the buttons in a Flash movie are active (`TRUE`, default) or inactive (`FALSE`). Button actions are triggered only when the `actionsEnabled` property is set to `TRUE`.

This property can be tested and set.

Example

This handler accepts a sprite reference and toggles the sprite's `buttonsEnabled` property on or off.

```
-- Lingo syntax
on ToggleButtons(whichSprite)
    sprite(whichSprite).buttonsEnabled = not(sprite(whichSprite).buttonsEnabled)
end

// JavaScript syntax
function ToggleActions(whichSprite) {
    sprite(whichSprite).buttonsEnabled = !(sprite(whichSprite).buttonsEnabled);
}
```

See also

actionsEnabled

buttonStyle

Usage

```
-- Lingo syntax
_movie.buttonStyle

// JavaScript syntax
_movie.buttonStyle;
```

Description

Movie property; determines the visual response of buttons while the mouse button is held down. Read/write.

This property applies only to buttons created with the Button tool in the Tool palette.

The `buttonStyle` property can have these values:

- 0 (list style: default)—Subsequent buttons are highlighted when the pointer passes over them. Releasing the mouse button activates the script associated with that button.
- `1` (dialog style)—Only the first button clicked is highlighted. Subsequent buttons are not highlighted. Releasing the mouse button while the pointer is over a button other than the original button clicked does not activate the script associated with that button.

Example

The following statement sets the `buttonStyle` property to 1:

```
-- Lingo syntax
_movie.buttonStyle = 1

// JavaScript syntax
_movie.buttonStyle = 1;
```

This statement remembers the current setting of the `buttonStyle` property by putting the current `buttonStyle` value in the variable `buttonStyleValue`:

```
-- Lingo syntax
buttonStyleValue = _movie.buttonStyle

// JavaScript syntax
var buttonStyleValue = _movie.buttonStyle;
```

See also

`Movie`

buttonType

Usage

```
member(whichCastMember).buttonType
the buttonType of member whichCastMember
```

Description

Button cast member property; indicates the specified button cast member's type. Possible values are `#pushButton`, `#checkBox`, and `#radioButton`. This property applies only to buttons created with the button tool in the Tool palette.

Example

This statement makes the button cast member Editorial a check box.

```
--Lingo Dot syntax:
member("Editorial").buttonType = #checkBox

--Lingo Verbose syntax:
set the buttonType of member "Editorial" to #checkBox

// JavaScript syntax
member("Editorial").buttonType = symbol("checkBox");
```

bytesStreamed

Usage

```
-- Lingo syntax
memberObjRef.bytesStreamed

// JavaScript syntax
memberObjRef.bytesStreamed;
```

Description

Flash and Shockwave Audio cast member property; indicates the number of bytes of the specified cast member that have been loaded into memory. The `bytesStreamed` property returns a value only when the Director movie is playing. It returns an integer value.

This property can be tested but not set.

Example

This handler accepts a cast member reference as a parameter, and it then uses the `stream` command to load the cast member into memory. Every time it streams part of the cast member into memory, it uses the `bytesStreamed` property to report in the Message window how many bytes have been streamed.

```
-- Lingo syntax
on fetchMovie(whichFlashMovie)
    repeat while member(whichFlashMovie).percentStreamed < 100
        stream(member(whichFlashMovie))
        put("Number of bytes streamed:" && member(whichFlashMovie).bytesStreamed)
    end repeat
end

// JavaScript syntax
function fetchMovie(whichFlashMovie)
    var i = member(whichFlashMovie).percentStreamed;
    while(i < 100) {
        stream(member(whichFlashMovie));
        trace( "Number of bytes streamed: " + member(whichFlashMovie).bytesStreamed);
    }
}
```

See also

`bufferSize`, `percentStreamed (Member)`, `stream()`

bytesStreamed (3D)

Usage

```
member(whichCastMember).bytesStreamed
```

Description

3D cast member property; indicates how much of the initial file import or the last requested file load has loaded.

Example

This statement shows that 325,300 bytes of the cast member named Scene have been loaded.

```
put member("Scene").bytesStreamed
```

```
-- 325300
```

See also

streamSize (3D), state (3D)

camera

Usage

```
member(whichCastMember).camera(whichCamera)
member(whichCastMember).camera[index]
member(whichCastMember).camera(whichCamera).whichCameraProperty
member(whichCastMember).camera[index].whichCameraProperty
sprite(whichSprite).camera{(index)}
sprite(whichSprite).camera{(index)}.whichCameraProperty
```

Description

3D element; an object at a vector position from which the 3D world is viewed.

Each sprite has a list of cameras. The view from each camera in the list is displayed on top of the view from camera with lower *index* positions. You can set the rect (camera) property of each camera to display multiple views within the sprite.

Cameras are stored in the camera palette of the cast member. Use the newCamera and deleteCamera commands to create and delete cameras in a 3D cast member.

The camera property of a sprite is the first camera in the list of cameras of the sprite. The camera referred to by sprite(*whichSprite*).camera is the same as sprite(*whichSprite*).camera(1). Use the addCamera and deleteCamera commands to build the list of cameras in a 3D sprite.

For a complete list of camera properties and commands, see the Using Director topics in the Director Help Panel.

Example

This statement sets the camera of sprite 1 to the camera named TreeCam of the cast member named Picnic.

```
-- Lingo syntax
sprite(1).camera = member("Picnic").camera("TreeCam")
```

This statement sets the camera of sprite 1 to camera 2 of the cast member named Picnic.

```
-- Lingo syntax
sprite(1).camera = member("Picnic").camera[2]

// JavaScript syntax
sprite(1).camera = member("Picnic").getProp("camera", 2);
```

See also

bevelDepth, overlay, modelUnderLoc, spriteSpaceToWorldSpace, fog, clearAtRender

cameraPosition

Usage

```
member(whichCastMember).cameraPosition
sprite(whichSprite).cameraPosition
```

Description

3D cast member and sprite property; indicates the position of the default camera.

The default value of this property is vector(0, 0, 250). This is the position of the default camera in a newly created 3D cast member.

Example

This statement shows that the position of the default camera of the cast member named Babyland is the vector (-117.5992, -78.9491, 129.0254)

```
-- Lingo syntax
member("Babyland").cameraPosition = vector(-117.5992, -78.9491, 129.0254)

// JavaScript syntax
member("Babyland").cameraPosition = vector(-117.5992, -78.9491, 129.0254);
```

See also

cameraRotation, autoCameraPosition

cameraRotation

Usage

```
member(whichCastMember).cameraRotation
sprite(whichSprite).cameraRotation
```

Description

3D cast member and sprite property; indicates the position of the default camera.

The default value of this property is vector(0, 0, 0). This is the rotation of the default camera in a newly created 3D cast member.

Example

This statement shows that the rotation of the default camera of the cast member named Babyland is the vector (82.6010, -38.8530, -2.4029).

```
member("babyland").cameraRotation = vector(82.6010, -38.8530, -2.4029)
```

See also

cameraPosition, autoCameraPosition

castLib

Usage

```
-- Lingo syntax
_movie.castLib[castNameOrNum]

// JavaScript syntax
_movie.castLib[castNameOrNum];
```

Description

Movie property; provides named or indexed access to the cast libraries of a movie, whether the movie is active or not. Read-only.

The *castNameOrNum* argument can be either a string that specifies the name of the movie to access or an integer that specifies the number of the movie to access.

This property provides functionality similar to the top level `castLib()` method, except that the `castLib()` method applies only to the currently active movie.

Example

This statement displays the number of the Buttons cast in the Message window.

```
-- Lingo syntax
put(_movie.castLib["Buttons"].number)

// JavaScript syntax
put(_movie.castLib["Buttons"].number);
```

See also

castLib(), Movie

castLibNum

Usage

```
-- Lingo syntax
memberObjRef.castLibNum

// JavaScript syntax
memberObjRef.castLibNum;
```

Description

Member property; determines the number of the cast library that a cast member belongs to. Read-only.

Example

This statement determines the number of the cast to which cast member Jazz is assigned.

```
-- Lingo syntax
put(member("Jazz").castLibNum)

// JavaScript syntax
put(member("Jazz").castLibNum);
```

The following statement changes the cast member assigned to sprite 5 by switching its cast to Wednesday Schedule.

```
-- Lingo syntax
sprite(5).castLibNum = castLib("Wednesday Schedule").number

// JavaScript syntax
sprite(5).castLibNum = castLib("Wednesday Schedule").number;
```

See also

Cast Library, Member

castMemberList

Usage

```
-- Lingo syntax
memberObjRef.castMemberList

// JavaScript syntax
memberObjRef.castMemberList;
```

Description

Cursor cast member property; specifies a list of cast members that make up the frames of a cursor. For *whichCursorCastMember*, substitute a cast member name (within quotation marks) or a cast member number. You can also specify cast members from different casts.

The first cast member in the list is the first frame of the cursor, the second cast member is the second frame, and so on.

If you specify cast members that are invalid for use in a cursor, they will be ignored, and the remaining cast members will be used.

This property can be tested and set.

Example

This command sets a series of four cast members for the animated color cursor cast member named myCursor.

```
-- Lingo syntax
member("myCursor").castmemberList = [member(1), member(2), member(1, 2), member(2, 2)]

// JavaScript syntax
member("myCursor").castmemberList = list(member(1), member(2), member(1, 2), member(2, 2));
```

center

Usage

```
member(whichCastMember).center
the center of member whichCastMember
```

Description

Cast member property; interacts with the `crop` cast member property.

- When `the crop` property is `FALSE`, `the center` property has no effect.
- When `crop` is `TRUE` and `center` is `TRUE`, cropping occurs around the center of the digital video cast member.
- When `crop` is `TRUE` and `center` is `FALSE`, the digital video's right and bottom sides are cropped.

This property can be tested and set.

Example

This statement causes the digital video cast member Interview to be displayed in the top left corner of the sprite.

```
-- Lingo Dot syntax:
member("Interview").center = FALSE

-- Lingo Verbose syntax:
```

```
set the center of member "Interview" to FALSE

// JavaScript syntax
member("Interview").center = false
```

See also

crop, centerRegPoint, regPoint, scale (Member)

centerRegPoint

Usage

```
-- Lingo syntax
memberObjRef.centerRegPoint

// JavaScript syntax
memberObjRef.centerRegPoint;
```

Description

Flash, vector shape, and bitmap cast member property; automatically centers the registration point of the cast member when you resize the sprite (`TRUE`, default); or repositions the registration point at its current point value when you resize the sprite, set the `defaultRect` property, or set the `regPoint` property (`FALSE`).

This property can be tested and set.

Example

This script checks to see if a Flash movie's `centerRegPoint` property is set to `TRUE`. If it is, the script uses the `regPoint` property to reposition the sprite's registration point to its upper left corner. By checking the `centerRegPoint` property, the script ensures that it does not reposition a registration point that had been previously set using the `regPoint` property.

```
-- Lingo syntax
property spriteNum

on beginSprite me
    if sprite(spriteNum).member.centerRegPoint = TRUE then
        sprite(spriteNum).member.regPoint = point(0,0)
    end if
end

// JavaScript syntax
function beginSprite() {
    var ctrRg = sprite(this.spriteNum).member.centerRegPoint;
    if (ctrRg == 1) {
        sprite(this.spriteNum).member.regPoint = point(0,0);
    }
}
```

See also

regPoint

centerStage

Usage

```
-- Lingo syntax
_movie.centerStage

// JavaScript syntax
_movie.centerStage;
```

Description

Movie property; determines whether the Stage is centered on the monitor when the movie is loaded (`TRUE`, default) or not centered (`FALSE`). Read/write.

Place the statement that includes this property in the movie that precedes the movie you want it to affect.

This property is useful for checking the Stage location before a movie plays from a projector.

Note: *Be aware that behavior while playing back in a projector differs between Windows and Mac systems. Settings selected during creation of the projector may override this property.*

Example

This statement sends the movie to a specific frame if the Stage is not centered:

```
-- Lingo syntax
if (_movie.centerStage = FALSE) then
     _movie.go("Off Center")
end if

// JavaScript syntax
if (_movie.centerStage == false) {
    _movie.go("Off Center");
}
```

This statement changes the `centerStage` property to the opposite of its current value:

```
-- Lingo syntax
_movie.centerStage = not(_movie.centerStage)

// JavaScript syntax
_movie.centerStage = !(_movie.centerStage)
```

See also

`fixStageSize`, `Movie`

changeArea

Usage

```
member(whichCastMember).changeArea
the changeArea of member whichCastMember
```

Description

Transition cast member property; determines whether a transition applies only to the changing area on the Stage (`TRUE`) or to the entire Stage (`FALSE`). Its effect is similar to selecting the Changing Area Only option in the Frame Properties Transition dialog box.

This property can be tested and set.

Example

This statement makes the transition cast member Wave apply only to the changing area on the Stage.

```
-- Lingo Dot syntax:
member("Wave").changeArea = TRUE

-- Lingo Verbose syntax:
set the changeArea of member "Wave" to TRUE

// JavaScript syntax
member("Wave").changeArea = true;
```

channelCount

Usage

```
-- Lingo syntax
soundChannelObjRef.channelCount

// JavaScript syntax
soundChannelObjRef.channelCount;
```

Description

Sound Channel property; determines the number of channels in the currently playing or paused sound in a given sound channel. Read-only.

This property is useful for determining whether a sound is in monaural or in stereo.

Example

This statement determines the number of channels in the sound cast member, Jazz.

```
-- Lingo syntax
put(member("Jazz").channelCount)

// JavaScript syntax
put(member("Jazz").channelCount);
```

This statement determines the number of channels in the sound member currently playing in sound channel 2:

```
-- Lingo syntax
put(sound(2).channelCount)

// JavaScript syntax
put(sound(2).channelCount);
```

See also

Sound Channel

chapter

Usage

```
-- Lingo syntax
```

```
dvdObjRef.chapter

// JavaScript syntax
dvdObjRef.chapter;
```

Description

DVD property; returns the number of the current chapter. Read/write.

Example

This statement returns the current chapter:

```
-- Lingo syntax
trace (member(1).chapter)-- 1

// JavaScript syntax
trace (member(1).chapter);// 1
```

See also

DVD

chapterCount

Usage

```
-- Lingo syntax
dvdObjRef.chapterCount

// JavaScript syntax
dvdObjRef.chapterCount;
```

Description

DVD property; returns the number of available chapters in a title. Read-only.

Example

This statement returns the number of chapters in the current title:

```
-- Lingo syntax
trace (member(1).chapterCount)-- 17

// JavaScript syntax
trace (member(1). chapterCount);// 17
```

See also

chapterCount(), DVD

characterSet

Usage

```
-- Lingo syntax
memberObjRef.characterSet

// JavaScript syntax
memberObjRef.characterSet;
```

Description

Font cast member property; returns a string containing the characters included for import when the cast member was created. If all characters in the original font were included, the result is an empty string.

Example

This statement displays the characters included when cast member 11 was created. The characters included during import were numerals and Roman characters.

```
-- Lingo syntax
put(member(11).characterSet)

// JavaScript syntax
put(member(11).characterSet);
```

See also

recordFont, bitmapSizes, originalFont

charSpacing

Usage

```
-- Lingo syntax
chunkExpression.charSpacing

// JavaScript syntax
chunkExpression.charSpacing;
```

Description

Text cast member property; enables specifying any additional spacing applied to each letter in the *chunkExpression* portion of the text cast member.

A value less than 0 indicates less spacing between letters. A value greater than 0 indicates more spacing between letters.

The default value is 0, which results in default spacing between letters.

Example

The following handler increases the current character spacing of the third through fifth words within the text cast member myCaption by a value of 2:

```
--Lingo syntax
on myCharSpacer
    mySpaceValue = member("myCaption").word[3..5].charSpacing
    member("myCaption").word[3..5].charSpacing = (mySpaceValue + 2)
end

// JavaScript syntax
function myCharSpacer() {
    var i = 3;
    while (i < 6) {
        var mySpaceValue = member("myCaption").getPropRef("word", i).charSpacing;
        member("myCaption").getPropRef("word", i).charSpacing = (mySpaceValue + 2);
    }
}
```

checkMark

Usage

```
the checkMark of menuItem whichItem of menu whichMenu
```

Description

Menu item property; determines whether a check mark appears next to the custom menu item (`TRUE`) or not (`FALSE`, default).

The *`whichItem`* value can be either a menu item name or a menu item number. The *`whichMenu`* value can be either a menu name or a menu number.

This property can be tested and set.

Note: *Menus are not available in Shockwave Player.*

Example

This handler turns off any items that are checked in the custom menu specified by the argument `theMenu`. For example, `unCheck ("Format")` turns off all the items in the Format menu.

```
-- Lingo syntax
on unCheck theMenu
    set n = the number of menuItems of menu theMenu
    repeat with i = 1 to n
        set the checkMark of menuItem i of menu theMenu to FALSE
    end repeat
end unCheck

// JavaScript syntax
function unCheck (theMenu) {
    var n = _menuBar.menu[theMenu].item.count; //the number of menuItems of menu theMenu
    for( i = 1 ; i <= n ; i++ )
        _menuBar.menu[theMenu].item[i].checkMark = false;
}
```

See also

`installMenu`, `enabled`, `name (menu item property)`, `number (menu items)`, `script`, `menu`

child (3D)

Usage

```
member(whichCastmember).model(whichParentNode).child(whichChildNodeName)
member(whichCastmember).model(whichParentNode).child[index]
```

Description

3D model, group, light, and camera property; returns the child node named *`whichChildNodeName`* or at the specified index in the parent node's list of children. A node is a model, group, camera, or light.

The transform of a node is parent-relative. If you change the position of the parent, its children move with it, and their positions relative to the parent are maintained. Changes to the rotation and scale properties of the parent are similarly reflected in its children.

Use the `addChild` method of the parent node or set the `parent` property of the child node to add to the parent's list of children. A child can have only one parent, but a parent can have any number of children. A child can also have children of its own.

Example

This statement shows that the second child of the model named Car is the model named Tire.

```
-- Lingo syntax
put member("3D").model("Car").child[2]
-- model("Tire")

// JavaScript syntax
put ( member("3D").getProp("model", 1).child[2] );
// model("Tire")
```

See also

`addChild`, `parent`

child (XML)

Usage

```
XMLnode.child[ childNumber ]
```

Description

XML property; refers to the specified child node of a parsed XML document's nested tag structure.

Example

Beginning with the following XML:

```
<?xml version="1.0"?>
    <e1>
        <tagName attr1="val1" attr2="val2"/>
        <e2> element 2</e2>
        <e3>element 3</e3>
        here is some text
    </e1>
```

This Lingo returns the name of the first child node of the preceding XML:

```
put gParserObject.child[1].name
-- "e1"
```

chunkSize

Usage

```
member(whichCastMember).chunkSize
the chunkSize of member whichCastMember
```

Description

Transition cast member property; determines the transition's chunk size in pixels from 1 to 128 and is equivalent to setting the smoothness slider in the Frame Properties: Transition dialog box. The smaller the chunk size, the smoother the transition appears.

This property can be tested and set.

Example

This statement sets the chunk size of the transition cast member Fog to 4 pixels.

Dot syntax:

```
member("Fog").chunkSize = 4
```

Verbose syntax:

```
set the chunkSize of member "Fog" to 4
```

clearAtRender

Usage

```
member(whichCastmember).camera(whichCamera).colorBuffer.clearAtRender
sprite(whichSprite).camera{(index)}.colorBuffer.clearAtRender
```

Description

3D property; indicates whether the color buffer is cleared after each frame. Setting the value to `FALSE`, which means the buffer is not cleared, gives an effect similar to trails ink effect. The default value for this property is `TRUE`.

Example

This statement prevents Director from erasing past images of the view from the camera. Models in motion will appear to smear across the stage.

```
-- Lingo syntax
sprite(1).camera.colorBuffer.clearAtRender = 0

// JavaScript syntax
sprite(1).camera.getPropRef("colorBuffer").clearAtRender = 0;
```

See also

`clearValue`

clearValue

Usage

```
member(whichCastmember).camera(whichCamera).colorBuffer.clearValue
sprite(whichSprite).camera{(index)}.colorBuffer.clearValue
```

Description

3D property; specifies the color used to clear out the color buffer if `colorBuffer.clearAtRender` is set to `TRUE`. The default setting for this property is `rgb(0, 0, 0)`.

Example

This statement sets the `clearValue` property of the camera to rgb(255, 0, 0). Spaces in the 3d world which are not occupied by models will appear red.

```
-- Lingo syntax
sprite(1).camera.colorBuffer.clearValue= rgb(255, 0, 0)
```

```
// JavaScript syntax
sprite(1).camera.getPropRef("colorBuffer").clearValue= color(255, 0, 0);
```

See also

clearAtRender

clickLoc

Usage

```
-- Lingo syntax
_mouse.clickLoc

// JavaScript syntax
_mouse.clickLoc;
```

Description

Mouse property; identifies as a point the last place on the screen where the mouse was clicked. Read-only.

Example

The following on `mouseDown` handler displays the last mouse click location:

```
-- Lingo syntax
on mouseDown
    put(_mouse.clickLoc)
end mouseDown

// JavaScript syntax
function mouseDown() {
    put(_mouse.clickLoc);
}
```

If the click were 50 pixels from the left end of the Stage and 100 pixels from the top of the Stage, the Message window would display the following:

```
point(50, 100)
```

See also

clickOn, Mouse

clickMode

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.clickMode

// JavaScript syntax
memberOrSpriteObjRef.clickMode;
```

Description

Flash cast member and sprite property; controls when the Flash movie sprite detects mouse click events (`mouseUp` and `mouseDown`) and when it detects rollovers (`mouseEnter`, `mouseWithin`, and `mouseLeave`). The `clickMode` property can have these values:

- `#boundingBox`—Detects mouse click events anywhere within the sprite's bounding rectangle and detects rollovers at the sprite's boundaries.
- `#opaque` (default)—Detects mouse click events only when the pointer is over an opaque portion of the sprite and detects rollovers at the boundaries of the opaque portions of the sprite if the sprite's ink effect is set to Background Transparent. If the sprite's ink effect is not set to Background Transparent, this setting has the same effect as `#boundingBox`.
- `#object`—Detects mouse click events when the mouse pointer is over any filled (nonbackground) area of the sprite and detects rollovers at the boundaries of any filled area. This setting works regardless of the sprite's ink effect.

This property can be tested and set.

Example

This script checks to see if the sprite, which is specified with an ink effect of Background Transparent, is currently set to be rendered direct to Stage. If the sprite is not rendered direct to Stage, the sprite's `clickMode` is set to `#opaque`. Otherwise (because ink effects are ignored for Flash movie sprites that are rendered direct to Stage), the sprite's `clickMode` is set to `#boundingBox`.

```
-- Lingo syntax
property spriteNum

on beginSprite me
    if sprite(spriteNum).directToStage = FALSE then
        sprite(spriteNum).clickMode = #opaque
    else
        sprite(spriteNum).clickMode = #boundingBox
    end if
end

// JavaScript syntax
function beginSprite(me){
    var dts = sprite(this.spriteNum).directToStage;
    if (dts == 0) {
        sprite(this.spriteNum).clickMode = symbol("opaque");
    } else {
        sprite(this.spriteNum).clickMode = symbol("boundingBox");
    }
}
```

clickOn

Usage

```
-- Lingo syntax
_mouse.clickOn

// JavaScript syntax
_mouse.clickOn;
```

Description

Mouse property; returns the last active sprite clicked by the user. Read-only.

An active sprite is a sprite that has a sprite or cast member script associated with it.

When the user clicks the Stage, `clickOn` returns 0. To detect whether the user clicks a sprite with no script, you must assign a mouse event script to it so that it can be detected by `clickOn`. For example:

```
-- Lingo syntax
on mouseUp me
    ...
end
```

Buttons, check boxes, and radio buttons are detected by `clickOn` even if there is no script attached to them.

The `clickOn` property can be checked within a loop. However, neither `clickOn` nor `clickLoc` change value when the handler is running. The value that you obtain is the value from before the handler started.

Example

This statement checks whether sprite 7 was the last active sprite clicked:

```
-- Lingo syntax
if (_mouse.clickOn = 7) then
    _player.alert("Sorry, try again.")
end if

// JavaScript syntax
if (_mouse.clickOn == 7) {
    _player.alert("Sorry, try again.");
}
```

This statement sets the `foreColor` property of the last active sprite that was clicked to a random color:

```
-- Lingo syntax
sprite(_mouse.clickOn).foreColor = (random(255) - 1)

// JavaScript syntax
sprite(_mouse.clickOn).foreColor = (random(255) - 1);
```

See also

clickLoc, Mouse

closed

Usage

```
-- Lingo syntax
memberObjRef.closed

// JavaScript syntax
memberObjRef.closed;
```

Description

Vector shape cast member property; indicates whether the end points of a path are closed or open.

Vector shapes must be closed in order to contain a fill.

The value can be as follows:

- TRUE—the end points are closed.
- FALSE—the end points are open.

closedCaptions

Usage

```
-- Lingo syntax
dvdObjRef.closedCaptions

// JavaScript syntax
dvdObjRef.closedCaptions;
```

Description

DVD property. Determines whether closed captioning is enabled (TRUE), or if it is not or could not be enabled (FALSE). Currently unsupported on the Mac. Read/write.

Example

These statements try to set closedCaptions to TRUE, and display an alert if they cannot be enabled:

```
-- Lingo syntax
member(3).closedCaptions = TRUE
if (member(3).closedCaptions = FALSE) then
    _player.alert("Closed captions cannot be enabled.")
end if

// JavaScript syntax
member(3).closedCaptions = true
if (member(3).closedCaptions == false) {
    _player.alert("Closed captions cannot be enabled.");
}
```

Currently unsupported on Mac.

See also

DVD

collision (modifier)

Usage

```
member(whichCastmember).model(whichModel).collision.collisionModifierProperty
```

Description

3D modifier; manages the detection and resolution of collisions. Adding the collision modifier to a model by using the addModifier command allows access to the following collision modifier properties:

enabled (collision) indicates whether collisions with the model are detected.

resolve indicates whether collisions with the model are resolved.

immovable indicates whether a model can be moved from frame to frame.

mode (collision) indicates the geometry used for collision detection.

Note: *For more detailed information about these properties, see the individual property entries.*

The collision modifier generates the following events. For more information about using collision events, see the `registerForEvent()` entry.

A `#collideAny` event is generated when a collision occurs between models to which the `collision` modifier has been attached.

A `#collideWith` event is generated when a collision occurs with a specific model to which the `collision` modifier has been attached.

The `collisionData` object is sent as an argument with the `#collideAny` and `#collideWith` events. See the `collisionData` entry for details of its properties.

See also

`addModifier`, `removeModifier`, `modifiers`

collisionData

Usage

```
on myHandlerName me, collisionData
```

Description

3D data object; sent as an argument with the `#collideWith` and `#collideAny` events to the handler specified in the `registerForEvent`, `registerScript`, and `setCollisionCallback` commands. The `collisionData` object has these properties:

`modelA` is one of the models involved in the collision.

`modelB` is the other model involved in the collision.

`pointOfContact` is the world position of the collision.

`collisionNormal` is the direction of the collision.

Example

This example has three parts. The first part is the first line of code, which registers the `#putDetails` handler for the `#collideAny` event. The second part is the `#putDetails` handler. When two models in the cast member MyScene collide, the `#putDetails` handler is called and the `collisionData` argument is sent to it. This handler displays the four properties of the `collisionData` object in the message window. The third part of the example shows the results from the message window. The first two lines show that the model named GreenBall was model A and the model named YellowBall was model B in the collision. The third line shows the point of contact of the two models. The last line shows the direction of the collision.

```
-- Lingo syntax

member("MyScene").registerForEvent(#collideAny, #putDetails, 0)

on putDetails me, collisionData
    put collisionData.modelA
    put collisionData.modelB
    put collisionData.pointOfContact
    put collisionData.collisionNormal
end
```

```
-- model("GreenBall")
-- model("YellowBall")
-- vector( 24.800, 0.000, 0.000 )
-- vector( -1.000, 0.000, 0.000 )

// JavaScript syntax

member("MyScene").registerForEvent(symbol("collideAny"),symbol("putDetails"), 0);

function putDetails (me, collisionData)
{
    put (collisionData.modelA);
    put (collisionData.modelB);
    put (collisionData.pointOfContact);
    put (collisionData.collisionNormal);
}
// model("GreenBall")
// model("YellowBall")
// vector( 24.800, 0.000, 0.000 )
// vector( -1.000, 0.000, 0.000 )
```

See also

`collisionData properties:` modelA, modelB, pointOfContact, collisionNormal

collisionData methods: resolveA, resolveB, collision (modifier)

collisionNormal

Usage

`collisionData.collisionNormal`

Description

3D `collisionData` property; a vector indicating the direction of the collision.

The `collisionData` object is sent as an argument with the `#collideWith` and `#collideAny` events to the handler specified in the `registerForEvent`, `registerScript`, and `setCollisionCallback` commands.

The `#collideWith` and `#collideAny` events are sent when a collision occurs between models to which collision modifiers have been added. The `resolve` property of the models' modifiers must be set to `TRUE`.

This property can be tested but not set.

Example

This example has two parts. The first part is the first line of code, which registers the #explode handler for the `#collideAny` event. The second part is the `#explode` handler. When two models in the cast member named MyScene collide, the `#explode` handler is called and the `collisionData` argument is sent to it. The first ten lines of the `#explode` handler create the model resource SparkSource and set its properties. This model resource is a single burst of particles. The tenth line sets the direction of the burst to `collisionNormal`, which is the direction of the collision. The eleventh line of the handler creates a model called SparksModel using the model resource SparkSource. The last line of the handler sets the position of SparksModel to the position where the collision occurred. The overall effect is a collision that causes a burst of sparks to fly in the direction of the collision from the point of contact.

```
-- Lingo syntax
```

```
member("MyScene").registerForEvent(#collideAny, #explode, 0)
on explode me, collisionData
    nmr = member("MyScene").newModelResource("SparkSource", #particle)
    nmr.emitter.mode = #burst
    nmr.emitter.loop = 0
    nmr.emitter.minSpeed = 30
    nmr.emitter.maxSpeed = 50
    nmr.emitter.angle = 45
    nmr.colorRange.start = rgb(0, 0, 255)
    nmr.colorRange.end = rgb(255, 0, 0)
    nmr.lifetime = 5000
    nmr.emitter.direction = vector(0,0,-1)
    nm = member("MyScene").newModel("SparksModel", nmr)
    nm.transform.position = collisionData.pointOfContact
    nm.pointAt(collisionData.pointOfContact + collisionData.collisionNormal)
end

// JavaScript syntax
member("MyScene").registerForEvent(symbol("collideAny"), symbol("explode"), 0);
function explode (me, collisionData) {
    nmr = member("MyScene").newModelResource("SparkSource", symbol("particle"));
    nmr.getPropRef("emitter").mode = symbol("burst");
    nmr.getPropRef("emitter").loop = 0;
    nmr.getPropRef("emitter").minSpeed = 30;
    nmr.getPropRef("emitter").maxSpeed = 50;
    nmr.getPropRef("emitter").angle = 45;
    nmr.getPropRef("colorRange").start = color(0, 0, 255);
    nmr.getPropRef("colorRange").end = color(255, 0, 0);
    nmr.lifetime = 5000;
    nmr.getPropRef("emitter").direction = vector(0,0,-1);
    nm = member("MyScene").newModel("SparksModel", nmr);
    nm.transform.position = collisionData.pointOfContact;
    nm.pointAt(collisionData.pointOfContact + collisionData.collisionNormal);
}
```

See also

pointOfContact, modelA, modelB, resolveA, resolveB, collision (modifier)

color()

Usage

```
color(#rgb, redValue, greenValue, blueValue)
color(#paletteIndex, paletteIndexNumber)
rgb(rgbHexString)
rgb(redValue, greenValue, blueValue)
paletteIndex(paletteIndexNumber)
```

Description

Function and data type; determines an object's color as either RGB or 8-bit palette index values. These are the same values as those used in the `color` member and `color` sprite properties, the `bgColor` member and `bgColor` sprite properties, and the `bgColor` Stage property.

The `color` function allows for either 24-bit or 8-bit color values to be manipulated as well as applied to cast members, sprites, and the Stage.

For RGB values, each color component has a range from 0 to 255, and all other values are truncated. For `paletteIndex` types, an integer from 0 to 255 is used to indicate the index number in the current palette, and all other values are truncated.

Example

This statement performs a math operation:

```
palColorObj = paletteIndex(20)
put palColorObj
-- paletteIndex(20)
put palColorObj / 2
-- paletteIndex(10)
```

This statement converts one color type to another type:

```
newColorObj = color(#rgb, 155, 0, 75)
put newColorObj
-- rgb(155, 0, 75)
newColorObj.colorType = #paletteIndex
put newColorObj
-- paletteIndex(106)
```

This statement obtains the hexadecimal representation of a color regardless of its type:

```
someColorObj = color(#paletteIndex, 32)
put someColorObj.hexString()
-- "#FF0099"
```

This statement determines individual RGB components and the paletteIndex value of a color regardless of its type:

```
newColorObj = color(#rgb, 155, 0, 75)
put newColorObj.green
-- 0
put newColorObj.paletteIndex
-- 106
newColorObj.green = 100
put newColorObj.paletteIndex
-- 94
put newColorObj
-- rgb(155, 100, 75)
newColorObj.paletteIndex = 45
put newColorObj
-- paletteIndex(45)
```

This statement changes the color of the fourth through the seventh characters of text member myQuotes:

```
member("myQuotes").char[4..7].color = rgb(200, 150, 75)
```

This Lingo displays the color of sprite 6 in the Message window, and then sets the color of sprite 6 to a new RGB value:

```
put sprite(6).color
-- rgb( 255, 204, 102 )
sprite(6).color = rgb(122, 98, 210)
```

***Note:** Setting the `paletteIndex` value of an RGB color type changes `colorType` to `paletteIndex`. Setting the RGB color type of a `paletteIndex` color sets its `colorType` value to RGB.*

See also

bgColor (Window)

color (fog)

Usage

```
member(whichCastmember).camera(whichCamera).fog.color
sprite(whichSprite).camera{(index)}.fog.color
```

Description

3D property; indicates the color introduced into the scene by the camera when the camera's `fog.enabled` property is set to `TRUE`.

The default setting for this property is `rgb(0, 0, 0)`.

Example

This statement sets the color of the fog of the camera named BayView to `rgb(255, 0, 0)`. If the camera's `fog.enabled` property is set to `TRUE`, models in the fog will take on a red hue.

```
member("MyYard").camera("BayView").fog.color = rgb(255, 0, 0)
```

See also

fog

color (light)

Usage

```
member(whichCastmember).light(whichLight).color
```

Description

3D light property; indicates the rgb value of the light.

The default value of this property is `rgb(191,191,191)`.

Example

This statement sets the color of the light named RoomLight to `rgb(255, 0, 255)`.

```
member("Room").light("RoomLight").color = rgb(255,0,255)
```

See also

fog

colorBufferDepth

Usage

```
getRendererServices().colorBufferDepth
```

Description

3D `rendererServices` property; indicates the color precision of the hardware output buffer of the user's system. The value is either 16 or 32, depending on the user's hardware settings.

This property can be tested but not set.

Example

This statement shows that the `colorBufferDepth` value of the user's video card is 32.

```
-- Lingo syntax
put getRendererServices().colorBufferDepth
-- 32

// JavaScript syntax
put (getRendererServices().colorBufferDepth);
// 32
```

See also

`getRendererServices()`, `getHardwareInfo()`, `depthBufferDepth`

colorDepth

Usage

```
-- Lingo syntax
_system.colorDepth

// JavaScript syntax
_system.colorDepth;
```

Description

System property; determines the color depth of the computer's monitor. Read/write.

- In Windows, using this property lets you check and set the monitor's color depth. Some video card and driver combinations may not enable you to set the `colorDepth` property. Always verify that the color depth has actually changed after you attempt to set it.
- On the Mac, this property lets you check the color depth of different monitors and change it when appropriate.

Possible values are the following:

`1`	Black and white
`2`	4 colors
`4`	16 colors
`8`	256 colors
`16`	32,768 or 65,536 colors
`32`	16,777,216 colors

If you try to set a monitor's color depth to a value that monitor does not support, the monitor's color depth doesn't change.

On computers with more than one monitor, the `colorDepth` property refers to the monitor displaying the Stage. If the Stage spans more than one monitor, the `colorDepth` property indicates the greatest depth of those monitors; `colorDepth` tries to set all those monitors to the specified depth.

Example

This statement tells Director to open the movie Full color only if the monitor color depth is set to 256 colors:

```
-- Lingo syntax
if (_system.colorDepth = 8) then
    window("Full color").open()
end if

// JavaScript syntax
if (_system.colorDepth == 8) {
    window("Full color").open()
}
```

The following handler tries to change the color depth, and if it can't, it displays an alert:

```
-- Lingo syntax
on tryToSetColorDepth(desiredDepth)
    _system.colorDepth = desiredDepth
    if (_system.colorDepth = desiredDepth) then
        return true
    else
        _player.alert("Please change your system to" && desiredDepth && "color depth and
reboot.")
        return false
    end if
end

// JavaScript syntax
function tryToSetColorDepth(desiredDepth) {
    _system.colorDepth = desiredDepth;
    if (_system.colorDepth == desiredDepth) {
        return true;
    }
    else {
        _player.alert("Please change your system to " + desiredDepth + " color depth and
reboot.");
        return false;
    }
}
```

See also

System

colorList

Usage

```
member(whichCastmember).modelResource(whichModelResource).colorList
member(whichCastmember).modelResource(whichModelResource).colorList[index]
member(whichCastmember).model(whichModel).meshdeform.mesh[meshIndex].colorList
member(whichCastmember).model(whichModel).meshdeform.mesh[meshIndex].colorList[index]
```

Description

3D property; allows you to get or set every color used in a mesh. This command is accessible only for model resources of the type #mesh. Any single color can be shared by several vertices (faces) of the mesh. Alternately, you can specify texture coordinates for the faces of the mesh and apply a shader to models that use this model resource.

This command must be set to a list of the same number of Lingo color values specified in the newMesh call.

Example

This statement shows that the third color in the `colorList` of the model resource Mesh2 is rgb(255, 0, 0).

```
-- Lingo syntax
put member("shapes").modelResource("mesh2").colorlist[3]
-- rgb(255,0,0)

// JavaScript syntax
put ( member("shapes").getProp("modelResource" , 1).colorlist[3] );
// color(255,0,0)
```

See also

face[ ], colors

colorRange

Usage

```
member(whichCastmember).modelResource(whichModelResource).colorRange.start
member(whichCastmember).modelResource(whichModelResource).colorRange.end
```

Description

3D `#particle` model resource properties; indicate the beginning color and ending color of the particles of a particle system.

The `start` property is the color of the particles when they are created. The `end` property is the color of particles at the end of their lives. The color of each particle gradually changes from the value of `start` to the value of `end` over the course of its life.

The `start` and `end` properties have a default value of `rgb(255, 255, 255)`.

Example

This statement sets the `colorRange` properties of the model resource named ThermoSystem. The first line sets the start value to `rgb(255, 0, 0)`, and the second line sets the end value to `rgb(0, 0, 255)`. The effect of this statement is that the particles of ThermoSystem are red when they first appear, and gradually change to blue during their lifetimes.

```
member(8,2).modelResource("ThermoSystem").colorRange.start =rgb(255,0,0)
member(8,2).modelResource("ThermoSystem").colorRange.end = rgb(0,0,255)
```

See also

emitter, blendRange, sizeRange

colors

Usage

```
member(whichCastmember).modelResource(whichModelResource).face[faceIndex].colors
```

Description

3D face property; a linear list of three integers indicating which index positions of the model resource's color list to use for the three vertices of the face. The color list is a linear list of `rgb` values.

The `colors` property is used only with model resources whose type is `#mesh`.

You must use the model resource's `build()` command after setting this property; otherwise, the changes will not take effect.

Example

This example creates a model resource whose type is #mesh, specifies its properties, and then creates a new model with it.

Line 1 uses the newMesh() command to create a #mesh model resource named Triangle, which has one face, three vertices, and a maximum of three colors. The number of normals and the number of texture coordinates are not set.

Line 2 sets the `vertexList` property to a list of three vectors.

Line 3 assigns the vectors of the `vertexList` property to the vertices of the first face of Triangle.

Line 4 sets the color list to three `rgb` values.

Line 5 assigns colors to the first face of Triangle. The third color in the color list is applied to the first vertex of Triangle, the second color to the second vertex, and the first color to the third vertex. The colors will spread across the first face of Triangle in gradients.

Line 6 creates the normals of Triangle with the `generateNormals()` command.

Line 7 uses the `build()` command to construct the mesh.

Line 8 creates a new model named TriModel that uses the new mesh.

```
nm = member("Shapes").newMesh("Triangle",1,3,0,3,0)
nm.vertexList = [vector(0,0,0), vector(20,0,0), vector(20, 20, 0)]
nm.face[1].vertices = [1,2,3]
nm.colorList = [rgb(255,255,0), rgb(0, 255, 0), rgb(0,0,255)]
nm.face[1].colors = [3,2,1]
nm.generateNormals(#smooth)
nm.build()
nm = member("Shapes").newModel("TriModel", nm)
```

See also

face, vertices, vertices, flat

colorSteps

Usage

```
member(whichCastmember).model(whichModel).toon.colorSteps
member(whichCastmember).model(whichModel).shader.colorSteps
member(whichCastmember).shader(whichShader).colorSteps
```

Description

3D `toon` modifier and `painter` shader property; the maximum number of colors available for use by the `toon` modifier or `painter` shader. The value of this property can be 2, 4, 8, or 16. If you set the value of `colorSteps` to any other number, it will be rounded to one of these.

The default value is 2.

Example

This statement limits the number of colors available for use by the toon modifier for the model named Teapot to 8. The teapot will be rendered with a maximum of eight colors.

```
-- Lingo syntax
member("shapes").model("Teapot").toon.colorSteps = 8

// JavaScript syntax
member("shapes").getProp("model", 1).getPropRef("toon").colorSteps = 8;
```

See also

highlightPercentage, shadowPercentage

commandDown

Usage

```
-- Lingo syntax
_key.commandDown

// JavaScript syntax
_key.commandDown;
```

Description

Key property; determines whether the Control key (Windows) or the Command key (Mac) is being pressed. Read-only.

This property returns TRUE if the Control or Command key is being pressed; otherwise, it returns FALSE.

You can use commandDown together with the key property to determine whether the Control or Command key is being pressed in combination with another key. This lets you create handlers that are executed when the user presses specified Control or Command key combinations.

Control or Command key equivalents for the Director authoring menus take precedence while the movie is playing, unless you have installed custom Lingo or JavaScript syntax menus or are playing a projector version of the movie.

Example

These statements pause a projector when the playhead enters a frame and the user is pressing Control+A (Windows) or Command+A (Mac).

```
-- Lingo syntax
on enterFrame
    if (_key.commandDown and _key.key = "a") then
        _movie.go(_movie.frame)
    end if
end

// JavaScript syntax
function enterFrame() {
    if (_key.commandDown && _key.key == "a") {
        _movie.go(_movie.frame);
    }
}
```

See also

Key, key

comments

Usage

```
-- Lingo syntax
memberObjRef.comments

// JavaScript syntax
memberObjRef.comments;
```

Description

Member property; provides a place to store any comments you want to maintain about the given cast member or any other strings you want to associate with the member. Read/write.

This property can also be set in the Property inspector's Member tab.

Example

This statement sets the comments of the member Backdrop to the string "Still need to license this artwork":

```
-- Lingo syntax
member("Backdrop").comments = "Still need to license this artwork"

// JavaScript syntax
member("Backdrop").comments = "Still need to license this artwork";
```

See also

Member

compressed

Usage

```
member(whichCastmember).texture(whichTexture).compressed
```

Description

3D texture property; indicates whether the source cast member of the texture is compressed (TRUE) or not (FALSE). The value of the compressed property changes automatically from TRUE to FALSE when the texture is needed for rendering. It can be set to FALSE to decompress the texture at an earlier time. It can be set to TRUE to release the decompressed representation from memory. Cast members used for textures will not be compressed if this value is TRUE (apart from the standard compression used for bitmap cast members when a Director movie is saved). The default value for this property is TRUE.

Example

This statement sets the compressed property of the texture Plutomap to TRUE.

```
-- Lingo syntax
member("scene").texture("Plutomap").compressed = TRUE

// Java Script
member("scene").getProp("texture",1).compressed = true;
```

See also

texture

constraint

Usage

```
-- Lingo syntax
spriteObjRef.constraint

// JavaScript syntax
spriteObjRef.constraint;
```

Description

Sprite property; determines whether the registration point of a sprite is constrained to the bounding rectangle of another sprite (1 or TRUE) or not (0 or FALSE, default). Read/write.

The constraint property is useful for constraining a moveable sprite to the bounding rectangle of another sprite to simulate a track for a slider control or to restrict where on the screen a user can drag an object in a game.

The constraint property affects moveable sprites and the locH and locV properties. The constraint point of a moveable sprite cannot be moved outside the bounding rectangle of the constraining sprite. (The constraint point for a bitmap sprite is the registration point. The constraint point for a shape sprite is its top left corner.) When a sprite has a constraint set, the constraint limits override any locH and locV property settings.

Example

This statement removes a constraint sprite property:

```
-- Lingo syntax
sprite(5).constraint = 0

// JavaScript syntax
sprite(5).constraint = 0;
```

This statement constrains sprite (i + 1) to the boundary of sprite 14:

```
-- Lingo syntax
sprite(i + 1).constraint = 14

// JavaScript syntax
sprite(i + 1).constraint = 14;
```

This statement checks whether sprite 3 is constrained and activates the handler showConstraint if it is:

```
-- Lingo syntax
if (sprite(3).constraint <> 0) then
    showConstraint
end if

// JavaScript syntax
if (sprite(3).constraint != 0) {
    showConstraint();
}
```

See also

locH, locV, Sprite

controlDown

Usage

```
-- Lingo syntax
_key.controlDown

// JavaScript syntax
_key.controlDown;
```

Description

Key property; determines whether the Control key is being pressed. Read-only.

This property returns `TRUE` if the Control key is being pressed; otherwise, it returns `FALSE`.

You can use `controlDown` together with the `key` property to determine whether the Control key is being pressed in combination with another key. This lets you create handlers that are executed when the user presses specified Control key combinations.

Control or key equivalents for the Director authoring menus take precedence while the movie is playing, unless you have installed custom Lingo or JavaScript syntax menus or are playing a projector version of the movie.

Example

This on `keyDown` handler checks whether the pressed key is the Control key, and if it is, the handler activates the on `doControlKey` handler. The argument `(_key.key)` identifies which key was pressed in addition to the Control key.

```
-- Lingo syntax
on keyDown
    if (_key.controlDown) then
        doControlKey(_key.key)
    end if
end

on doControlKey(theKey)
    trace("The " & theKey & " key is down")
end

// JavaScript syntax
function keyDown() {
    if (_key.controlDown) {
        doControlKey(_key.key);
    }
}

function doControlKey(theKey) {
    trace("The " & theKey & " key is down");
}
```

See also

Key, key

controller

Usage

```
member(whichCastMember).controller
```

```
the controller of member whichCastMember
```

Description

Digital video cast member property; determines whether a digital video movie cast member shows or hides its controller. Setting this property to 1 shows the controller; setting it to 0 hides the controller.

The `controller` member property applies to a QuickTime® digital video only.

- Setting `the controller` member property for a Video for Windows digital video performs no operation and generates no error message.
- Checking the `controller` member property for a Video for Windows digital video always returns `FALSE`.

The digital video must be in direct-to-stage playback mode to display the controller.

Example

This statement causes the QuickTime cast member Demo to display its controller.

```
--Lingo Dot syntax:
member("Demo").controller = 1

--Lingo Verbose syntax:
set the controller of member "Demo" to 1

// JavaScript syntax
member("Demo").controller = 1;
```

See also

directToStage

copyrightInfo (Movie)

Usage

```
-- Lingo syntax
_movie.copyrightInfo

// JavaScript syntax
_movie.copyrightInfo;
```

Description

Movie property; enters a string during authoring in the Movie Properties dialog box. This property is provided to allow for enhancements in future versions of Shockwave Player. Read-only.

See also

aboutInfo, Movie

copyrightInfo (SWA)

Usage

```
-- Lingo syntax
memberObjRef.copyrightInfo
```

```
// JavaScript syntax
memberObjRef.copyrightInfo;
```

Description

Shockwave Audio (SWA) cast member property; displays the copyright text in a SWA file. This property is available only after the SWA sound begins playing or after the file has been preloaded using the `preLoadBuffer` command.

This property can be tested and set.

Example

This statement tells Director to display the copyright information for the Shockwave Audio file SWAfile in a field cast member named Info Display..

```
-- Lingo syntax
whatState = member("SWAfile").state
if whatState > 1 AND whatState < 9 then
    member("Info Display").text = member("SWAfile").copyrightInfo
end if

// JavaScript syntax
var whatState = member("SWAfile").state;
if (whatState > 1 && whatState < 9) {
    member("Info Display").text = member("SWAfile").copyrightInfo;
}
```

count

Usage

```
list.count
count (list)
count(theObject)
object.count
textExpression.count
```

Description

Property (Lingo only); returns the number of entries in a linear or property list, the number of properties in a parent script without counting the properties in an ancestor script, or the chunks of a text expression such as characters, lines, or words.

The `count` command works with linear and property lists, objects created with parent scripts, and `the globals` property.

To see an example of `count()` used in a completed movie, see the Text movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement displays the number 3, the number of entries:

```
put [10,20,30].count
-- 3
```

See also

globals

count (3D)

Usage

```
member(whichCastmember).light.count
member(whichCastmember).camera.count
member(whichCastmember).modelResource(whichModelResource).bone.count
member(whichCastmember).model.count
member(whichCastmember).group.count
member(whichCastmember).shader.count
member(whichCastmember).texture.count
member(whichCastmember).modelResource.count
member(whichCastmember).motion.count
member(whichCastmember).light.child.count
member(whichCastmember).camera.child.count
member(whichCastmember).model.child.count
member(whichCastmember).group.child.count
sprite(whichSprite).camera{(index)}.backdrop.count
member(whichCastmember).camera(whichCamera).backdrop.count
sprite(whichSprite).camera{(index)}.overlay.count
member(whichCastmember).camera(whichCamera).overlay.count
member(whichCastmember).model(whichModel).modifier.count
member(whichCastmember).model(whichModel).keyframePlayer.playlist.count
member(whichCastmember).model(whichModel).bonesPlayer.playlist.count
member(whichCastmember).modelResource(whichModelResource).face.count
member(whichCastmember).model(whichModel).meshDeform.mesh[index].textureLayer.count
member(whichCastmember).model(whichModel).meshDeform.mesh.count
member(whichCastmember).model(whichModel).meshDeform.mesh[index].face.count
```

Description

3D property; returns the number of items in the given list that is associated with the given 3D object. Can be used with any type of object.

The `face.count` property allows you to get the number of triangles in the mesh for a model resource whose type is `#mesh`.

This property can be tested but not set.

Example

These examples determine the number of various types of objects within a 3D cast member called 3D World.

```
numberOfCameras = member("3D World").camera.count
put member("3D World").light.count
-- 3
numberOfModels = member("3D World").model.count
numberOfTextures = member("3D World").texture.count
put member("3D World").modelResource("mesh2").face.count
-- 4
```

This statement shows that the first mesh of the model named Ear is composed of 58 faces.

```
put member("Scene").model("Ear").meshdeform.mesh[1].face.count
-- 58
```

This statement shows that the model named Ear is composed of three meshes.

```
put member("Scene").model("Ear").meshdeform.mesh.count
-- 3
```

This statement shows that the first mesh of the model named Ear has two texture layers.

```
put member("Scene").model("Ear").meshdeform.mesh[1].textureLayer.count
-- 2
```

See also

cameraCount()

cpuHogTicks

Usage

```
the cpuHogTicks
```

Description

System property; determines how often Director releases control of the CPU to let the computer process background events, such as events in other applications, network events, clock updates, and other keyboard events.

The default value is 20 ticks. To give more time to Director before releasing the CPU to background events or to control how the computer responds to network operations, set cpuHogTicks to a higher value.

To create faster auto-repeating key performance but slower animation, set cpuHogTicks to a lower value. In a movie, when a user holds down a key to generate a rapid sequence of auto-repeating key presses, Director typically checks for auto-repeating key presses less frequently than the rate set in the computer's control panel.

The cpuHogTicks property works only on the Mac.

Example

This statement tells Director to release control of the CPU every 6 ticks, or every 0.10 of a second:

```
the cpuHogTicks = 6
```

See also

milliseconds

creaseAngle

Usage

```
member(whichCastmember).model(whichModel).inker.creaseAngle
member(whichCastmember).model(whichModel).toon.creaseAngle
```

Description

3D inker and toon modifier property; indicates the sensitivity of the line drawing function of the modifier to the presence of creases in the model's geometry. Higher settings result in more lines (detail) drawn at creases.

The creases property of the modifier must be set to TRUE for the creaseAngle property to have an effect.

CreaseAngle has a range of -1.0 to +1.0. The default setting is 0.01.

Example

This statement sets the creaseAngle property of the inker modifier applied to the model named Teapot to 0.10. A line will be drawn at all creases in the model that exceed this threshold. This setting will only take effect if the inker modifier's creases property is set to TRUE.

```
-- Lingo syntax
member("shapes").model("Teapot").addModifier(#inker);
member("shapes").model("Teapot").inker.creaseAngle = 0.10

// JavaScript syntax
member("shapes").getProp("model",1).addModifier(symbol("inker"));
member("shapes").getProp("model",1).getPropRef("inker").creaseAngle = 0.10;
```

See also

creases, lineColor, lineOffset, useLineOffset

creases

Usage

```
member(whichCastmember).model(whichModel).inker.creases
member(whichCastmember).model(whichModel).toon.creases
```

Description

3D `toon` and `inker` modifier property; determines whether lines are drawn at creases in the surface of the model.

The default setting for this property is `TRUE`.

Example

This statement sets the `creases` property of the `inker` modifier for the model named Teapot to `TRUE`. A line will be drawn on all creases in the model that exceed the threshold set by the `inker` modifier's `creaseAngle` property.

```
-- Lingo syntax
member("shapes").model("Teapot").addModifier(#inker);
member("shapes").model("Teapot").inker.creases = TRUE

// JavaScript syntax
member("shapes").getProp("model",1).addModifier(symbol("inker"));
member("shapes").getProp("model",1).getPropRef("inker").creases = true;
```

See also

creaseAngle, lineColor, lineOffset, useLineOffset

creationDate

Usage

```
-- Lingo syntax
memberObjRef.creationDate

// JavaScript syntax
memberObjRef.creationDate;
```

Description

Member property; records the date that the cast member was first created by using the system date on the computer. Read-only.

You can use this property to schedule a project; Director does not use it for anything.

Example

Although you typically inspect the `creationDate` property using the Property inspector or the Cast window list view, you can check it in the Message window:

```
-- Lingo syntax
put(member(1).creationDate)

// JavaScript syntax
put(member(1).creationDate);
```

See also

Member

crop

Usage

```
member(whichCastMember).crop
the crop of member whichCastMember
```

Description

Cast member property; scales a digital video cast member to fit exactly inside the sprite rectangle in which it appears (`FALSE`), or it crops but doesn't scale the cast member to fit inside the sprite rectangle (`TRUE`).

This property can be tested and set.

Example

This statement instructs Lingo to crop any sprite that refers to the digital video cast member Interview.

```
-- Lingo Dot syntax:
member("Interview").crop = TRUE

-- Lingo Verbose syntax:
set the crop of member "Interview" to TRUE

// JavaScript syntax
member("Interview").crop = true;
```

See also

center

cuePointNames

Usage

```
-- Lingo syntax
memberObjRef.cuePointNames

// JavaScript syntax
memberObjRef.cuePointNames;
```

Description

Cast member property; creates list of cue point names, or if a cue point is not named, inserts an empty string (`""`) as a placeholder in the list. Cue point names are useful for synchronizing sound, QuickTime, and animation.

This property is supported by SoundEdit cast members, QuickTime digital video cast members, and Xtra extension cast members that contain cue points. Xtra extensions that generate cue points at run time may not be able to list cue point names.

Example

This statement obtains the name of the third cue point of a cast member.

```
-- Lingo syntax
put member("symphony").cuePointNames[3]

// JavaScript syntax
put(member("symphony").cuePointNames[3]);
```

See also

`cuePointTimes`, `mostRecentCuePoint`

cuePointTimes

Usage

```
-- Lingo syntax
memberObjRef.cuePointTimes

// JavaScript syntax
memberObjRef.cuePointTimes;
```

Description

Cast member property; lists the times of the cue points, in milliseconds, for a given cast member. Cue point times are useful for synchronizing sound, QuickTime, and animation.

This property is supported by SoundEdit cast members, QuickTime digital video cast members, and Xtra extension cast members that support cue points. Xtra extensions that generate cue points at run time may not be able to list cue point names.

Example

This statement obtains the time of the third cue point for a sound cast member.

```
-- Lingo syntax
put member("symphony").cuePointTimes[3]

// JavaScript syntax
put(member("symphony").cuePointTimes[3]);
```

See also

`cuePointNames`, `mostRecentCuePoint`

currentLoopState

Usage

```
member(whichCastmember).model(whichModel).keyframePlayer.currentLoopState
member(whichCastmember).model(whichModel).bonesPlayer.currentLoopState
```

Description

3D `keyframePlayer` and `bonesPlayer` modifier property; indicates whether the motion being executed by the model repeats continuously (`TRUE`) or plays to the end and is replaced by the next motion in the modifier's playlist (`FALSE`).

The default setting for this property is the value of the looped parameter of the `play()` command that initiated playback of the motion, or the value of the `queue()` command that added the motion to the modifier's playlist. Changing the `currentLoopState` property also changes the value of the `#looped` property of the motion's entry in the modifier's playlist.

Example

This statement causes the motion that is being executed by the model named Monster to repeat continuously.

```
-- Lingo syntax
member("NewAlien").model("Monster").addModifier(#keyframeplayer)
member("NewAlien").model("Monster").keyframePlayer.currentLoopState = TRUE

// JavaScript syntax
member("NewAlien").getPropRef("model",1).addModifier(symbol("keyframeplayer"));
member("NewAlien").getProp("model",1).getPropRef("keyframePlayer").currentLoopState =
true;
```

See also

`loop (3D)`, `play() (3D)`, `queue() (3D)`, `playlist`

currentSpriteNum

Usage

```
-- Lingo syntax
_player.currentSpriteNum

// JavaScript syntax
_player.currentSpriteNum;
```

Description

Player property; indicates the channel number of the sprite whose script is currently running. Read-only.

This property is valid in behaviors and cast member scripts. When used in frame scripts or movie scripts, the `currentSpriteNum` property's value is 0.

The `currentSpriteNum` property is similar to the Sprite object's `spriteNum` property.

Note: *This property was more useful during transitions from older movies to Director 6, when behaviors were introduced. It allowed some behavior-like functionality without having to completely rewrite script. It is not necessary when authoring with behaviors and is therefore less useful than in the past.*

Example

The following handler in a cast member or movie script switches the cast member assigned to the sprite involved in the `mouseDown` event:

```
-- Lingo syntax
on mouseDown
    sprite(_player.currentSpriteNum).member = member("DownPict")
end

// JavaScript syntax
function mouseDown() {
    sprite(_player.currentSpriteNum).member = member("DownPict");
}
```

See also

`Player`, `spriteNum`

currentTime (3D)

Usage

```
member(whichCastmember).model(whichModel).keyframePlayer.currentTime
member(whichCastmember).model(whichModel).bonesPlayer.currentTime
```

Description

3D `keyframePlayer` and `bonesPlayer` modifier property; indicates the local time of the motion being executed by the model. The `currentTime` property is measured in milliseconds, but it only corresponds to real time when the motion is playing at its original speed.

Playback of a motion by a model is the result of either a `play()` or `queue()` command. The `scale` parameter of the `play()` or `queue()` command is multiplied by the modifier's `playRate` property, and the resulting value is multiplied by the motion's original speed to determine how fast the model will execute the motion and how fast the motion's local time will run. So if the `scale` parameter has a value of 2 and the modifier's `playRate` property has a value of 3, the model will execute the motion six times as fast as its original speed and local time will run six times as fast as real time.

The `currentTime` property resets to the value of the `cropStart` parameter of the `play()` or `queue()` command at the beginning of each iteration of a looped motion.

Example

This statement shows the local time of the motion being executed by the model named Alien3.

```
-- Lingo syntax
member("NewAlien").model("Alien3").addModifier(#keyframeplayer)
put member("NewAlien").model("Alien3").keyframePlayer.play()
put member("NewAlien").model("Alien3").keyframePlayer.currentTime
-- 1393.8599

// JavaScript syntax
member("NewAlien").getPropRef("model",1).addModifier(symbol("keyframeplayer"));
member("NewAlien").getProp("model",1).getPropRef("keyframePlayer").play();
put(member("NewAlien").getProp("model",1).getPropRef("keyframePlayer").currentTime );
// 1393.8599
```

See also

play() (3D), queue() (3D), playlist

currentTime (DVD)

Usage

```
-- Lingo syntax
dvdObjRef.currentTime

// JavaScript syntax
dvdObjRef.currentTime;
```

Description

DVD property; returns the elapsed time, in milliseconds. Read/write.

Example

This statement returns the elapsed time:

```
-- Lingo syntax
trace (member(1).currentTime)-- 11500

// JavaScript syntax
trace (member(1).currentTime);// 11500
```

This statement sets `currentTime` to a specific point in the current title:

```
-- Lingo syntax
member(1).currentTime = 22000

// JavaScript syntax
member(1).currentTime = 22000
```

See also

DVD

currentTime (QuickTime, AVI)

Usage

```
-- Lingo syntax
spriteObjRef.currentTime

// JavaScript syntax
spriteObjRef.currentTime;
```

Description

Digital video sprite property; determines the current time of a digital video movie playing in the channel specified by *`whichSprite`*. The `movieTime` value is measured in ticks.

This property can be tested and set.

To see an example of `currentTime` used in a completed movie, see the QT and Flash movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement displays the current time of the QuickTime movie in channel 9 in the Message window:

```
-- Lingo syntax
put(sprite(9).currentTime)

// JavaScript syntax
put(sprite(9).currentTime);
```

This statement sets the current time of the QuickTime movie in channel 9 to the value in the variable `Poster`:

```
-- Lingo syntax
sprite(9).currentTime = Poster

// JavaScript syntax
sprite(9).currentTime = Poster;
```

See also

duration (Member)

currentTime (RealMedia)

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.currentTime

// JavaScript syntax
memberOrSpriteObjRef.currentTime;
```

Description

RealMedia sprite or cast member property; allows you to get or set the current time of the RealMedia stream, in milliseconds. If the RealMedia cast member is not playing, the value of this property is 0, which is the default setting. This is a playback property, and it is not saved.

If the stream is playing when the `currentTime` property is set or changed, a seek action takes place, the stream rebuffers, and then playback resumes at the new time. If the stream is paused (`#paused mediaStatus` value) when `currentTime` is set or changed, the stream redraws the frame at the new time, and it resumes playback if `pausedAtStart` is set to `FALSE`. When the stream is paused or stopped in the RealMedia viewer, `mediaStatus` is `#paused`. When the stream is stopped by the Lingo `stop` command, `mediaStatus` is `#closed`. This property has no effect if the stream's `mediaStatus` value is `#closed`. When you set integer values, they are clipped to the range from 0 to the duration of the stream.

Setting `currentTime` is equivalent to invoking the `seek` command: `x.seek(n)` is the same as `x.currentTime = n`. Changing `currentTime` or calling `seek` will require the stream to be rebuffered.

Example

The following examples show that the current time of the sprite 2 and the cast member Real is 15,534 milliseconds (15.534 seconds) from the beginning of the stream.

```
-- Lingo syntax
put(sprite(2).currentTime) -- 15534
put(member("Real").currentTime) -- 15534

// JavaScript syntax
put(sprite(2).currentTime) // 15534
```

```
put(member("Real").currentTime) // 15534
```

The following examples cause playback to jump 20,000 milliseconds (20 seconds) into the stream of sprite 2 and the cast member Real.

```
-- Lingo syntax
sprite(2).currentTime = 20000
member("Real").currentTime = 20000

// JavaScript syntax
sprite(2).currentTime = 20000
member("Real").currentTime = 20000
```

See also

duration (RealMedia, SWA), seek(), mediaStatus (RealMedia, Windows Media)

currentTime (Sprite)

Usage

```
-- Lingo syntax
spriteObjRef.currentTime

// JavaScript syntax
spriteObjRef.currentTime;
```

Description

Sprite and sound channel property; returns the current playing time, in milliseconds, for a sound sprite, QuickTime digital video sprite, or any Xtra extension that supports cue points. For a sound channel, returns the current playing time of the sound member currently playing in the given sound channel.

This property can be tested, but can only be set for traditional sound cast members (WAV, AIFF, SND). When this property is set, the range of allowable values is from zero to the `duration` of the member.

Shockwave Audio (SWA) sounds can appear as sprites in sprite channels, but they play sound in a sound channel. You should refer to SWA sound sprites by their sprite channel number rather than by a sound channel number.

Example

This statement displays the current time, in seconds, of the sound sprite in sprite channel 10.

```
-- Lingo syntax
member("time").text = string(sprite(10).currentTime/ 1000)

// JavaScript syntax
member("time").text = (sprite(10).currentTime / 1000).toString();
```

This statement causes the sound playing in sound channel 2 to skip to the point 2.7 seconds from the beginning of the sound cast member:

```
-- Lingo syntax
sound(2).currentTime = 2700

// JavaScript syntax
sound(2).currentTime = 2700;
```

See also

duration (Member)

cursor

Usage

```
-- Lingo syntax
spriteObjRef.cursor

// JavaScript syntax
spriteObjRef.cursor;
```

Description

Sprite property; determines the cursor used when the pointer is over a sprite. Read/write.

This property stays in effect until you turn it off by setting the cursor to 0. Use the `cursor` property to change the cursor when the mouse pointer is over specific regions of the screen and to indicate regions where certain actions are possible when the user clicks on them.

When you set the `cursor` property in a given frame, Director keeps track of the sprite rectangle to determine whether to alter the cursor. This rectangle persists when the movie enters another frame unless you set the `cursor` property for that channel to 0.

- Use the following syntax to specify the number of a cast member to use as a cursor and its optional mask.

```
-- Lingo syntax
spriteObjRef.cursor = [castMemberObjRef, maskCastMemberObjRef]

// JavaScript syntax
spriteObjRef.cursor = [castMemberObjRef, maskCastMemberObjRef];
```

- Use the following syntax to specify default system cursors.

```
-- Lingo syntax
spriteObjRef.cursor = castMemberObjRef

// JavaScript syntax
spriteObjRef.cursor = castMemberObjRef;
```

The `cursor` property can be set to one of the following integer values:

Value	Description
-1, 0	Arrow
1	I-Beam
2	Cross
3	Crossbar
4	Watch (Mac) or Hour glass (Windows)
5	North South East West (NSEW)
6	North South (NS)
200	Blank (hides cursor)
254	Help
256	Pencil
257	Eraser

Value	Description
258	Select
259	Bucket
260	Hand
261	Rectangle tool
262	Rounded rectangle tool
263	Circle tool
264	Line tool
265	Rich text tool
266	Text field tool
267	Button tool
268	Check box tool
269	Radio button tool
270	Placement tool
271	Registration point tool
272	Lasso
280	Finger
281	Dropper
282	Wait mouse down 1
283	Wait mouse down 2
284	Vertical size
285	Horizontal size
286	Diagonal size
290	Closed hand
291	No-drop hand
292	Copy (closed hand)
293	Inverse arrow
294	Rotate
295	Skew
296	Horizontal double arrow
297	Vertical double arrow
298	Southwest Northeast double arrow
299	Northwest Southeast double arrow
300	Smear/smooth brush

Value	Description
301	Air brush
302	Zoom in
303	Zoom out
304	Zoom cancel
305	Start shape
306	Add point
307	Close shape
308	Zoom camera
309	Move camera
310	Rotate camera
457	Custom

To use custom cursors, set the `cursor` property to a list containing the cast member to be used as a cursor or to the number that specifies a system cursor. In Windows, a cursor must be a cast member, not a resource; if a cursor is not available because it is a resource, Director displays the standard arrow cursor instead. For best results, don't use custom cursors when creating cross-platform movies.

Custom cursor cast members must be no larger than 16 by 16 pixels, and must be 1-bit in depth.

If the sprite is a bitmap that has matte ink applied, the cursor changes only when the cursor is over the matte portion of the sprite.

When the cursor is over the location of a sprite that has been removed, rollover still occurs. Avoid this problem by not performing rollovers at these locations or by relocating the sprite up above the menu bar before deleting it.

On the Mac, you can use a numbered cursor resource in the current open movie file as the cursor by setting `cursor` to the number of the cursor resource.

Example

This statement changes the cursor that appears over sprite 20 to a watch (Mac) or hourglass (Windows) cursor.

```
-- Lingo syntax
sprite(20).cursor = 4

// JavaScript syntax
sprite(20).cursor = 4;
```

See also

Sprite

cursorSize

Usage

```
-- Lingo syntax
memberObjRef.cursorSize
```

```
// JavaScript syntax
memberObjRef.cursorSize;
```

Description

Cursor cast member property; specifies the size of the animated color cursor cast member `whichCursorCastMember`.

Specify size:	For cursors up to:
`16`	16 by 16 pixels
`32`	32 by 32 pixels

Bitmap cast members smaller than the specified size are displayed at full size, and larger ones are scaled proportionally to the specified size.

The default value is 32 for Windows and 16 for the Mac. If you set an invalid value, an error message appears when the movie runs (but not when you compile).

This property can be tested and set.

Example

This command resizes the animated color cursor stored in cast member 20 to 32 by 32 pixels.

```
-- Lingo syntax
member(20).cursorSize = 32

// JavaScript syntax
member(20).cursorSize = 32;
```

curve

Usage

```
-- Lingo syntax
memberObjRef.curve[curveListIndex]

// JavaScript syntax
memberObjRef.curve[curveListIndex];
```

Description

This property contains the `vertexList` of an individual curve (shape) from a vector shape cast member. You can use the curve property along with the `vertex` property to get individual vertices of a specific curve in a vector shape.

A `vertexList` is a list of vertices, and each vertex is a property list containing up to three properties: a #vertex property with the location of the vertex, a #handle1 property with the location of the first control point for that vertex, and a #handle2 property with the location of the second control point for that vertex. See `vertexList`.

Example

This statement displays a sample list of vertices of the third curve of vector shape member SimpleCurves:

```
-- Lingo syntax
put(member("SimpleCurves").curve[3])
-- [[#vertex: point(113.0000, 40.0000), #handle1: point(32.0000, 10.0000), #handle2: point(-
32.0000, -10.0000)], [#vertex: point(164.0000, 56.0000)]]
```

```
// JavaScript syntax
put(member("SimpleCurves").curve[3]);
// [[#vertex: point(113.0000, 40.0000), #handle1: point(32.0000, 10.0000),#handle2: point(-
32.0000, -10.0000)], [#vertex: point(164.0000, 56.0000)]]
```

This statement moves the first vertex of the first curve in a vector shape down and to the right by 10 pixels:

```
-- Lingo syntax
member(1).curve[1].vertex[1] = member(1).curve[1].vertex[1] + point(10, 10)

// JavaScript syntax
member(1).curve[1].vertex[1] = member(1).curve[1].vertex[1] + point(10, 10);
```

The following code moves a sprite to the location of the first vertex of the first curve in a vector shape. The vector shape's `originMode` must be set to `#topLeft` for this to work.

```
-- Lingo syntax
vertexLoc = member(1).curve[1].vertex[1]
spriteLoc = mapMemberToStage(sprite(3), vertexLoc)
sprite(7).loc = spriteLoc

// JavaScript syntax
var vertexLoc = member(1).curve[1].vertex[1];
var spriteLoc = mapMemberToStage(sprite(3), vertexLoc);
sprite(7).loc = spriteLoc;
```

debug

Usage

```
member(whichCastmember).model(whichModel).debug
```

Description

3D model property; indicates whether the bounding sphere and local axes of the model are displayed.

Example

This statement sets the `debug` property of the model Dog to `TRUE`.

```
-- Lingo syntax
member("ParkScene").model("Dog").debug = TRUE

// JavaScript syntax
member("ParkScene").getProp("model", 1).debug = true;
```

See also

boundingSphere

debugPlaybackEnabled

Usage

```
-- Lingo syntax
_player.debugPlaybackEnabled

// JavaScript syntax
_player.debugPlaybackEnabled;
```

Description

Player property; in Windows, opens a Message window for debugging purposes in Shockwave and projectors. On the Mac, a log file is generated to allow `put` statements to output data for debugging purposes. Read/write.

In Windows, this property does not have any effect when used in the Director application. Once the Message window is closed, it cannot be reopened for a particular Shockwave Player or projector session. If more than one movie with Shockwave content uses this script in a single browser, only the first will open a Message window, and the Message window will be tied to the first movie alone.

In Macintosh , to view the message window entries, open the 1msgdump.txt file in the location \Library\Logs\.

To open this Message window, set the `debugPlaybackEnabled` property to `TRUE`. To close the window, set the `debugPlaybackEnabled` property to `FALSE`.

Example

This statement opens the Message window in either Shockwave Player or a projector:

```
-- Lingo syntax
_player.debugPlaybackEnabled = TRUE

// JavaScript syntax
_player.debugPlaybackEnabled = true;
```

See also

`Player`, `put()`

decayMode

Usage

```
member(whichCastmember).camera(whichCamera).fog.decayMode
sprite(whichSprite).camera{(index)}.fog.decayMode
```

Description

3D property; indicates the manner in which fog density builds from minimum to maximum density when the camera's `fog.enabled` property is set to `TRUE`.

The following are the possible values for this property:

- `#linear`: the fog density is linearly interpolated between `fog.near` and `fog.far`.
- `#exponential`: `fog.far` is the saturation point; `fog.near` is ignored.
- `#exponential2`: `fog.near` is the saturation point; `fog.far` is ignored.

The default setting for this property is `#exponential`.

Example

This statement sets the `decayMode` property of the fog of the camera Defaultview to `#linear`. If the fog's enabled property is set to `TRUE`, the density of the fog will steadily increase between the distances set by the fog's `near` and far properties. If the `near` property is set to 100 and the far property is set to 1000, the fog will begin 100 world units in front of the camera and gradually increase in density to a distance of 1000 world units in front of the camera.

```
-- Lingo syntax
member("3d world").camera("Defaultview").fog.decayMode = #linear
```

```
// Java Script
member("3d world").getProp("camera",1).getPropRef("fog").decayMode = symbol("linear");
```

See also

fog, near (fog), far (fog), enabled (fog)

defaultRect

Usage

```
-- Lingo syntax
memberObjRef.defaultRect

// JavaScript syntax
memberObjRef.defaultRect;
```

Description

Cast member property; controls the default size used for all new sprites created from a Flash movie or vector shape cast member. The `defaultRect` setting also applies to all existing sprites that have not been stretched on the Stage. You specify the property values as a Director rectangle; for example, rect(0,0,32,32).

The `defaultRect` member property is affected by the cast member's `defaultRectMode` member property. The `defaultRectMode` property is always set to `#Flash` when a movie is inserted into a cast, which means the original `defaultRect` setting is always the size of the movie as it was originally created in Flash. Setting `defaultRect` after that implicitly changes the cast member's `defaultRectMode` property to `#fixed`.

This property can be tested and set.

Example

This handler accepts a cast reference and a rectangle as parameters. It then searches the specified cast for Flash cast members and sets their `defaultRect` property to the specified rectangle.

```
-- Lingo syntax
on setDefaultFlashRect(whichCast, whichRect)
    repeat with i = 1 to castLib(whichCast).member.count
        if member(i, whichCast).type = #flash then
            member(i, whichCast).defaultRect = whichRect
        end if
    end repeat
end

// JavaScript syntax
function setDefaultFlashRect(whichCast, whichRect) {
    var i = 1;
    while( i < (castLib(whichCast).member.count) + 1)
        var tp = member(i, whichCast).type;
        if (tp == "flash") {
            member(i, whichCast).defaultRect = whichRect;
            i++;
        }
    }
}
```

See also

defaultRectMode, flashRect

defaultRectMode

Usage

```
-- Lingo syntax
memberObjRef.defaultRectMode

// JavaScript syntax
memberObjRef.defaultRectMode;
```

Description

Cast member property; controls how the default size is set for all new sprites created from Flash movie or vector shape cast members. You specify the property value as a Director rectangle; for example, rect(0,0,32,32).

The `defaultRectMode` property does not set the actual size of a Flash movie's default rectangle; it only determines how the default rectangle is set. The `defaultRectMode` member property can have these values:

- `#flash` (default)—Sets the default rectangle using the size of the movie as it was originally created in Flash.
- `#fixed`—Sets the default rectangle using the fixed size specified by the `defaultRect` member property.

The `defaultRect` member property is affected by the cast member's `defaultRectMode` member property. The `defaultRectMode` property is always set to `#flash` when a movie is inserted into a cast, which means the original `defaultRect` setting is always the size of the movie as it was originally created in Flash. Setting `defaultRect` after that implicitly changes the cast member's `defaultRectMode` property to `#fixed`.

This property can be tested and set.

Example

This handler accepts a cast reference and a rectangle as parameters. It then searches the specified cast for Flash cast members, sets their `defaultRectMode` property to `#fixed`, and then sets their `defaultRect` property to rect(0,0,320,240).

```
-- Lingo syntax
on setDefaultRectSize(whichCast)
    repeat with i = 1 to castLib(whichCast).member.count
        if member(i, whichCast).type = #flash then
            member(i, whichCast).defaultRectMode = #fixed
            member(i, whichCast).defaultRect = rect(0,0,320,240)
        end if
    end repeat
end

// JavaScript syntax
function setDefaultRectSize(whichCast) {
    var i = 1;
    while( i < (castLib(whichCast).member.count) + 1)
        var tp = member(i, whichCast).type;
        if (tp == "flash") {
            member(i, whichCast).defaultRectMode = symbol("fixed");
            member(i, whichCast).defaultRect = rect(0,0,320,240);
            i++;
        }
    }
}
```

See also

`flashRect`, `defaultRect`

density

Usage

```
member(whichCastmember).shader(whichShader).density
member(whichCastmember).model(whichModel).shader.density
member(whichCastmember).model(whichModel).shaderList{[index]}.density
```

Description

3D `#engraver` and `#newsprint` shader property; adjusts the number of lines or dots used to create the effects of these specialized shader types. Higher values result in more lines or dots.

For `#engraver` shaders, this property adjusts the number of lines used to create the image. The range is 0 to 100 and the default value is 40.

For `#newsprint` shaders, this property adjusts the number of dots used to create the image. The value can be from 0 to 100 and the default value is 45.

Example

The following statement sets the `density` property of the shader named EngShader to 10. The lines used by this `#engraver` shader to create its stylized image will be coarse and far apart.

```
-- Lingo syntax
member("scene").shader("EngShader").density = 10
// JavaScript syntax
member("scene").getProp("shader",1).density = 10;
```

The following statement sets the `density` property of the shader gbShader to 100. The dots used by this `#newsprint` shader to create its stylized image will be very fine and close together.

```
-- Lingo syntax
member("scene").shader("gbShader").density = 100
// JavaScript syntax
member("scene").getProp("shader", 2).density = 100;
```

See also

newShader

depth (3D)

Usage

```
member(whichCastmember).model(whichModel).sds.depth
```

Description

3D subdivision surfaces (`sds`) modifier property; specifies the maximum number of levels of resolution that the model can display when using the `sds` modifier.

If the `sds` modifier's `error` and `tension` settings are low, increasing the `depth` property will have a more pronounced effect on the model's geometry.

The `sds` modifier cannot be used with the `inker` or `toon` modifiers, and you should be careful when using the `sds` modifier with the `lod` modifier.

Example

This statement sets the `depth` property of the sds modifier for the model named Baby to 3. If the sds modifier's `error` and `tension` settings are low, this will cause a very pronounced effect on Baby's geometry.

```
-- Lingo syntax
member("Scene").model("Baby").addModifier(#sds)
member("Scene").model("Baby").sds.depth = 3

// JavaScript syntax
member("Scene").getProp("model",2).addModifier(symbol("sds"));
member("Scene").getProp("model", 2).getPropRef("sds").depth = 3;
```

See also

sds (modifier), error, tension

depth (Bitmap)

Usage

```
imageObject.depth
member(whichCastMember).depth
the depth of member whichCastMember
```

Description

Image object or bitmap cast member property; displays the color depth of the given image object or bitmap cast member.

Depth	Number of Colors
1	Black and white
2	4 colors
4, 8	16 or 256 palette-based colors, or gray levels
16	Thousands of colors
32	Millions of colors

This property can be tested but not set.

Example

This statement displays the color depth of the image object stored in the variable `newImage`. The output appears in the Message window.

```
-- Lingo syntax
put(newImage.depth)

// JavaScript syntax
trace(newImage.depth);
```

This statement displays the color depth of the cast member Shrine in the Message window:

```
-- Lingo syntax
put(member("Shrine").depth)

// JavaScript syntax
```

```
put(member("Shrine").depth);
```

depthBufferDepth

Usage

```
getRendererServices().depthBufferDepth
```

Description

3D `rendererServices` property; indicates the precision of the hardware depth buffer of the user's system. The value is either 16 or 24, depending on the user's hardware settings.

Example

This statement shows that the `depthBufferDepth` value of the user's video card is 16:

```
-- Lingo syntax
put getRendererServices().depthBufferDepth
-- 16

// JavaScript syntax
put (getRendererServices().depthBufferDepth);
// 16
```

See also

`getRendererServices()`, `getHardwareInfo()`, `colorBufferDepth`

deskTopRectList

Usage

```
-- Lingo syntax
_system.deskTopRectList

// JavaScript syntax
_system.deskTopRectList;
```

Description

System property; displays the size and position on the desktop of the monitors connected to a computer. Read-only.

This property is useful for checking whether objects such as windows, sprites, and pop-up windows appear entirely on one screen.

The result is a list of rectangles, where each rectangle is the boundary of a monitor. The coordinates for each monitor are relative to the upper left corner of monitor 1, which has the value (0,0). The first set of rectangle coordinates is the size of the first monitor. If a second monitor is present, a second set of coordinates shows where the corners of the second monitor are relative to the first monitor.

Example

This statement tests the size of the monitors connected to the computer and displays the result in the Message window:

```
-- Lingo syntax
put(_system.deskTopRectList)
```

```
// JavaScript syntax
put(_system.deskTopRectList);
```

This handler tells how many monitors are in the current system:

```
-- Lingo syntax
on countMonitors
    return _system.deskTopRectList
end

// JavaScript syntax
function countMonitors() {
    return _system.deskTopRectList;
}
```

See also

System

diffuse

Usage

```
member(whichCastmember).shader(whichShader).diffuse
member(whichCastmember).model(whichModel).shader.diffuse
member(whichCastmember).model(whichModel).shaderList{[index]}.diffuse
```

Description

3D `#standard` shader property; indicates a color that is blended with the first texture of the shader when the following conditions are met:

- the shader's `useDiffuseWithTexture` property is set to `TRUE`, and either
- the `blendFunction` property of the shader is set to `#add` or `#multiply`, or
- the `blendFunction` property of the shader is set to `#blend`, the `blendSource` property of the shader is set to `#constant`, and the value of the `blendConstant` property of the shader is less than 100.

The default value is of this property is `rgb( 255, 255, 255 )`.

Example

This statement sets the `diffuse` property of the shader named Globe to `rgb(255, 0, 0)`.

```
-- Lingo syntax
member("MysteryWorld").shader("Globe").diffuse = rgb(255, 0, 0)

// JavaScript syntax
member("MysteryWorld").getProp("shader", 1).diffuse = color(255, 0, 0);
```

See also

diffuseColor, useDiffuseWithTexture, blendFunction, blendSource, blendConstant

diffuseColor

Usage

```
member(whichCastmember).diffuseColor
```

Description

3D cast member property; indicates a color that is blended with the first texture of the first shader of the cast member when the following conditions are met:

- the shader's `useDiffuseWithTexture` property is set to `TRUE`, and either
- the `blendFunction` property of the shader is set to `#add` or `#multiply`, or
- the `blendFunction` property of the shader is set to `#blend`, the `blendSource` property of the shader is set to `#constant`, and the value of the `blendConstant` property of the shader is less than 100.

The default value is of the `diffuseColor` property is `rgb( 255, 255, 255 )`.

Example

This statement sets the `diffuseColor` property of the cast member named Room to `rgb(255, 0, 0)`.

```
-- Lingo syntax
member("Room").diffuseColor = rgb(255, 0, 0)

// JavaScript syntax
member("Room").diffuseColor = color(255, 0, 0);
```

See also

`diffuse`, `useDiffuseWithTexture`, `blendFunction`, `blendSource`, `blendConstant`

diffuseLightMap

Usage

```
member(whichCastmember).shader(whichShader).diffuseLightMap
member(whichCastmember).model(whichModel).shader.diffuseLightMap
member(whichCastmember).model(whichModel).shaderList{[index]}.diffuseLightMap
```

Description

3D `#standard` shader property; specifies the texture to use for diffuse light mapping.

When you set this property, the following properties are automatically set:

- The second texture layer of the shader is set to the texture you specified.
- The value of `textureModeList[2]` is set to `#diffuse`.
- The value of `blendFunctionList[2]` is set to `#multiply`.
- The value of `blendFunctionList[1]` is set to `#replace`.

Example

This statement sets the texture named Oval as the `diffuseLightMap` property of the shader used by the model named GlassBox.

```
-- Lingo syntax
```

```
member("3DPlanet").model("GlassBox").shader.diffuseLightMap =
member("3DPlanet").texture("Oval")

// JavaScript syntax
member("3DPlanet").getProp("model", 1).shaderList[1].diffuseLightMap =
member("3DPlanet").getprop("texture", 1);
```

See also

blendFunctionList, textureModeList, glossMap, region, specularLightMap

digitalVideoTimeScale

Usage

```
-- Lingo syntax
_player.digitalVideoTimeScale

// JavaScript syntax
_player.digitalVideoTimeScale;
```

Description

Player property; determines the time scale, in units per second, that the system uses to track digital video cast members. Read/write.

The `digitalVideoTimeScale` property can be set to any value you choose.

The value of this property determines the fraction of a second that is used to track the video, as in the following examples:

- 100—The time scale is 1/100 of a second (and the movie is tracked in 100 units per second).
- 500—The time scale is 1/500 of a second (and the movie is tracked in 500 units per second).
- 0—Director uses the time scale of the movie that is currently playing.

Set `digitalVideoTimeScale` to precisely access tracks by ensuring that the system's time unit for video is a multiple of the digital video's time unit. Set the `digitalVideoTimeScale` property to a higher value to enable finer control of video playback.

Example

This statement sets the time scale that the system uses to measure digital video to 600 units per second:

```
-- Lingo syntax
_player.digitalVideoTimeScale = 600

// JavaScript syntax
_player.digitalVideoTimeScale = 600;
```

See also

Player

digitalVideoType

Usage

```
member(whichCastMember).digitalVideoType
the digitalVideoType of member whichCastMember
```

Description

Cast member property; indicates the format of the specified digital video. Possible values are `#quickTime` or `#videoForWindows`.

This property can be tested but not set.

Example

The following statement tests whether the cast member Today's Events is a QuickTime or AVI (Audio-Video Interleaved) digital video and displays the result in the Message window:

```
-- Lingo syntax
put member("Today's Events").digitalVideoType

// JavaScript syntax
put ( member("Today's Events").digitalVideoType );
```

See also

`QuickTimeVersion()`

direction

Usage

```
member(whichCastmember).modelResource(whichModelResource).emitter.direction
```

Description

3D emitter property; a vector that indicates the direction in which the particles of a particle system are emitted. A particle system is a model resource whose type is `#particle`.

The primary direction of particle emission is the vector set by the emitter's `direction` property. However, the direction of emission of a given particle will deviate from that vector by a random angle between 0 and the value of the emitter's `angle (3D)` property.

Setting `direction` to `vector(0,0,0)` causes the particles to be emitted in all directions.

The default value of this property is `vector(1,0,0)`.

Example

In this example, ThermoSystem is a model resource whose type is `#particle`. This statement sets the `direction` property of ThermoSystem's emitter to `vector(1, 0, 0)`, which causes the particles of ThermoSystem to be emitted into a conical region whose axis is the X axis of the 3D world.

```
-- Lingo syntax
member("Fires").modelResource("ThermoSystem").emitter.direction = vector(1,0,0)

// JavaScript syntax
member("Fires").getProp("modelResource", 1).getPropRef("emitter").direction =
vector(1,0,0);
```

See also

emitter, angle (3D)

directionalColor

Usage

```
member(whichCastmember).directionalColor
```

Description

3D cast member property; indicates the RGB color of the default directional light of the cast member.

The default value of this property is rgb(255, 255, 255).

Example

This statement sets the `directionalColor` property of the cast member named Room to `rgb(0, 255, 0)`. The default directional light of the cast member will be green. This property can also be set in the Property inspector.

```
-- Lingo syntax
member("Room").directionalcolor = rgb(0, 255, 0)

// JavaScript syntax
member("Room").directionalcolor = color(0, 255, 0);
```

See also

directionalPreset

directionalPreset

Usage

```
member(whichCastmember).directionalPreset
```

Description

3D cast member property; indicates the direction from which the default directional light shines, relative to the camera of the sprite.

Changing the value of this property results in changes to the `position` and `rotation` properties of the light's transform.

Possible values of `directionalPreset` include the following:

- `#topLeft`
- `#topCenter`
- `#topRight`
- `#middleLeft`
- `#middleCenter`
- `#middleRight`
- `#bottomLeft`

- #bottomCenter
- #bottomRight
- #None

The default value of this property is #topCenter.

Example

This statement sets the directionalPreset property of the cast member named Room to #middleCenter. This points the default light of Room so it will shine on the middle center the current view of the camera of the sprite. This property can also be set in the Property inspector.

```
-- Lingo syntax
member("Room").directionalpreset = #middleCenter

// JavaScript syntax
member("Room").directionalpreset = symbol("middleCenter");
```

See also

directionalColor

directToStage

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.directToStage

// JavaScript syntax
memberOrSpriteObjRef.directToStage;
```

Description

Cast member and sprite property; determines the layer where a digital video, animated GIF, vector shape, 3D, Windows Media, or Flash Asset cast member plays.

If this property is TRUE (1), the cast member plays in front of all other layers on the Stage, and ink effects have no affect.

If this property is FALSE (0), the cast member can appear in any layer of the Stage's animation planes, and ink effects affect the appearance of the sprite.

- Use the syntax member(*whichCastMember*).directToStage for digital video or animated GIFs.
- Use the syntax sprite(*whichSprite*).directToStage for Flash or vector shapes.
- Use either syntax for 3D cast members or sprites.

Using this property improves the playback performance of the cast member or sprite.

No other cast member can appear in front of a directToStage sprite. Also, ink effects do not affect the appearance of a directToStage sprite.

When a sprite's directToStage property is TRUE, Director draws the sprite directly to the screen without first compositing it in the Director offscreen buffer. The result can be similar to the trails ink effect of the Stage.

Explicitly refresh a trailed area by turning the `directToStage` property off and on, using a full-screen transition, or "wiping" another sprite across this area. (In Windows, if you don't do this, you can branch to another similar screen, and the video may not completely disappear.)

To see an example of `directToStage` used in a completed movie, see the QT and Flash movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement makes the QuickTime movie The Residents always play in the top layer of the Stage:

```
-- Lingo syntax
member("The Residents").directToStage = 1

// JavaScript syntax
member("The Residents").directToStage = 1;
```

disableImagingTransformation

Usage

```
-- Lingo syntax
_player.disableImagingTransformation

// JavaScript syntax
_player.disableImagingTransformation;
```

Description

Player property; determines whether Director automatically takes Stage scrolling or zooming into account capturing the image of the Stage. Read/write.

When `TRUE`, this property prevents Director from automatically taking Stage scrolling or zooming into account when the `image` property is used to get the image of the Stage. Zooming and scrolling of the Stage will affect the appearance of the image captured by using `image`.

When `FALSE`, Director will always capture the image of the Stage as if the Stage window was zoomed at 100% and was not scrolled out from the center of the Stage window. `FALSE` is the default value.

Example

This statement sets `disableImagingTransformation` to `TRUE`:

```
-- Lingo syntax
_player.disableImagingTransformation = TRUE

// JavaScript syntax
_player.digitalVideoTimeScale = true;
```

See also

image (Image), Player

displayFace

Usage

```
member(whichTextCastmember).displayFace
member(which3DCastmember).modelResource(whichModelResource).displayFace
```

Description

3D text property; a linear list indicating which face or faces of the 3D text to display. Possible values include `#front`, `#tunnel`, and `#back`. You can show any combination of faces, and the list can be in any order.

The default value of this property is `[#front, #back, #tunnel]`.

For text cast members, this is a member property. For extruded text in a 3D cast member, this is a model resource property.

Example

In this example, the cast member named Rugsign is a text cast member. This statement sets the `displayFace` property of Rugsign to `[#tunnel]`. When Rugsign is displayed in 3D mode, its front and back faces will not appear.

```
-- Lingo syntax
member("Rugsign").displayFace = [#tunnel]

// JavaScript syntax
member("Rugsign").displayFace = list( symbol("tunnel") );
```

In this example, the model resource of the model named Slogan is extruded text. This statement sets the `displayFace` property of Slogan's model resource to `[#back, #tunnel]`. The front face of Slogan will not be drawn.

```
-- Lingo syntax
member("scene").model("Slogan").resource.displayFace = [#back, #tunnel]
```

See also

`extrude3D`, `displayMode`

displayMode

Usage

```
member(whichTextCastmember).displayMode
```

Description

Text cast member property; specifies whether the text will be rendered as 2D text or 3D text.

If this property is set to `#Mode3D`, the text is shown in 3D. You can set the 3D properties (such as `displayFace` and `bevelDepth`) of the text, as well as the usual text properties (such as `text` and `font`). The sprite containing this cast member becomes a 3D sprite.

If this property is set to `#ModeNormal`, the text is shown in 2D.

The default value of this property is `#ModeNormal`.

Example

In this example, the cast member named Logo is a text cast member. This statement causes Logo to be displayed in 3D.

```
-- Lingo syntax
member("Logo").displayMode = #mode3D

// JavaScript syntax
member("Logo").displayMode = symbol("mode3D");
```

See also

extrude3D

displayRealLogo

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.displayRealLogo

// JavaScript syntax
memberOrSpriteObjRef.displayRealLogo;
```

Description

RealMedia sprite or cast member property; allows you to set or get whether the RealNetworks® logo is displayed (TRUE) or not (FALSE). When set to TRUE, this property displays the RealNetworks logo in the RealMedia viewer at the beginning of the stream, when the video is stopped, or when the video is rewound.

The default value of this property is TRUE (1). Integer values other than 1 or 0 are treated as TRUE.

Example

The following examples show that the displayRealLogo property for sprite 2 and the cast member Real is set to TRUE, which means that the RealNetworks logo is displayed when the movie starts to play and when it is stopped or rewound.

```
-- Lingo syntax
put(sprite(2).displayRealLogo) -- 1
put(member("Real").displayRealLogo) -- 1

// JavaScript syntax
trace(sprite(2).displayRealLogo); // 1
put(member("Real").displayRealLogo); // 1
```

The following examples set the displayRealLogo property for sprite 2 and the cast member Real to FALSE, which means that the RealNetworks logo is not displayed.

```
-- Lingo syntax
sprite(2).displayRealLogo = FALSE
member("Real").displayRealLogo = FALSE

// JavaScript syntax
sprite(2).displayRealLogo = 0;
member("Real").displayRealLogo = 0;
```

displayTemplate

Usage

```
-- Lingo syntax
_movie.displayTemplate

// JavaScript syntax
_movie.displayTemplate;
```

Description

Movie property; provides access to a list of properties that are applied to the window in which a movie is playing back. Read/write.

The `displayTemplate` property provides access to the properties of the Window object that are used to specify default window settings. Therefore, `displayTemplate` is used on the Movie object to return or set default window settings in the same way the `appearanceOptions` and `titlebarOptions` properties are used on the Window object.

The `displayTemplate` property provides access to the following properties.

Property	Description
appearanceOptions	A property list that stores appearance options for a window. The appearance options are `mask`, `border`, `metal`, `dragRegionMask`, `shadow`, and `liveresize`. For more information, see `appearanceOptions`.
dockingEnabled	Determines whether a movie in a window (MIAW) will be dockable when opened during authoring. If `TRUE`, the window can be docked. If `FALSE`, the window cannot be docked. The default value is `FALSE`. For more information, see `dockingEnabled`.
resizable	Determines whether a window is resizable. If `TRUE`, the window is resizable. If `FALSE`, the window is not resizable. The default value is `TRUE`. For more information, see `resizable`.
title	Returns or sets the title of the display template. For more information, see `title`.
titlebarOptions	A property list that stores title bar options for a window. The title bar options are `icon`, `visible`, `closebox`, `minimizebox`, `maximizebox`, and `sideTitlebar`. For more information, see `titlebarOptions`.
systemTrayIcon	(Microsoft Windows only) Determines whether a window has an associated icon in the system tray of a user's desktop.
systemTrayTooltip	(Microsoft Windows only) Determines the string that appears in the tooltip pop-up of the system tray icon.
type	Returns or sets the type of a window. If a window's type is set, all of the properties pertaining to that window type are set as well. The types of windows are `tool`, `document`, and `dialog`. For more information, see `type`.

Example

These statements display the `displayTemplate` properties and their corresponding values in the Message window.

```
-- Lingo syntax
trace(_movie.displayTemplate)

// JavaScript syntax
trace(_movie.displayTemplate);
```

These statements set various `displayTemplate` properties.

```
-- Lingo syntax
```

```
_movie.displayTemplate.dockingEnabled = TRUE
_movie.displayTemplate.resizable = FALSE
_movie.displayTemplate.appearanceOptions.mask = member("mask")
_movie.displayTemplate.titlebarOptions.sideTitlebar = TRUE

// JavaScript syntax
_movie.displayTemplate.dockingEnabled = true;
_movie.displayTemplate.resizable = false;
_movie.displayTemplate.appearanceOptions.mask = member("mask");
_movie.displayTemplate.titlebarOptions.sideTitlebar = true;
```

See also

appearanceOptions, dockingEnabled, Movie, resizable, systemTrayIcon, title (Window), titlebarOptions, type (Window), Window

distribution

Usage

```
member(whichCastmember).modelResource(whichModelResource).emitter.distribution
```

Description

3D emitter property; indicates how the particles of a particle system are distributed across the emitter's region at their creation. The possible values of this property are `#gaussian` or `#linear`. The default value is `#linear`.

Example

In this example, ThermoSystem is a model resource whose type is `#particle`. This statement sets the `distribution` property of ThermoSystem's emitter to `#linear`, which causes the particles of ThermoSystem to be evenly distributed across their origin region at their birth.

```
-- Lingo syntax
member("Fires").modelResource("ThermoSystem").emitter.distribution = #linear

// JavaScript syntax
member("Fires").getProp("modelResource", 1).getPropRef("emitter").distribution =
symbol("linear");
```

See also

emitter, region

dither

Usage

```
-- Lingo syntax
memberObjRef.dither

// JavaScript syntax
memberObjRef.dither;
```

Description

Bitmap cast member property; dithers the cast member when it is displayed at a color depth of 8 bits or less (256 colors) if the display must show a color gradation not in the cast member (TRUE), or tells Director to choose the nearest color out of those available in the current palette (FALSE).

For both performance and quality reasons, you should set dither to TRUE only when higher display quality is necessary. Dithering is slower than remapping, and artifacts may be more apparent when animating over a dithered image.

If the color depth is greater than 8 bits, this property has no effect.

See also

depth (Bitmap)

dockingEnabled

Usage

```
-- Lingo syntax
_movie.displayTemplate.dockingEnabled
windowObjRef.dockingEnabled

// JavaScript syntax
_movie.displayTemplate.dockingEnabled;
windowObjRef.dockingEnabled;
```

Description

Movie and Window property; specifies whether a movie in a window (MIAW) will be a dockable window when opened during authoring. Read/write.

This property cannot be accessed directly from a Movie object; you access this property from the Movie object's displayTemplate property.

The default value of this property is FALSE, which specifies that a MIAW will not be dockable when opened during authoring. If this property is set to TRUE, the value of the Window object's type property determines how the window will appear during authoring.

- If dockingEnabled is TRUE and type is set to #document, the MIAW will look and act like a document windows in Director. The window will appear in the "view port" area and be dockable with the Stage, Score, and Cast windows, media editors, and message windows. However, the window will not be able to group with any of these windows.
- If dockingEnabled is TRUE and type is set to #tool, the MIAW will look and act like one of tool windows in Director. The window will be able to group with all tool windows except the Property inspector and the Tool palette.
- If dockingEnabled is TRUE and type is set to #dialog, the type is ignored and the window will be an authoring window.

This property is ignored in projectors.

Example

These statements set the dockingEnabled property to TRUE.

```
-- Lingo syntax
_movie.displayTemplate.dockingEnabled = TRUE -- from the Movie object
window("Instructions").dockingEnabled = TRUE -- from the Window object
```

```
// JavaScript syntax
_movie.displayTemplate.dockingEnabled = true; // from the Movie object
window("Instructions").dockingEnabled = true; // from the Window object
```

See also

appearanceOptions, displayTemplate, Movie, titlebarOptions, type (Window), Window

domain

Usage

```
-- Lingo syntax
dvdObjRef.domain

// JavaScript syntax
dvdObjRef.domain;
```

Description

DVD property; returns a symbol that indicates the current domain. Read-only.

Example

This statement returns the current domain:

```
-- Lingo syntax
trace (member(1).domain)-- #title

// JavaScript syntax
trace (member(1).domain);// #title
```

See also

DVD

doubleClick

Usage

```
-- Lingo syntax
_mouse.doubleClick

// JavaScript syntax
_mouse.doubleClick;
```

Description

Mouse property; tests whether two mouse clicks within the time set for a double-click occurred as a double-click rather than two single clicks (TRUE), or if they didn't occur within the time set, treats them as single clicks (FALSE). Read-only.

Example

This statement branches the playhead to the frame Enter Bid when the user double-clicks the mouse button:

```
-- Lingo syntax
if (_mouse.doubleClick) then
```

```
    _movie.go("Enter Bid")
end if

// JavaScript syntax
if (_mouse.doubleClick) {
    _movie.go("Enter Bid");
}
```

See also

clickLoc, clickOn, Mouse

drag

Usage

```
member(whichCastmember).modelResource(whichModelResource).drag
```

Description

3D #particle model resource property; indicates the percentage of each particle's velocity that is lost in each simulation step. This property has a range of 0 (no velocity lost) to 100 (all velocity lost and the particle stops moving). The default value is 0.

Example

In this example, ThermoSystem is a model resource whose type is #particle. This statement sets the drag property of ThermoSystem to 5, applying a large resistance to the motion of the particles of ThermoSystem and preventing them from traveling very far.

```
-- Lingo syntax
member("Fires").modelResource("ThermoSystem").drag = 5

// JavaScript syntax
member("Fires").getProp("modelResource", 1).drag = 5;
```

See also

wind, gravity

drawRect

Usage

```
-- Lingo syntax
windowObjRef.drawRect

// JavaScript syntax
windowObjRef.drawRect;
```

Description

Window property; identifies the rectangular coordinates of the Stage of the movie that appears in a window. Read/write.

The coordinates are given as a rectangle, with entries in the order left, top, right, and bottom.

This property is useful for scaling or panning movies, but it does not rescale text and field cast members. Scaling bitmaps can affect performance.

Example

This statement displays the current coordinates of the movie window called Control Panel:

```
-- Lingo syntax
put(window("Control Panel").drawRect)

// JavaScript syntax
put(window("Control Panel").drawRect);
```

The following statement sets the rectangle of the movie to the values of the rectangle named movieRectangle. The part of the movie within the rectangle is what appears in the window.

```
-- Lingo syntax
movieRectangle = rect(10, 20, 200, 300)
window("Control Panel").drawRect = movieRectangle

// JavaScript syntax
var movieRectangle = rect(10, 20, 200, 300);
window("Control Panel").drawRect = movieRectangle;
```

The following lines cause the Stage to fill the main monitor area:

```
-- Lingo syntax
_movie.stage.drawRect = _system.deskTopRectList[1]
_movie.stage.rect = _system.deskTopRectList[1]

// JavaScript syntax
_movie.stage.drawRect = _system.deskTopRectList[1];
_movie.stage.rect = _system.deskTopRectList[1];
```

See also

rect(), Window

dropShadow

Usage

```
-- Lingo syntax
memberObjRef.dropShadow

// JavaScript syntax
memberObjRef.dropShadow;
```

Description

Cast member property; determines the size of the drop shadow in pixels, for text in a field cast member.

Example

This statement sets the drop shadow of the field cast member Comment to 5 pixels:

```
--Lingo syntax
member("Comment").dropShadow = 5

// JavaScript syntax
member("Comment").dropShadow = 5;
```

duration (3D)

Usage

```
member(whichCastmember).motion(whichMotion).duration
motionObjectReference.duration
```

Description

3D property; lets you get the time in milliseconds that it takes the motion specified in the *whichMotion* parameter to play to completion. This property is always greater than or equal to 0.

Example

This statement shows the length in milliseconds of the motion Kick.

```
-- Lingo syntax
put member("GbMember").motion("Kick").duration
-- 5100.0000

// JavaScript syntax
put( member("GbMember").getProp("motion", 1).duration );
// 5100.0000
```

See also

motion, currentTime (3D), play() (3D), queue() (3D)

duration (DVD)

Usage

```
-- Lingo syntax
dvdObjRef.duration

// JavaScript syntax
dvdObjRef.duration;
```

Description

DVD property; returns the total title time, in milliseconds. Read-only.

Example

This statement returns the duration of the current title:

```
--Lingo syntax
trace (member(1).duration)-- 1329566

// JavaScript syntax
trace (member(1).duration);// 1329566
```

See also

DVD

duration (Member)

Usage

```
-- Lingo syntax
memberObjRef.duration

// JavaScript syntax
memberObjRef.duration;
```

Description

Cast member property; determines the duration of the specified Shockwave Audio (SWA), transition, Windows Media, and QuickTime cast members.

- When *whichCastMember* is a streaming sound file, this property indicates the duration of the sound. The `duration` property returns 0 until streaming begins. Setting `preLoadTime` to 1 second allows the bit rate to return the actual duration.
- When *whichCastMember* is a digital video cast member, this property indicates the digital video's duration. The value is in ticks.
- When *whichCastMember* is a transition cast member, this property indicates the transition's duration. The value for the transition is in milliseconds. During playback, this setting has the same effect as the Duration setting in the Frame Transition dialog box.

This property can be tested for all cast members that support it, but only set for transitions.

To see an example of `duration` used in a completed movie, see the QT and Flash movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

If the SWA cast member Louie Prima has been preloaded, this statement displays the sound's duration in the field cast member Duration Displayer:

```
-- Lingo syntax
on exitFrame
    if member("Louie Prima").state = 2 then
        member("Duration Displayer").text = string(member("Louie Prima").duration)
    end if
end

// JavaScript syntax
function exitFrame() {
    if (member("Louie Prima").state == 2) {
        member("Duration Displayer").text = member("Louie Prima").duration.toString()
    }
}
```

duration (RealMedia, SWA)

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.duration

// JavaScript syntax
memberOrSpriteObjRef.duration;
```

Description

RealMedia or Shockwave audio sprite or cast member property; returns the duration of a RealMedia or Shockwave Audio stream, in milliseconds. The duration of the stream is not known until the cast member starts to play. If the stream is from a live feed or has not been played, the value of this property is 0. This property can be tested but not set.

Example

The following examples show that the duration of the RealMedia stream in sprite 2 and the cast member Real is 100,500 milliseconds (100.500 seconds).

```
-- Lingo syntax
put(sprite(2).duration) -- 100500
put(member("Real").duration) -- 100500

// JavaScript syntax
put(sprite(2).duration); // 100500
put(member("Real").duration); // 100500
```

See also

play() (RealMedia, SWA, Windows Media), seek(), currentTime (RealMedia)

editable

Usage

```
-- Lingo syntax
spriteObjRef.editable

// JavaScript syntax
spriteObjRef.editable;
```

Description

Sprite property; determines whether a specified sprite can be edited on the Stage (TRUE) or not (FALSE). Read/write.

When the cast member property is set, the setting is applied to all sprites that contain the field.

When this property is set, only the specified sprite is affected.

You can also make a field sprite editable by using the Editable option in the Field Cast Member Properties dialog box.

You can make a field sprite editable by using the Editable option in the Score.

For the value set by a script to last beyond the current sprite, the sprite must be a scripted sprite.

Example

This handler first makes the sprite channel a puppet and then makes the field sprite editable:

```
-- Lingo syntax
on myNotes
    _movie.puppetSprite(5, TRUE)
    sprite(5).editable = TRUE
end

// JavaScript syntax
function myNotes() {
    _movie.puppetSprite(5, true);
```

```
    sprite(5).editable = true;
}
```

This statement checks whether a field sprite is editable and displays a message if it is:

```
-- Lingo syntax
if (sprite(13).editable = TRUE) then
    member("Notice").text = "Please enter your answer below."
end if

// JavaScript syntax
if (sprite(13).editable == true) {
    member("Notice").text = "Please enter your answer below.";
}
```

See also

Sprite

editShortCutsEnabled

Usage

```
-- Lingo syntax
_movie.editShortCutsEnabled

// JavaScript syntax
_movie.editShortCutsEnabled;
```

Description

Movie property; determines whether cut, copy, and paste operations and their keyboard shortcuts function in the current movie. Read/write.

When set to TRUE, these text operations function. When set to FALSE, these operations are not allowed. The default is TRUE for movies made in Director 8 and later, FALSE for movies made in versions of Director prior to Director 8.

Example

This statement disables cut, copy, and paste operations:

```
-- Lingo syntax
_movie.editShortCutsEnabled = 0

// JavaScript syntax
_movie.editShortCutsEnabled = 0;
```

See also

Movie

elapsedTime

Usage

```
-- Lingo syntax
soundChannelObjRef.elapsedTime

// JavaScript syntax
```

```
soundChannelObjRef.elapsedTime;
```

Description

Sound Channel property; gives the time, in milliseconds, that the current sound member in a sound channel has been playing. Read-only.

The elapsed time starts at 0 when the sound begins playing and increases as the sound plays, regardless of any looping, setting of the `currentTime` or other manipulation. Use the `currentTime` to test for the current absolute time within the sound.

The value of this property is a floating-point number, allowing for measurement of sound playback to fractional milliseconds.

Example

This `idle` handler displays the elapsed time for sound channel 4 in a field on the Stage during idles:

```
-- Lingo syntax
on idle
    member("time").text = string(sound(4).elapsedTime)
end idle

// JavaScript syntax
function idle() {
    member("time").text = sound(4).elapsedTime.toString();
}
```

See also

`currentTime (Sprite)`, `Sound Channel`

emissive

Usage

```
member(whichCastmember).shader(whichShader).emissive
member(whichCastmember).model(whichModel).shader.emissive
member(whichCastmember).model(whichModel).shaderList{[index]}.emissive
```

Description

3D `#standard` shader property; adds light to the shader independently of the lighting in the scene. For example, a model using a shader whose `emissive` property is set to `rgb(255, 255, 255)` will appear to be illuminated by a white light, even if there are no lights in the scene. The model will not, however, illuminate any other models or contribute any light to the scene.

The default value for this property is `rgb(0, 0, 0)`.

Example

This statement sets the `emissive` property of the shader named Globe to `rgb(255, 0, 0)`. Models using this shader will appear to be illuminated by a red light:

```
-- Lingo syntax
member("MysteryWorld").shader("Globe").emissive = rgb(255, 0, 0)

// JavaScript syntax
member("MysteryWorld").getProp("shader", 1).emissive = color(255, 0, 0);
```

See also

silhouettes

emitter

Usage

```
--Lingo Usage
member(whichCastmember).modelResource(whichModelResource).emitter.numParticles
member(whichCastmember).modelResource(whichModelResource).emitter.mode
member(whichCastmember).modelResource(whichModelResource).emitter.loop
member(whichCastmember).modelResource(whichModelResource).emitter.direction
member(whichCastmember).modelResource(whichModelResource).emitter.region
member(whichCastmember).modelResource(whichModelResource).emitter.distribution
member(whichCastmember).modelResource(whichModelResource).emitter.angle
member(whichCastmember).modelResource(whichModelResource).emitter.path
member(whichCastmember).modelResource(whichModelResource).emitter.pathStrength
member(whichCastmember).modelResource(whichModelResource).emitter.minSpeed
member(whichCastmember).modelResource(whichModelResource).emitter.maxSpeed

// JavaScript usage
member(whichCastmember).getPropRef("modelresource",whichModelResourceIndex).getPropRef("em
itter").numParticles
member(whichCastmember).getPropRef("modelresource",whichModelResourceIndex).getPropRef("em
itter").mode
member(whichCastmember).getPropRef("modelresource",whichModelResourceIndex).getPropRef("em
itter").loop
member(whichCastmember).getPropRef("modelresource",whichModelResourceIndex).getPropRef("em
itter").direction
member(whichCastmember).getPropRef("modelresource",whichModelResourceIndex).getPropRef("em
itter").region
member(whichCastmember).getPropRef("modelresource",whichModelResourceIndex).getPropRef("em
itter").distribution
member(whichCastmember).getPropRef("modelresource",whichModelResourceIndex).getPropRef("em
itter").angle
member(whichCastmember).getPropRef("modelresource",whichModelResourceIndex).getPropRef("em
itter").path
member(whichCastmember).getPropRef("modelresource",whichModelResourceIndex).getPropRef("em
itter").pathStrength
member(whichCastmember).getPropRef("modelresource",whichModelResourceIndex).getPropRef("em
itter").minSpeed
member(whichCastmember).getPropRef("modelresource",whichModelResourceIndex).getPropRef("em
itter").maxSpeed
```

Description

3D particle system element; controls the initial propulsion of particles from a model resource whose type is `#particle`.

The “See also” section of this entry contains a complete list of emitter properties. For more information, see the individual property entries.

See also

numParticles, loop (emitter), direction, distribution, region, angle (3D), path (3D), pathStrength, minSpeed, maxSpeed

emulateMultibuttonMouse

Usage

```
-- Lingo syntax
_player.emulateMultibuttonMouse

// JavaScript syntax
_player.emulateMultibuttonMouse;
```

Description

Player property; determines whether a movie interprets a mouse click with the Control key pressed on the Mac the same as a right mouse click in Windows (TRUE) or not (FALSE, default). Read/write.

Right-clicking has no direct Mac equivalent.

Setting this property to TRUE lets you provide consistent mouse button responses for cross-platform movies.

Example

The following sets the emulateMultibuttonMouse property to TRUE:

```
-- Lingo syntax
_player.emulateMultibuttonMouse = TRUE

// JavaScript syntax
_player.emulateMultibuttonMouse = true;
```

See also

Player

enabled

Usage

```
the enabled of menuItem whichItem of menu whichMenu
```

Description

Menu item property; determines whether the menu item specified by *whichItem* is displayed in plain text and is selectable (TRUE, default) or appears dimmed and is not selectable (FALSE).

The expression *whichItem* can be either a menu item name or a menu item number. The expression *whichMenu* can be either a menu name or a menu number.

The enabled property can be tested and set.

Note: *Menus are not available in Shockwave Player.*

Example

This handler enables or disables all the items in the specified menu. The argument theMenu specifies the menu; the argument Setting specifies TRUE or FALSE. For example, the calling statement ableMenu("Special", FALSE) disables all the items in the Special menu.

```
-- Lingo syntax
on ableMenu theMenu, vSetting
    set n = the number of menuItems of menu theMenu
    repeat with i = 1 to n
```

```
        set the enabled of menuItem i of menu theMenu to vSetting
    end repeat
end ableMenu
```

```
// JavaScript syntax
function ableMenu (theMenu, vSetting){
    n = _menuBar.menu[theMenu].item.count;
    for( i = 1 ; i <= n ; i++)
        _menuBar.menu[theMenu].item[i].enabled = vSetting;
}
```

See also

name (menu property), number (menus), checkMark, script, number (menu items)

enabled (collision)

Usage

```
member(whichCastmember).model(whichModel).collision.enabled
```

Description

3D collision property; allows you to get or set whether (TRUE) or not (FALSE) collisions are detected on models. Setting this property to FALSE temporarily disables the collision modifier without removing it from the model.

The default setting for this property is TRUE.

Example

This statement activates the collision modifier for the model box:

```
-- Lingo syntax
member("3d world").model("box").collision.enabled = TRUE
```

```
// JavaScript syntax
member("3d world").getProp("model",1).getPropRef("collision").enabled = true;
```

See also

addModifier, collision (modifier), modifier

enabled (fog)

Usage

```
member(whichCastmember).camera(whichCamera).fog.enabled
sprite(whichSprite).camera{(index)}.fog.enabled
```

Description

3D camera property; indicates whether the camera adds fog to the view from the camera. The default setting for this property is FALSE.

Example

This statement creates fog in the view from the camera named BayView:

```
member("MyYard").camera("BayView").fog.enabled = TRUE
```

See also

fog

enabled (sds)

Usage

```
member(whichCastmember).model(whichModel).sds.enabled
```

Description

3D sds modifier property; indicates whether the sds modifier attached to a model is used by the model.

The default setting for this property is TRUE.

An attempt to add the sds modifier to a model that already has the inker or toon modifier attached fails without an error message. Likewise, an attempt to add the inker or toon modifier to a model that already has the sds modifier attached also fails without an error message. Be careful when using the sds modifier with the lod modifier. For more information, see the sds (modifier) entry.

Example

This statement turns on the sds modifier attached to the model Baby:

```
member("Scene").model("Baby").sds.enabled = TRUE
```

See also

sds (modifier), modifier, addModifier

enableFlashLingo

Usage

```
-- Lingo syntax
_movie.enableFlashLingo

// JavaScript syntax
_movie.enableFlashLingo;
```

Description

Movie property; determines whether a sprite with Flash content can make any direct scripting callbacks when using the Flash getURL() method. Read/write.

The Flash getURL() method loads a new URL into a blank browser window.

If enableFlashLingo is set to TRUE, a sprite with Flash content can execute any valid script command (subject to standard Shockwave Player-safe rules) when getURL() is called.

If enableFlashLingo is set to FALSE, a sprite with Flash content is prevented from executing script commands when getURL() is called. The default value of this property is FALSE.

This property is useful when creating a movie that displays Flash content of unknown origin, such as in a projector that browses a system folder for SWF files, or a movie with Shockwave content that accepts a URL for a SWF file from an end user.

Example

This statement sets the `enableFlashLingo` property to `TRUE`:

```
-- Lingo syntax
_movie.enableFlashLingo = TRUE

// JavaScript syntax
_movie.enableFlashLingo = true;
```

See also

Movie

endAngle

Usage

```
member(whichCastmember).modelResource(whichModelResource).endAngle
```

Description

3D `#cylinder` or `#sphere` model resource property; indicates how much of the sphere or cylinder is drawn.

The surface of a sphere is generated by sweeping a 2D half circle arc around the sphere's Y axis from `startAngle` to `endAngle`. If `startAngle` is set to 0 and `endAngle` is set to 360, the result is a complete sphere. To draw a section of a sphere, set `endAngle` to a value less than 360.

The surface of a cylinder is generated by sweeping a 2D line around the sphere's Y axis from `startAngle` to `endAngle`. If `startAngle` is set to 0 and `endAngle` is set to 360, the result is a complete cylinder. To draw a section of a cylinder, set `endAngle` to a value less than 360.

The default setting for this property is 360.

Example

For this example, assume that the cast member named MyMember contains a model that uses the model resource named Sphere4, whose `endAngle` value is 310, leaving an opening of 50°. The handler `closeSphere` closes that opening in a way that makes it look like it is sliding shut. The repeat loop changes the `endAngle` value of the sphere 1° at a time. The `updateStage` command in the repeat loop forces the Stage to redraw after every 1° increment.

```
-- Lingo syntax
on closeSphere
    MyAngle = member("MyMember").modelresource("Sphere4").endAngle
    repeat with r = 1 to 50
        MyAngle = MyAngle + 1
        member("MyMember").modelresource("Sphere4").endAngle = MyAngle
        updateStage
    end repeat
end

// JavaScript syntax
function closeSphere(){
    var MyAngle = member("MyMember").getProp("modelresource", 1).endAngle;
    for( r = 1; r <= 50; r++)
    {
        MyAngle++;
        member("MyMember").getProp("modelresource", 1).endAngle = MyAngle;
        _movie.updateStage();
    }
```

```
}
```

See also

state (3D)

endColor

Usage

```
-- Lingo syntax
memberObjRef.endColor

// JavaScript syntax
memberObjRef.endColor;
```

Description

Vector shape cast member property; the ending color of a gradient shape's fill specified as an RGB value.

`endColor` is only valid when the `fillMode` is set to `#gradient`, and the starting color is set with `fillColor`.

This property can be tested and set.

To see an example of `endColor` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

See also

color(), fillColor, fillMode

endFrame

Usage

```
-- Lingo syntax
spriteObjRef.endFrame

// JavaScript syntax
spriteObjRef.endFrame;
```

Description

Sprite property; returns the frame number of the end frame of the sprite span. Read-only.

This property is useful in determining the span in the Score of a particular sprite.

This property is available only in a frame that contains the sprite. It cannot be applied to sprites in different frames of the movie.

Example

This statement output reports the ending frame of the sprite in channel 5 in the Message window:

```
-- Lingo syntax
put(sprite(5).endFrame)

// JavaScript syntax
put(sprite(5).endFrame);
```

See also

Sprite, startFrame

endTime

Usage

```
-- Lingo syntax
soundChannelObjRef.endTime

// JavaScript syntax
soundChannelObjRef.endTime;
```

Description

Sound Channel property; specifies the end time of the currently playing, paused, or queued sound. Read/write.

The end time is the time within the sound member when it will stop playing. It's a floating-point value, allowing for measurement and control of sound playback to fractions of milliseconds. The default value is the normal end of the sound.

This property may be set to a value other than the normal end of the sound only when passed as a parameter with the `queue()` or `setPlayList()` methods.

Example

These statemenets check whether the sound member Jingle is set to play all the way through in sound channel 1:

```
-- Lingo syntax
if (sound(1).startTime > 0 and sound(1).endTime < member("Jingle").duration) then
_player.alert("Not playing the whole sound.")
end if

// JavaScript syntax
if (sound(1).startTime > 0 && sound(1).endTime < member("Jingle").duration) {
    _player.alert("Not playing the whole sound.");
}
```

See also

queue(), setPlayList(), Sound Channel

environmentPropList

Usage

```
-- Lingo syntax
_system.environmentPropList

// JavaScript syntax
_system.environmentPropList;
```

Description

System property; contains a list with information about the environment under which the Director content is currently running. Read-only.

This design enables Adobe to add information to the `environmentPropList` property in the future, without affecting existing movies.

The information is in the form of property and value pairs for that area.

`#shockMachine`	Integer `TRUE` or `FALSE` value indicating whether the movie is playing in ShockMachine.
`#shockMachineVersion`	String indicating the installed version number of ShockMachine.
`#platform`	String containing "Mac,PowerPC", or "Windows,32" . This is based on the current OS and hardware that the movie is running under.
`#runMode`	String containing "Author", "Projector", or "Plugin". This is based on the current application that the movie is running under.
`#colorDepth`	Integer representing the bit depth of the monitor the Stage appears on. Possible values are 1, 2, 4, 8, 16, or 32.
`#internetConnected`	Symbol indicating whether the computer the movie is playing on has an active Internet connection. Possible values are `#online` and `#offline`.
`#uiLanguage`	String indicating the language the player is using to display its user interface.
`#osLanguage`	String indicating the native language of the computer's operating system.
#osVersion	The value of #osVersion is a string. On Windows, The #osVersion property is is populated with information obtained with the GetVersionEx() system call. The values in the string are from the OSVERSIONINFO structure: "Windows CE" or "Windows NT" or "Windows 2000" or Windows XP" or "Windows 95" or "Windows 98" or "Windows ME" dwMajorVersion dwMinorVersion dwOSVersionInfoSize dwPlatformId szCSDVersion On Mac, the values in the string are from the Gestalt(gestaltSystemVersion) call: "Mac OS" major version minor version sub-version
`#productBuildVersion`	String indicating the internal build number of the playback application.

The properties contain exactly the same information as the properties and functions of the same name.

Example

This statement displays the environment list in the Message window:

```
-- Lingo syntax
put(_system.environmentPropList)

// JavaScript syntax
put(_system.environmentPropList);
```

See also

System

error

Usage

```
member(whichCastmember).model(whichModel).sds.error
```

Description

3D #sds modifier property; indicates the percentage of error tolerated by the modifier when synthesizing geometric detail in models.

This property works only when the modifier's subdivision property is set to #adaptive. The tension and depth (3D) properties of the modifier combine with the error property to control the amount of subdivision performed by the modifier.

Example

The following statement sets the error property of the #sds modifier of the model named Baby to 0. If the modifier's tension setting is low, its depth setting is high, and its subdivision setting is #adaptive, this will cause a very pronounced effect on Baby's geometry.

```
-- Lingo syntax
member("Scene").model("Baby").addModifier(#sds)
member("Scene").model("Baby").sds.subdivision = #adaptive
member("Scene").model("Baby").sds.error = 0

// JavaScript syntax
member("Scene").getProp("model",2).addModifier(symbol("sds"));
member("Scene").getProp("model",2).getPropRef("sds").subdivision = symbol("adaptive");
member("Scene").getProp("model",2).getPropRef("sds").error = 0;
```

See also

sds (modifier), subdivision, depth (3D), tension

eventPassMode

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.eventPassMode

// JavaScript syntax
memberOrSpriteObjRef.eventPassMode;
```

Description

Flash cast member property and sprite property; controls when a Flash movie passes mouse events to behaviors that are attached to sprites that lie underneath the flash sprite. The eventPassMode property can have these values:

- #passAlways (default)—Always passes mouse events.
- #passButton—Passes mouse events only when a button in the Flash movie is clicked.
- #passNotButton—Passes mouse events only when a nonbutton object is clicked.

- `#passNever`—Never passes mouse events.

This property can be tested and set.

Example

The following frame script checks to see whether the buttons in a Flash movie sprite are currently enabled, and if so, sets `eventPassMode` to `#passNotButton`; if the buttons are disabled, the script sets `eventPassMode` to `#passAlways`. The effect of this script is the following:

- Mouse events on nonbutton objects alwa.ys pass to sprite scripts.
- Mouse events on button objects are passed to sprite scripts when the buttons are disabled. When the buttons are enabled, mouse events on buttons are stopped.

```
-- Lingo syntax
on enterFrame
    if sprite(5).buttonsEnabled = TRUE then
        sprite(5).eventPassMode= #passNotButton
    else
        sprite(5).eventPassMode = #passAlways
    end if
end

// JavaScript syntax
function enterFrame() {
    var btEn = sprite(5).buttonsEnabled;
    if (btEn == 1) {
        sprite(5).eventPassMode= symbol("passNotButton");
    } else {
        sprite(5).eventPassMode = symbol("passAlways");
    }
}
```

exitLock

Usage

```
-- Lingo syntax
_movie.exitLock

// JavaScript syntax
_movie.exitLock;
```

Description

Movie property; determines whether a user can quit to the Windows desktop or Mac Finder from projectors (`FALSE`, default) or not (`TRUE`). Read/write.

The user can quit to the desktop by pressing Control+period (Windows) or Command+period (Mac), Control+Q (Windows) or Command+Q (Mac), or Control+W (Windows) or Command+W (Mac); the Escape key is also supported in Windows.

Example

This statement sets the `exitLock` property to `TRUE`:

```
-- Lingo syntax
_movie.exitLock = TRUE
```

```
// JavaScript syntax
_movie.exitLock = true;
```

Assuming that `exitLock` is set to `TRUE`, nothing occurs automatically when the Control+period/Q/W, Esc, or Command+period/Q/W keys are used. This handler checks keyboard input for keys to exit and takes the user to a custom quit sequence:

```
-- Lingo syntax
on checkExit
    if ((_key.commandDown) and (_key.key = "." or _key.key = "q") and (_movie.exitLock =
TRUE)) then _movie.go("quit sequence")
end checkExit

// JavaScript syntax
function checkExit() {
    if ((_key.commandDown) && (_key.key == "." || _key.key == "q") && (_movie.exitLock ==
true)) {
        _movie.go("quit sequence");
    }
}
```

See also

Movie

externalParamCount

Usage

```
-- Lingo syntax
_player.externalParamCount

// JavaScript syntax
_player.externalParamCount;
```

Description

Player property; returns the number of parameters that an HTML <EMBED> or <OBJECT> tag is passing to a movie with Shockwave content. Read-only.

This property is valid only for movies with Shockwave content that are running in a browser. It doesn't work for movies during authoring or for projectors.

For more information about the valid external parameters, see `externalParamName()` and `externalParamValue()`.

Example

This handler determines whether an <OBJECT> or <EMBED> tag is passing any external parameters to a movie with Shockwave content and runs Lingo statements if parameters are being passed:

```
-- Lingo syntax
if (_player.externalParamCount > 0) then
    -- perform some action
end if

// JavaScript syntax
if (_player.externalParamCount > 0) {
    // perform some action;
}
```

See also

externalParamName(), externalParamValue(), Player

face

Usage

```
\-- Lingo Usage
member(whichCastmember).modelResource(whichModelResource).face.count
member(whichCastmember).modelResource(whichModelResource).face[index].colors
member(whichCastmember).modelResource(whichModelResource).face[index].normals
member(whichCastmember).modelResource(whichModelResource).face[index].shader
member(whichCastmember).modelResource(whichModelResource).face[index].textureCoordinates
member(whichCastmember).modelResource(whichModelResource).face[index].vertices
member(whichCastmember).model(whichModel).meshdeform.face.count
member(whichCastmember).model(whichModel).meshdeform.mesh[index].face.count
member(whichCastmember).model(whichModel).meshdeform.mesh[meshIndex].face[faceIndex]
member(whichCastmember).model(whichModel).meshdeform.mesh[meshIndex].face[faceIndex].neigh
bor{[neighborIndex]}

// JavaScript Usage
member(whichCastmember).getProp("modelResource",
whichModelResourceIndex).getPropRef("face", index).colors
member(whichCastmember).getProp("modelResource",
whichModelResourceIndex).getPropRef("face", index).normals
member(whichCastmember).getProp("modelResource",
whichModelResourceIndex).getPropRef("face", index).shader
member(whichCastmember).getProp("modelResource",
whichModelResourceIndex).getPropRef("face", index).textureCoordinates
member(whichCastmember).getProp("modelResource",
whichModelResourceIndex).getPropRef("face", index).vertices
member(whichCastmember).getProp("model",
whichModelIndex).getPropRef("meshdeform").getPropRef("mesh", meshIndex).face[faceIndex]
member(whichCastmember).getProp("model",
whichModelIndex).getPropRef("meshdeform").getPropRef("mesh",
meshIndex).face[faceIndex].neighbor{[neighborIndex]}
```

Description

3D `#mesh` model resource and `meshdeform` modifier property. All model resources are meshes composed of triangles. Each triangle is a face.

You can access the properties of the faces of model resources whose type is `#mesh`. Changes to any of these properties do not take effect until you call the `build()` command.

Note: *For detailed information about the following properties, see the individual property entries.*

- `count` indicates the number of triangles in the mesh.
- `colors` indicates which indices in the color list of the model resource to use for each of the vertices of the face.
- `normals` indicates which indices in the normal list of the model resource to use for each of the vertices of the face.
- `shadowPercentage` identifies the shader used when the face is rendered.
- `textureCoordinates` indicates which indices in the texture coordinate list of the model resource to use for each of the vertices of the face.
- `vertices` indicates which indices in the vertex list of the model resource to use to define the face.

See the entry for meshDeform for descriptions of its face properties.

See also

build(), newMesh, meshDeform (modifier)

face[ ]

Usage

```
member(whichCastmember).model(whichModel).meshdeform.mesh[meshIndex].face[faceIndex]
```

Description

3D meshdeform modifier property; indicates which indices in the vertex list of the model resource were used to define the face.

This property can be tested but not set. You can specify the vertices of a face of the #mesh model resource by setting its vertexList and vertices properties and calling the build command.

Example

This statement shows that the first face of the first mesh of the model named Floor is defined by the first three vectors in the vertex list of the model resource used by Floor:

```
put member("Scene").model("Floor").meshdeform.mesh[1].face[1]
-- [1, 2, 3]
```

See also

meshDeform (modifier), face, vertexList (mesh deform), vertices

far (fog)

Usage

```
member(whichCastmember).camera(whichCamera).fog.far
sprite(whichSprite).camera{(index)}.fog.far
```

Description

3D camera property; indicates the distance from the camera, in world units, where the fog reaches its maximum density when the camera's fog.enabled property is set to TRUE.

The default value for this property is 1000.

Example

The following statement sets the far property of the fog of the camera named BayView to 5000. If the fog's enabled property is set to TRUE, the fog will be densest 5000 world units in front of the camera.

```
-- Lingo syntax
member("MyYard").camera("BayView").fog.far = 5000

// JavaScript syntax
member("MyYard").getProp("camera",1).getPropRef("fog").far = 5000;
```

See also

fog, near (fog)

fieldOfView

Usage

```
-- Lingo syntax
spriteObjRef.fieldOfView

// JavaScript syntax
spriteObjRef.fieldOfView;
```

Description

QTVR sprite property; gives the specified sprite's current field of view in degrees.

This property can be tested and set.

Example

This statement sets the `fieldOfView` property of camera 1 to 90:

```
-- Lingo syntax
member("3d world").camera[1].fieldOfView = 90

// JavaScript syntax
member("3d world").getProp("camera",1).fieldOfView = 90;
```

fieldOfView (3D)

Usage

```
member(whichCastmember).camera(whichCamera).fieldOfView
sprite(whichSprite).camera{(index)}.fieldOfView
```

Description

3D camera property; indicates the angle formed by two rays: one drawn from the camera to the top of the projection plane, and the other drawn from the camera to the bottom of the projection plane.

The images of the models in the 3D world are mapped onto the projection plane, which is positioned in front of the camera like a screen in front of a movie projector. The projection plane is what you see in the 3D sprite. The top and bottom of the projection plane are defined by the `fieldOfView` property. Note, however, that the sprite is not resized as the value of the `fieldOfView` property changes. Instead, the image of the projection plane is scaled to fit the rect of the sprite.

The value of this property is meaningful only when the value of the camera's `projection` property is set to `#perspective`. When the `projection` property is set to `#orthographic`, use the camera's `orthoHeight` property to define the top and bottom of the projection plane.

The default setting for this property is 30.0.

Example

This statement sets the `fieldOfView` property of camera 1 to 90:

```
member("3d world").camera[1].fieldOfView = 90
```

See also

orthoHeight

fileFreeSize

Usage

```
-- Lingo syntax
_movie.fileFreeSize

// JavaScript syntax
_movie.fileFreeSize;
```

Description

Movie property; returns the number of unused bytes in the current movie caused by changes to the cast libraries and cast members within a movie. Read-only.

The `Save` and `Compact` and `Save As` commands rewrite the file to delete this free space.

When the movie has no unused space, `fileFreeSize` returns 0.

Example

This statement displays the number of unused bytes that are in the current movie:

```
-- Lingo syntax
put(_movie.fileFreeSize)

// JavaScript syntax
put(_movie.fileFreeSize);
```

See also

Movie

fileName (Cast)

Usage

```
-- Lingo syntax
castObjRef.fileName

// JavaScript syntax
castObjRef.fileName;
```

Description

Cast library property; returns or sets the filename of a cast library. Read-only for internal cast libraries, read/write for external cast libraries.

For external cast libraries, `fileName` returns the cast's full pathname and filename.

For internal cast libraries, `fileName` returns a value depending on which internal cast library is specified.

- If the first internal cast library is specified, `fileName` returns the name of the movie.

- If any other internal cast library is specified, `fileName` returns an empty string.

This property accepts URLs as references. However, to use a cast library from the Internet and minimize download time, use the `downloadNetThing()` or `preloadNetThing()` methods to download the cast's file to a local disk, and then set `fileName` to the file on the disk.

If a movie sets the filename of an external cast, do no use the Duplicate cast members for Faster Loading option in the Project Options dialog box.

Example

This statement displays the pathname and filename of the Buttons external cast in the Message window:

```
-- Lingo syntax
trace(castLib("Buttons").fileName)

// JavaScript syntax
trace(castLib("Buttons").fileName);
```

This statement sets the filename of the Buttons external cast to Content.cst:

```
-- Lingo syntax
castLib("Buttons").fileName = _movie.path & "Content.cst"

// JavaScript syntax
castLib("Buttons").fileName = _movie.path + "Content.cst";
```

The movie then uses the external cast file Content.cst as the Buttons cast.

These statements download an external cast from a URL to the Director application folder and then make that file the external cast named Cast of Thousands:

```
-- Lingo syntax
downloadNetThing("http://wwwcbDeMille.com/Thousands.cst", _player.applicationPath &
"Thousands.cst")
castLib("Cast of Thousands").fileName = _player.applicationPath & "Thousands.cst"

// JavaScript syntax
downloadNetThing("http://wwwcbDeMille.com/Thousands.cst", _player.applicationPath +
"Thousands.cst");
castLib("Cast of Thousands").fileName = _player.applicationPath + "Thousands.cst";
```

See also

Cast Library, downloadNetThing, preloadNetThing()

fileName (Member)

Usage

```
-- Lingo syntax
memberObjRef.fileName

// JavaScript syntax
memberObjRef.fileName;
```

Description

Member property; refers to the name of the file assigned to a linked cast member. Read/write.

This property is useful for switching the external linked file assigned to a cast member while a movie plays, similar to the way you can switch cast members. When the linked file is in a different folder than the movie, you must include the file's pathname.

You can also make unlinked media linked by setting the filename of those types of members that support linked media.

This property also accepts URLs as a reference. However, to use a file from a URL and minimize download time, use the `downloadNetThing()` or `preloadNetThing()` methods to download the file to a local disk first and then set the `fileName` property to the file on the local disk.

After the filename is set, Director uses that file the next time the cast member is used.

Example

This statement links the QuickTime movie "ChairAnimation" to cast member 40:

```
-- Lingo syntax
member(40).fileName = "ChairAnimation"

// JavaScript syntax
member(40).fileName = "ChairAnimation";
```

These statements download an external file from a URL to the Director application folder and make that file the media for the sound cast member Norma Desmond Speaks:

```
-- Lingo syntax
downloadNetThing("http://wwwcbDeMille.com/Talkies.AIF", _player.applicationPath &
"Talkies.AIF")
member("Norma Desmond Speaks").fileName = _player.applicationPath & "Talkies.AIF"

// JavaScript syntax
downloadNetThing("http://wwwcbDeMille.com/Talkies.AIF", _player.applicationPath +
"Talkies.AIF");
member("Norma Desmond Speaks").fileName = _player.applicationPath + "Talkies.AIF";
```

See also

`downloadNetThing`, `Member`, `preloadNetThing()`

fileName (Window)

Usage

```
-- Lingo syntax
windowObjRef.fileName

// JavaScript syntax
windowObjRef.fileName;
```

Description

Window property; refers to the filename of the movie assigned to a window. Read/write.

When the linked file is in a different folder than the movie, you must include the file's pathname.

To be able to play the movie in a window, you must set the `fileName` property to the movie's filename.

The `fileName` property accepts URLs as a reference. However, to use a movie file from a URL and minimize the download time, use the `downloadNetThing()` or `preloadNetThing()` methods to download the movie file to a local disk first and then set `fileName` property to the file on the local disk.

Example

This statement assigns the file named Control Panel to the window named Tool Box:

```
-- Lingo syntax
window("Tool Box").fileName = "Control Panel"

// JavaScript syntax
window("Tool Box").fileName = "Control Panel";
```

This statement displays the filename of the file assigned to the window named Navigator:

```
-- Lingo syntax
trace(window("Navigator").fileName)

// JavaScript syntax
trace(window("Navigator").fileName);
```

These statements download a movie file from a URL to the Director application folder and then assign that file to the window named My Close Up:

```
-- Lingo syntax
downLoadNetThing("http://www.cbDeMille.com/Finale.DIR", _player.applicationPath &
"Finale.DIR")
window("My Close Up").fileName = _player.applicationPath & "Finale.DIR"

// JavaScript syntax
downLoadNetThing("http://www.cbDeMille.com/Finale.DIR", _player.applicationPath +
"Finale.DIR");
window("My Close Up").fileName = _player.applicationPath + "Finale.DIR";
```

See also

downloadNetThing, preloadNetThing(), Window

fileSize

Usage

```
-- Lingo syntax
_movie.fileSize

// JavaScript syntax
_movie.fileSize;
```

Description

Movie property; returns the number of bytes in the current movie saved on disk. Read-only.

This is the same number returned when selecting File Properties in Windows or Get Info in the Mac Finder.

Example

This statement displays the number of bytes in the current movie:

```
-- Lingo syntax
put(_movie.fileSize)
```

```
// JavaScript syntax
put(_movie.fileSize);
```

See also

Movie

fileVersion

Usage

```
-- Lingo syntax
_movie.fileVersion

// JavaScript syntax
_movie.fileVersion;
```

Description

Movie property; indicates the version, as a string, of Director in which the movie was last saved. Read-only.

Example

This statement displays the version of Director that last saved the current movie:

```
-- Lingo syntax
put(_movie.fileVersion)

// JavaScript syntax
put(_movie.fileVersion);
```

See also

Movie

fillColor

Usage

```
-- Lingo syntax
memberObjRef.fillColor

// JavaScript syntax
memberObjRef.fillColor;
```

Description

Vector shape cast member property; the color of the shape's fill specified as an RGB value.

It's possible to use `fillColor` when the `fillMode` property of the shape is set to `#solid` or `#gradient`, but not if it is set to `#none`. If the `fillMode` is `#gradient`, `fillColor` specifies the starting color for the gradient. The ending color is specified with `endColor`.

This property can be tested and set.

To see an example of `fillColor` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement sets the fill color of the member Archie to a new RGB value:

```
-- Lingo syntax
member("Archie").fillColor = color( 24, 15, 153)

// JavaScript syntax
member("Archie").fillColor = color( 24, 15, 153);
```

See also

endColor, fillMode

fillCycles

Usage

```
-- Lingo syntax
memberObjRef.fillCycles

// JavaScript syntax
memberObjRef.fillCycles;
```

Description

Vector shape cast member property; the number of fill cycles in a gradient vector shape's fill, as specified by an integer value from 1 to 7.

This property is valid only when the `fillMode` property of the shape is set to `#gradient`.

This property can be tested and set.

To see an example of `fillCycles` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement sets the fillCycles of member Archie to 3:

```
-- Lingo syntax
member("Archie").fillCycles = 3

// JavaScript syntax
member("Archie").fillCycles = 3;
```

See also

endColor, fillColor, fillMode

fillDirection

Usage

```
-- Lingo syntax
memberObjRef.fillDirection

// JavaScript syntax
memberObjRef.fillDirection;
```

Description

Vector shape cast member property; specifies the amount in degrees to rotate the fill of the shape.

This property is only valid when the `fillMode` property of the shape is set to `#gradient`.

This property can be tested and set.

To see an example of `fillDirection` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

Following behavior changes the fillMode of the vector shape to 'gradient' and genrates a simple animation by continuosusly modifying the 'fillDirection' of the vector shape

```
-- Lingo syntax
on beginSprite(me)
    member("VectorShape").fillMode = #gradient
end

on exitFrame(me)
    member("VectorShape").fillDirection = (member("VectorShape").fillDirection + 10 ) mod \
360
end

// JavaScript syntax
function beginSprite(me)
{
    member("VectorShape").fillMode = symbol("gradient");
}

function exitFrame(me)
{
    member("VectorShape").fillDirection = (member("VectorShape").fillDirection + 10 ) % 360;
}
```

See also

`fillMode`

filled

Usage

```
member(whichCastMember).filled
the filled of member whichCastMember
```

Description

Shape cast member property; indicates whether the specified cast member is filled with a pattern (`TRUE`) or not (`FALSE`).

Example

The following statements make the shape cast member Target Area a filled shape and assign it the pattern numbered 1, which is a solid color:

```
-- Lingo syntax
member("Target Area").filled = TRUE
member("Target Area").pattern = 1
```

```
// Java Script
member("Target Area").filled = true;
member("Target Area").pattern = 1;
```

See also

fillColor, fillMode

fillMode

Usage

```
-- Lingo syntax
memberObjRef.fillMode

// JavaScript syntax
memberObjRef.fillMode;
```

Description

Vector shape cast member property; indicates the fill method for the shape, using the following possible values:

- #none—The shape is transparent
- #solid—The shape uses a single fill color
- #gradient—The shape uses a gradient between two colors

This property can be tested and set when the shape is closed; open shapes have no fill.

To see an example of fillMode used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement sets the fillMode of member Archie to gradient:

```
-- Lingo syntax
member("Archie").fillMode = #gradient

// JavaScript syntax
member("Archie").fillMode = symbol("gradient");
```

See also

endColor, fillColor

fillOffset

Usage

```
-- Lingo syntax
memberObjRef.fillOffset

// JavaScript syntax
memberObjRef.fillOffset;
```

Description

Vector shape cast member property; specifies the horizontal and vertical amount in pixels (within the `defaultRect` space) to offset the fill of the shape.

This property is only valid when the `fillMode` property of the shape is set to `#gradient`, but can be both tested and set.

To see an example of `fillOffset` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement changes the fill offset of the vector shape cast member miette to a horizontal offset of 33 pixels and a vertical offset of 27 pixels:

```
-- Lingo syntax
member("miette").fillOffset = point(33, 27)

// JavaScript syntax
member("miette").fillOffset = point(33, 27);
```

See also

`defaultRect`, `fillMode`

fillScale

Usage

```
-- Lingo syntax
memberObjRef.fillScale

// JavaScript syntax
memberObjRef.fillScale;
```

Description

Vector shape cast member property; specifies the amount to scale the fill of the shape. This property is referred to as "spread" in the vector shape window.

This property is only valid when the `fillMode` property of the shape is set to `#gradient`, but can be both tested and set.

To see an example of `fillScale` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement sets the `fillScale` of member Archie to 33:

```
-- Lingo syntax
member("Archie").fillScale = 33.00

// JavaScript syntax
member("Archie").fillScale = 33.00;
```

See also

`fillMode`

filterlist

Usage

```
-- Lingo syntax
spriteObjRef.filterlist
```

```
// JavaScript syntax
spriteObjRef.filterlist;
```

Description

Sprite property; determines whether any Bitmap filter is applied to a sprite. This is a list so bitmap filters can be applied to it by appending to the list.

Note: *You cannot duplicate a filterlist using the duplicate() method.*

Example

The first statement sets the variable named myFilter to the Blur filter. The next line sets the blur filter to the sprite(1).

```
--Lingo syntax
MyFilter=filter(#BlurFilter)
sprite(1).filterlist.append(MyFilter)
```

```
// JavaScript syntax
var MyFilter = filter(symbol("BlurFilter"));
sprite(1).filterlist.append(MyFilter);
```

See also

Bitmap filters in Using Director.

firstIndent

Usage

```
-- Lingo syntax
chunkExpression.firstIndent

// JavaScript syntax
chunkExpression.firstIndent;
```

Description

Text cast member property; contains the number of pixels the first indent in *`chunkExpression`* is offset from the left margin of the *`chunkExpression`*.

The value is an integer: less than 0 indicates a hanging indent, 0 is no indention, and greater than 0 is a normal indention.

This property can be tested and set.

Example

This statement sets the indent of the first line of member Desk to 0 pixels:

```
--Lingo syntax
member("Desk").firstIndent = 0

// JavaScript syntax
member("Desk").firstIndent = 0;
```

See also

leftIndent, rightIndent

fixedLineSpace

Usage

```
-- Lingo syntax
chunkExpression.fixedLineSpace

// JavaScript syntax
chunkExpression.fixedLineSpace;
```

Description

Text cast member property; controls the height of each line in the *chunkExpression* portion of the text cast member.

The value itself is an integer, indicating height in absolute pixels of each line.

The default value is 0, which results in natural height of lines.

Example

This statement sets the height in pixels of each line of member Desk to 24:

```
--Lingo syntax
member("Desk").fixedLineSpace = 24

// JavaScript syntax
member("Desk").fixedLineSpace = 24;
```

fixedRate

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.fixedRate

// JavaScript syntax
memberOrSpriteObjRef.fixedRate;
```

Description

Cast member property and sprite property; controls the frame rate of a Flash movie or animated GIF. The `fixedRate` property can have integer values. The default value is 15.

This property is ignored if the sprite's `playbackMode` property is anything other than `#fixed`.

This property can be tested and set.

Example

The following handler adjusts the frame rate of a Flash movie sprite. As parameters, the handler accepts a sprite reference, an indication of whether to speed up or slow down the Flash movie, and the amount to adjust the speed.

```
-- Lingo syntax
on adjustFixedRate(whichSprite, adjustType, howMuch)
```

```
    case adjustType of
        #faster:
            sprite(whichSprite).fixedRate = sprite(whichSprite).fixedRate + howMuch
        #slower:
            sprite(whichSprite).fixedRate = sprite(whichSprite).fixedRate - howMuch
    end case
end

// JavaScript syntax
function adjustFixedRate(whichSprite, adjustType, howMuch) {
    switch(adjustType) {
        case "faster":
            sprite(whichSprite).fixedRate = sprite(whichSprite).fixedRate + howMuch;
            break;
        case "slower":
            sprite(whichSprite).fixedRate = sprite(whichSprite).fixedRate - howMuch;
            break;
    }
}
```

See also

playBackMode

fixStageSize

Usage

```
-- Lingo syntax
_movie.fixStageSize

// JavaScript syntax
_movie.fixStageSize;
```

Description

Movie property; determines whether the Stage size remains the same when you load a new movie (TRUE, default), or not (FALSE), regardless of the Stage size saved with that movie, or the setting for the centerStage. Read/write.

The fixStageSize property cannot change the Stage size for a movie that is currently playing.

Example

The following statement determines whether the fixStageSize property is turned on. If fixStageSize is FALSE, it sends the playhead to a specified frame.

```
-- Lingo syntax
if (_movie.fixStageSize = FALSE) then
    _movie.go("proper size")
end if

// JavaScript syntax
if (_movie.fixStageSize == false) {
    _movie.go("proper size");
}
```

This statement sets the fixStageSize property to the opposite of its current setting:

```
-- Lingo syntax
_movie.fixStageSize = not(_movie.fixStageSize)
```

```
// JavaScript syntax
_movie.fixStageSize = !(_movie.fixStageSize);
```

See also

centerStage, Movie

flashRect

Usage

```
-- Lingo syntax
memberObjRef.flashRect

// JavaScript syntax
memberObjRef.flashRect;
```

Description

Cast member property; indicates the size of a Flash movie or vector shape cast member as it was originally created. The property values are indicated as a Director rectangle: for example, rect(0,0,32,32).

For linked Flash cast members, the `FlashRect` member property returns a valid value only when the cast member's header has finished loading into memory.

This property can be tested but not set.

Example

This sprite script resizes a Flash movie sprite so that it is equal to the original size of its Flash movie cast member:

```
-- Lingo syntax
property spriteNum

on beginSprite me
    sprite(spriteNum).rect = sprite(spriteNum).member.FlashRect
end

// JavaScript syntax
function beginSprite() {
    sprite(this.spriteNum).rect = sprite(this.spriteNum).member.FlashRect;
}
```

See also

defaultRect, defaultRectMode, state (Flash, SWA)

flat

Usage

```
member(whichCastmember).shader(whichShader).flat
member(whichCastmember).model(whichModel).shader.flat
member(whichCastmember).model(whichModel).shaderList{[index]}.flat
```

Description

3D `#standard` shader property; indicates whether the mesh should be rendered with flat shading (`TRUE`) or Gouraud shading (`FALSE`).

Flat shading uses one color per face of the mesh. The color used for the face is the color of its first vertex. Flat shading is faster than Gouraud shading.

Gouraud shading assigns a color to each vertex of a face and interpolates the colors across the face in a gradient. Gouraud shading requires more time and calculation, but creates a smoother surface.

The default value for this property is FALSE.

Example

The following statement sets the flat property of the shader named Wall to TRUE. The mesh of a model that uses this shader will be rendered with one color per face.

```
-- Lingo syntax
member("MysteryWorld").shader("Wall").flat = TRUE

// JavaScript syntax
member("MysteryWorld").getProp("shader", 1).flat = true;
```

See also

mesh (property), colors, vertices, generateNormals()

flipH

Usage

```
-- Lingo syntax
spriteObjRef.flipH

// JavaScript syntax
spriteObjRef.flipH;
```

Description

Sprite property; indicates whether a sprite's image has been flipped horizontally on the Stage (TRUE) or not (FALSE). Read-only.

The image itself is flipped around its registration point.

This means any rotation or skew remains constant; only the image data itself is flipped.

Example

This statement displays the flipH of sprite 5:

```
-- Lingo syntax
put(sprite(5).flipH)

// JavaScript syntax
put(sprite(5).flipH);
```

See also

flipV, rotation, skew, Sprite

flipV

Usage

```
-- Lingo syntax
spriteObjRef.flipV

// JavaScript syntax
spriteObjRef.flipV;
```

Description

Sprite property; indicates whether a sprite's image has been flipped vertically on the Stage (`TRUE`) or not (`FALSE`). Read-only.

The image itself is flipped around its registration point.

This means any rotation or skew remains constant; only the image data itself is flipped.

Example

This statement displays the `flipV` of sprite 5:

```
-- Lingo syntax
put(sprite(5).flipV)

// JavaScript syntax
put(sprite(5).flipV);
```

See also

`flipH`, `rotation`, `skew`, `Sprite`

floatPrecision

Usage

```
the floatPrecision
```

Description

Movie property; rounds off the display of floating-point numbers to the number of decimal places specified. The value of `floatPrecision` must be an integer. The maximum value is 15 significant digits; the default value is 4.

The `floatPrecision` property determines only the number of digits used to display floating-point numbers; it does not change the number of digits used to perform calculations.

- If `floatPrecision` is a number from 1 to 15, floating-point numbers display that number of digits after the decimal point. Trailing zeros remain.
- If `floatPrecision` is zero, floating-point numbers are rounded to the nearest integer. No decimal points appear.
- If `floatPrecision` is a negative number, floating-point numbers are rounded to the absolute value for the number of decimal places. Trailing zeros are dropped.

This property can be tested and set.

Example

This statement rounds off the square root of 3.0 to three decimal places:

```
the floatPrecision = 3
x = sqrt(3.0)
put x
-- 1.732
```

This statement rounds off the square root of 3.0 to eight decimal places:

```
the floatPrecision = 8
put x
-- 1.73205081
```

fog

Usage

```
\-- Lingo Usage
member(whichCastmember).camera(whichCamera).fog.color
sprite(whichSprite).camera{(index)}.fog.color
member(whichCastmember).camera(whichCamera).fog.decayMode
sprite(whichSprite).camera{(index)}.fog.decayMode
member(whichCastmember).camera(whichCamera).fog.enabled
sprite(whichSprite).camera{(index)}.fog.enabled
member(whichCastmember).camera(whichCamera).fog.far
sprite(whichSprite).camera{(index)}.fog.far
member(whichCastmember).camera(whichCamera).fog.near
sprite(whichSprite).camera{(index)}.fog.near
```

```
// JavaScript Usage
member(whichCastmember).getProp("camera", whichCameraIndex).getPropRef("fog").color
sprite(whichSprite).camera.getPropRef("fog").color
member(whichCastmember).getProp("camera", whichCameraIndex).getPropRef("fog").decayMode
sprite(whichSprite).camera.getPropRef("fog").decayMode
member(whichCastmember).getProp("camera", whichCameraIndex).getPropRef("fog").enabled
sprite(whichSprite).camera.getPropRef("fog").enabled
member(whichCastmember).getProp("camera", whichCameraIndex).getPropRef("fog").far
sprite(whichSprite).camera.getPropRef("fog").far
member(whichCastmember).getProp("camera", whichCameraIndex).getPropRef("fog").near
sprite(whichSprite).camera.getPropRef("fog").near
```

Description

3D camera property; fog introduces a coloring and blurring of models that increases with distance from the camera. The effect is similar to real fog, except that it can be any color.

See also

color (fog), decayMode, enabled (fog), far (fog), near (fog)

folder

Usage

```
-- Lingo syntax
dvdObjRef.folder
```

```
// JavaScript syntax
```

```
dvdObjRef.folder;
```

Description

DVD property. Determines the pathname of the folder from which a DVD is playing. Read/write.

The pathname must be a string.

The folder property can be set either in the Property inspector or through scripting. The current implementation has the following requirements:

Windows:

- You must provide video_ts at the end of the file path for the targeted local DVD media. For example, C:\video_ts or C:\myLocalDVDContent\video_ts

Mac:

- The value of the folder property's path must begin with /Volumes/
- Adding video_ts to the path entered for the folder property's value is currently optional. For example, if the DVD video_ts folder is located on the root of the start-up drive, the value for the folder property could be entered in either of the following two ways:
  - /Volumes/Mac HD/myLocalDVDContent/video_ts

  or
  - /Volumes/Mac HD/myLocalDVDContent

To edit the folder property value in the Property inspector:

1 Select the DVD cast member, and then activate the DVD tab of the Property inspector while in list view mode.

2 Under the Playback Properties section, select the folder property's editable value field and then enter the file path for the location of the targeted DVD media.

Use the following examples as a guide for setting the folder property through scripting. These statements set the pathname of the DVD folder property:

Example

Windows:

```
-- Lingo syntax
member(2).folder = "C:\myLocalDVDContent\video_ts"

// JavaScript syntax
member(2).folder = "C:\\myLocalDVDContent\\video_ts";
```

Mac:

```
-- Lingo syntax
member(2).folder = "/Volumes/Mac HD/myLocalDVDContent"

// JavaScript syntax
member(2).folder = "/Volumes/Mac HD/myLocalDVDContent";
```

Note: *If a video_ts folder cannot be found when the first DVD cast member is created, an error alert will appear that says "Unable to locate DVD volume." This alert will only appear once per session. At that point, you can still name any newly created DVD member and then set the folder property to a location that contains a valid video_ts folder.*

Issues with Mac DVD folder pathnames

On Mac computers, the format of the pathname for the folder property should use a forward slash (/) as the path's delimeter, instead of the standard Mac delimiter colon (:). In addition, /volumes/ should be concatenated at the start of the pathname of the DVD folder. For example, if the DVD folder is located on the root of the boot drive, it would look like the following:

```
member (2).folder = "/Volumes/Mac HD/Test_DVD/video_ts"
```

When the _movie.path command is used for retrieving the path of the projector or movie on a Mac, it will contain a colon (:) instead of the forward slash (/). The use of the colon in the DVD folder's pathname will cause an error. As a workaround, developers can use a script to replace the colon characters in the pathname with forward slashes.

See also

DVD

font

Usage

```
-- Lingo syntax
memberObjRef.font

// JavaScript syntax
memberObjRef.font;
```

Description

Text and field cast member property; determines the font used to display the specified cast member and requires that the cast member contain characters, if only a space. The parameter *whichCastMember* can be either a cast member name or number.

The `font` member property can be tested and set.

To see an example of `font` used in a completed movie, see the Text movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement sets the variable named `oldFont` to the current `font` setting for the field cast member Rokujo Speaks:

```
-- Lingo syntax
oldFont = member("Rokujo Speaks").font

// JavaScript syntax
var oldFont = member("Rokujo Speaks").font;
```

See also

text, alignment, fontSize, fontStyle, lineHeight

fontSize

Usage

```
-- Lingo syntax
memberObjRef.fontSize
```

```
// JavaScript syntax
memberObjRef.fontSize;
```

Description

Field cast member property; determines the size of the font used to display the specified field cast member and requires that the cast member contain characters, if only a space. The parameter *whichCastMember* can be either a cast member name or number.

This property can be tested and set. When tested, it returns the height of the first line in the field. When set, it affects every line in the field.

To see an example of `fontSize` used in a completed movie, see the Text movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement sets the variable named `oldSize` to the current `fontSize` of member setting for the field cast member Rokujo Speaks:

```
--Lingo syntax
oldSize = member("Rokujo Speaks").fontSize

// JavaScript syntax
var oldSize = member("Rokujo Speaks").fontSize;
```

This statement sets the third line of the text cast member myMenu to 24 points:

```
member("myMenu").fontSize = 12

// JavaScript syntax
member("myMenu").fontSize = 12;
```

See also

text, alignment, font, fontStyle, lineHeight

fontStyle

Usage

```
-- Lingo syntax
memberObjRef.fontStyle
memberObjRef.char[whichChar].fontStyle
memberObjRef.line[whichLine].fontStyle
memberObjRef.word[whichWord].fontStyle

// JavaScript syntax
memberObjRef.fontStyle;
memberObjRef.getPropRef("char", whichChar).fontStyle
memberObjRef.getPropRef("line", whichLine).fontStyle;
memberObjRef.getPropRef("word", whichWord).fontStyle;
```

Description

Cast member property; determines the styles applied to the font used to display the specified field cast member, character, line, word, or other chunk expression and requires that the field cast member contain characters, if only a space.

The value of the property is a string of styles delimited by commas. Lingo uses a font that is a combination of the styles in the string. The available styles are plain, bold, italic, underline, shadow, outline, and extended; on the Mac, condensed also is available.

Use the style plain to remove all currently applied styles. The parameter *whichCastMember* can be either a cast member name or number.

This property can be tested and set.

To see an example of `fontStyle` used in a completed movie, see the Text movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement sets the variable named `oldStyle` to the current `fontStyle` setting for the field cast member Rokujo Speaks:

```
--Lingo syntax
oldStyle = member("Rokujo Speaks").fontStyle

// JavaScript syntax
var oldStyle = member("Rokujo Speaks").fontStyle;
```

This statement sets the `fontStyle` member property for the field cast member Poem to bold italic:

```
--Lingo syntax
member("Poem").fontStyle = "bold, italic"

// JavaScript syntax
member("Poem").fontStyle = "bold, italic";
```

This statement sets the `fontStyle` property of the third word of the cast member Son's Names to italic:

```
--Lingo syntax
member("Son's Names").word[3].fontStyle = "italic"

// JavaScript syntax
member("Son's Names").getPropRef("word", 3).fontStyle = "italic";
```

This statement sets the `fontStyle` property of the text cast member to bold, or bold and italic:

```
--Lingo syntax
member("text").fontStyle=[#bold]
or
member("text").fontStyle=[#bold, #italic]

// JavaScript syntax
member("text").fontStyle = list(symbol("bold"));
```

See also

`text`, `alignment`, `fontSize`, `font`, `lineHeight`

foreColor

Usage

```
-- Lingo syntax
spriteObjRef.foreColor

// JavaScript syntax
```

```
spriteObjRef.foreColor;
```

Description

Sprite property; returns or sets the foreground color of a sprite. Read/write.

It is not recommended to apply this property to bitmap cast members deeper than 1-bit, as the results are difficult to predict.

It is recommended that the newer `color` property be used instead of the `foreColor` property.

Example

The following statement sets the variable `oldColor` to the foreground color of sprite 5:

```
-- Lingo syntax
oldColor = sprite(5).foreColor

// JavaScript syntax
var oldColor = sprite(5).foreColor;
```

The following statement makes 36 the number for the foreground color of a random sprite from sprites 11 to 13:

```
-- Lingo syntax
sprite(10 + random(3)).foreColor = 36

// JavaScript syntax
sprite(10 + random(3)).foreColor = 36;
```

See also

backColor, color(), Sprite

frame

Usage

```
-- Lingo syntax
_movie.frame

// JavaScript syntax
_movie.frame;
```

Description

Movie property; returns the number of the current frame of the movie. Read-only.

Example

This statement sends the playhead to the frame before the current frame:

```
-- Lingo syntax
_movie.go(_movie.frame - 1)

// JavaScript syntax
_movie.go(_movie.frame - 1);
```

See also

go(), Movie

frameCount

Usage

```
-- Lingo syntax
memberObjRef.frameCount

// JavaScript syntax
memberObjRef.frameCount;
```

Description

Flash cast member property; indicates the number of frames in the Flash movie cast member. The `frameCount` member property can have integer values.

This property can be tested but not set.

Example

This sprite script displays, in the Message window, the channel number and the number of frames in a Flash movie:

```
-- Lingo syntax
property spriteNum

on beginSprite me
    put(""The Flash movie in channel"" && spriteNum && has"" &&
/sprite(spriteNum).member.frameCount && ""frames.""
end

// JavaScript syntax
function beginSprite() {
    trace("The Flash movie in channel " + (this.spriteNum) + " has " +
sprite(this.spriteNum).member.frameCount + " frames.");
}
```

frameLabel

Usage

```
-- Lingo syntax
_movie.frameLabel

// JavaScript syntax
_movie.frameLabel;
```

Description

Movie property; identifies the label assigned to the current frame. Read/write during a Score recording session only.

When the current frame has no label, the value of the `frameLabel` property is 0.

Example

The following statement checks the label of the current frame. In this case, the current `frameLabel` value is Start:

```
-- Lingo syntax
put(_movie.frameLabel)

// JavaScript syntax
put(_movie.frameLabel);
```

See also

labelList, Movie

framePalette

Usage

```
-- Lingo syntax
_movie.framePalette

// JavaScript syntax
_movie.framePalette;
```

Description

Movie property; identifies the cast member number of the palette used in the current frame, which is either the current palette or the palette set in the current frame. Read/write during a Score recording session only.

When you want exact control over colors, use Shockwave Player.

Example

The following statement checks the palette used in the current frame. In this case, the palette is cast member 45.

```
-- Lingo syntax
put(_movie.framePalette)

// JavaScript syntax
put(_movie.framePalette);
```

This statement makes palette cast member 45 the palette for the current frame:

```
-- Lingo syntax
_movie.framePalette = 45

// JavaScript syntax
_movie.framePalette = 45;
```

See also

Movie

frameRate

Usage

```
-- Lingo syntax
memberObjRef.frameRate

// JavaScript syntax
memberObjRef.frameRate;
```

Description

Cast member property; specifies the playback frame rate for the specified digital video, or Flash movie cast member.

The possible values for the frame rate of a digital video member correspond to the radio buttons for selecting digital video playback options.

- When the `frameRate` member property is between 1 and 255, the digital video movie plays every frame at that frame rate. The `frameRate` member property cannot be greater than 255.
- When the `frameRate` member property is set to -1 or 0, the digital video movie plays every frame at its normal rate. This allows the video to sync to its soundtrack. When the frameRate is set to any value other than -1 or 0, the digital video soundtrack will not play.
- When the `frameRate` member property is set to -2, the digital video movie plays every frame as fast as possible.

For Flash movie cast members, the property indicates the frame rate of the movie created in Flash.

This property can be tested but not set.

Example

This statement sets the frame rate of the QuickTime digital video cast member Rotating Chair to 30 frames per second:

```
-- Lingo syntax
member("Rotating Chair").frameRate = 30

// JavaScript syntax
member("Rotating Chair").frameRate = 30;
```

This statement instructs the QuickTime digital video cast member Rotating Chair to play every frame as fast as possible:

```
-- Lingo syntax
member("Rotating Chair").frameRate = -2

// JavaScript syntax
member("Rotating Chair").frameRate = -2;
```

The following sprite script checks to see if the sprite's cast member was originally created in Flash with a frame rate of less than 15 frames per second. If the movie's frame rate is slower than 15 frames per second, the script sets the playBackMode property for the sprite so it can be set to another rate. The script then sets the sprite's fixedRate property to 15 frames per second.

```
-- Lingo syntax
property spriteNum

on beginSprite me
    if sprite(spriteNum).member.frameRate < 15 then
        sprite(spriteNum).playBackMode = #fixed
        sprite(spriteNum).fixedRate = 15
    end if
end

// JavaScript syntax
function beginSprite () {
    var fr = sprite(this.spriteNum).member.frameRate;
    if (fr < 15) {
        sprite(this.spriteNum).playBackMode = symbol("fixed");
        sprite(this.spriteNum).fixedRate = 15;
    }
}
```

See also

fixedRate, playRate, currentTime (QuickTime, AVI), playBackMode

frameRate (DVD)

Usage

```
-- Lingo syntax
dvdObjRef.frameRate

// JavaScript syntax
dvdObjRef.frameRate;
```

Description

DVD property. Returns the frameRate value of the DVD. Read-only.

The frameRate value is returned as one of the following floating-point numbers:

Float	Description
0.0	The frameRate value could not be determined either because it is not in the title domain or because the title is not a one sequential video title.
25.0	The DVD is authored to play at 25 frames per second.
30.0	The DVD is authored to play at 30 frames per second.
29.97	The DVD is authored to play at 29.97 frames per second.

See also

DVD

frameScript

Usage

```
-- Lingo syntax
_movie.frameScript

// JavaScript syntax
_movie.frameScript;
```

Description

Movie property; contains the unique cast member number of the frame script assigned to the current frame. Read/write during a Score recording session only.

During a Score generation session, you can also assign a frame script to the current frame by setting the `frameScript` property.

If there is no frame script assigned to the current frame, this property returns 0.

Example

The following statement displays the number of the script assigned to the current frame. In this case, the script number is 25.

```
-- Lingo syntax
put(_movie.frameScript)

// JavaScript syntax
put(_movie.frameScript);
```

This statement makes the script cast member Button responses the frame script for the current frame:

```
-- Lingo syntax
_movie.frameScript = member("Button responses")

// JavaScript syntax
_movie.frameScript = member("Button responses");
```

See also

Movie

frameSound1

Usage

```
-- Lingo syntax
_movie.frameSound1

// JavaScript syntax
_movie.frameSound1;
```

Description

Movie property; determines the number of the cast member assigned to the first sound channel in the current frame. Read/write.

This property can also be set during a Score recording session.

Example

As part of a Score recording session, this statement assigns the sound cast member Jazz to the first sound channel:

```
-- Lingo syntax
_movie.frameSound1 = member("Jazz").number

// JavaScript syntax
_movie.frameSound1 = member("Jazz").number;
```

See also

frameSound2, Movie

frameSound2

Usage

```
-- Lingo syntax
_movie.frameSound2

// JavaScript syntax
_movie.frameSound2;
```

Description

Movie property; determines the number of the cast member assigned to the second sound channel in the current frame. Read/write.

This property can also be set during a Score recording session.

Example

As part of a Score recording session, this statement assigns the sound cast member Jazz to the second sound channel:

```
-- Lingo syntax
_movie.frameSound2 = member("Jazz").number

// JavaScript syntax
_movie.frameSound2 = member("Jazz").number;
```

See also

frameSound1, Movie

frameTempo

Usage

```
-- Lingo syntax
_movie.frameTempo

// JavaScript syntax
_movie.frameTempo;
```

Description

Movie property; indicates the tempo assigned to the current frame. Read/write during a Score recording session only.

Example

The following statement checks the tempo used in the current frame. In this case, the tempo is 15 frames per second.

```
-- Lingo syntax
put(_movie.frameTempo)

// JavaScript syntax
put(_movie.frameTempo);
```

See also

Movie, puppetTempo()

frameTransition

Usage

```
-- Lingo syntax
_movie.frameTransition

// JavaScript syntax
_movie.frameTransition;
```

Description

Movie property; specifies the number of the transition cast member assigned to the current frame. Read/write only during a Score recording session to specify transitions.

Example

When used in a Score recording session, this statement makes the cast member Fog the transition for the frame that Lingo is currently recording

```
-- Lingo syntax
_movie.frameTransition = member("Fog")

// JavaScript syntax
_movie.frameTransition = member("Fog");
```

See also

Movie

front

Usage

```
member(whichCastmember).modelResource(whichModelResource).front
```

Description

3D #box model resource property; indicates whether the side of the box intersected by its -Z axis is sealed (TRUE) or open (FALSE).

The default value for this property is TRUE.

Example

This statement sets the front property of the model resource named Crate to FALSE, meaning the front of this box will be open:

```
member("3D World").modelResource("Crate").front = FALSE
```

See also

back, bottom (3D), top (3D), left (3D), right (3D)

frontWindow

Usage

```
-- Lingo syntax
_player.frontWindow

// JavaScript syntax
_player.frontWindow;
```

Description

Player property; indicates which movie in a window (MIAW) is currently frontmost on the screen. Read-only.

When the Stage is frontmost, frontWindow is the Stage. When a media editor or floating palette is frontmost, frontWindow returns VOID (Lingo) or null (JavaScript syntax).

Example

This statement determines whether the window "Music" is currently the frontmost window and, if it is, brings the window "Try This" to the front:

```
-- Lingo syntax
if (_player.frontWindow = "Music") then
    window("Try This").moveToFront()
end if

// JavaScript syntax
if (_player.frontWindow == "Music") {
    window("Try This").moveToFront();
}
```

See also

Player

fullScreen

Usage

```
-- Lingo syntax
dvdObjRef.fullScreen

// JavaScript syntax
dvdObjRef.fullScreen;
```

Description

DVD property; specifies whether the DVD should play back in full screen mode. Read/write.

Pressing the escape key returns display mode to non-fullscreen and sets the property to false.

Currently unsupported on Mac.

Example

This statement makes the dvd play in full screen mode

```
-- Lingo syntax
member("DVDMember").fullScreen = TRUE

// Java Script
member("DVDMember").fullScreen = true;
```

See also

DVD

getBoneID

Usage

```
memberReference.modelResource.getBoneID("boneName")
```

Description

3D model resource property; returns the index number of the bone named *boneName* in the model resource. This property returns 0 if no bone by that name can be found.

Example

This statement returns an ID number for the bone ShinL:

```
put member("ParkScene").modelResource("LittleKid").getBoneId("ShinL")
-- 40
```

See also

bone

globals

Usage

```
the globals
```

Description

System property; this property contains a special property list of all current global variables with a value other than VOID. Each global variable is a property in the list, with the associated paired value.

You can use the following list operations on globals:

- count()—Returns the number of entries in the list.
- getPropAt(n)—Returns the name of the *n*th entry.
- getProp(x)—Returns the value of an entry with the specified name.
- getAProp(x)—Returns the value of an entry with the specified name.

Note: *The globals property automatically contains the property #version, which is the version of Director running. This means there will always be at least one entry in the list, even if no global variables have been declared yet.*

This property differs from showGlobals in that the globals can be used in contexts other than the Message window. To display the globals in the Message window, use showGlobals.

See also

showGlobals(), clearGlobals()

glossMap

Usage

```
member(whichCastmember).shader(whichShader).glossMap
member(whichCastmember).model(whichModel).shader.glossMap
member(whichCastmember).model(whichModel).shaderList{[index]}.glossMap
```

Description

3D #standard shader property; specifies the texture to use for gloss mapping.

When you set this property, the following properties are automatically set:

- The fourth texture layer of the shader is set to the texture you specified.
- The value of `textureModeList[4]` is set to `#none`.
- The value of `blendFunctionList[4]` is set to `#multiply`.

Example

This statement sets the texture named Oval as the `glossMap` value for the shader used by the model named GlassBox:

```
-- Lingo syntax
member("3DPlanet").model("GlassBox").shader.glossMap = member("3DPlanet").texture("Oval")

// Java Script
member("House").getPropRef("model", 1).shaderList[1].glossMap =
member("House").getPropRef("texture",1);
```

See also

`blendFunctionList`, `textureModeList`, `region`, `specularLightMap`, `diffuseLightMap`

gravity

Usage

```
member(whichCastmember).modelResource(whichModelResource).gravity
```

Description

3D particle model resource property; when used with a model resource whose type is `#particle`, allows you to get or set the `gravity` property of the resource as a vector.

This property defines the gravity force applied to all particles in each simulation step.

The default value for this property is `vector(0,0,0)`.

Example

In this example, ThermoSystem is a model resource of the type `#particle`. This statement sets the `gravity` property of ThermoSystem to the vector (0, -.1, 0), which pulls the particles of thermoSystem gently down the y axis.

```
-- Lingo syntax
member("Fires").modelResource("ThermoSystem").gravity = vector(0, -.1, 0)

// JavaScript syntax
member("Fires").getProp("modelResource", 1).gravity = vector(0, -.1, 0);
```

See also

`drag`, `wind`

gradientType

Usage

```
-- Lingo syntax
memberObjRef.gradientType

// JavaScript syntax
memberObjRef.gradientType;
```

Description

Vector shape cast member property; specifies the actual gradient used in the cast member's fill.

Possible values are `#linear` or `#radial`. The `gradientType` is only valid when the `fillMode` is set to `#gradient`.

This property can be tested and set.

Example

This handler toggles between linear and radial gradients in cast member "backdrop":

```
-- Lingo syntax
on mouseUp me
    if member("backdrop").gradientType = #radial then
        member("backdrop").gradientType = #linear
    else
        member("backdrop").gradientType = #radial
    end if
end

// JavaScript syntax
function mouseUp() {
    var gt = member("backdrop").gradientType;
    if (gt == "radial") {
        member("backdrop").gradientType = symbol("linear");
    } else {
        member("backdrop").gradientType = symbol("radial");
    }
}
```

See also

`fillMode`

group

Usage

```
member(whichCastmember).group(whichGroup)
member(whichCastmember).group[index]
```

Description

3D element; a node in the 3D world that has a name, transform, parent, and children, but no other properties.

Every 3D cast member has a default group named World that cannot be deleted. The parent hierarchy of all models, lights, cameras, and groups that exist in the 3D world terminates in `group("world")`.

Example

The first line of this example shows the second group from the cast member 3Dobjects. The second line shows the group RBCollection01 from the cast member 3Dobjects.

```
-- Lingo syntax
put member("3Dobjects").group("RBCollection01")
put member("3Dobjects").group[2]

// Javascript
put(member("3Dobjects").getPropRef("group",2));
```

See also

newGroup, deleteGroup, child (3D), parent

height

Usage

```
-- Lingo syntax
imageObjRef.height
memberObjRef.height
spriteObjRef.height

// JavaScript syntax
imageObjRef.height;
memberObjRef.height;
spriteObjRef.height;
```

Description

Image, Member, and Sprite property; for vector shape, Flash, animated GIF, RealMedia, Windows Media, bitmap, and shape cast members, determines the height, in pixels, of the cast member displayed on the Stage. Read-only for cast members and image objects, read/write for sprites.

Example

This statement assigns the height of cast member Headline to the variable `vHeight`:

```
-- Lingo syntax
vHeight = member("Headline").height

// JavaScript syntax
var vHeight = member("Headline").height;
```

This statement sets the height of sprite 10 to 26 pixels:

```
-- Lingo syntax
sprite(10).height = 26

// JavaScript syntax
sprite(10).height = 26;
```

See also

Member, Sprite, width

height (3D)

Usage

```
member(whichCastmember).modelResource(whichModelResource).height
member(whichCastmember).texture(whichTexture).height
```

Description

3D `#box` model resource, `#cylinder` model resource, and texture property; indicates the height of the object.

The height of a `#box` or `#cylinder` model resource is measured in world units and can be tested and set. The default value for this property is 50.

The height of a texture is measured in pixels and can be tested but not set. The height of the texture is rounded from the height of the source of the texture to the nearest power of 2.

Example

This statement sets the height of the model resource named Tower to 225.0 world units:

```
member("3D World").modelResource("Tower").height = 225.0
```

This statement shows that the height of the texture named Marsmap is 512 pixels.

```
put member("scene").texture("Marsmap").height
-- 512
```

See also

length (3D), width (3D)

heightVertices

Usage

```
member(whichCastmember).modelResource(whichModelResource).\heightVertices
```

Description

3D `#box` model resource property; indicates the number of mesh vertices along the height of the box. Increasing this value increases the number of faces, and therefore the fineness, of the mesh.

The height of a box is measured along its Y axis.

Set the `renderStyle` property of a model's shader to `#wire` to see the faces of the mesh of the model's resource. Set the `renderStyle` property to `#point` to see just the vertices of the mesh.

The value of this property must be greater than or equal to 2. The default value is 4.

Example

The following statement sets the `heightVertices` property of the model resource named Tower to 10. Nine polygons will be used to define the geometry of the model resource along its Z axis; therefore, there will be ten vertices.

```
member("3D World").modelResource("Tower").heightVertices = 10
```

See also

height (3D)

highlightPercentage

Usage

```
member(whichCastmember).model(whichModel).toon.highlightPercentage
member(whichCastmember).model(whichModel).shader.highlightPercentage
member(whichCastmember).shader(whichShader).highlightPercentage
```

Description

3D `toon` modifier and `#painter` shader property; indicates the percentage of available colors that are used in the area of the model's surface where light creates highlights.

The range of this property is 0 to 100, and the default value is 50.

The number of colors used by the `toon` modifier and `#painter` shader for a model is determined by the `colorSteps` property of the model's `toon` modifier or `#painter` shader.

Example

This example sets the `highlightPercentage` property of the `toon` modifier in the model Sphere.

```
-- Lingo syntax
member("3Dobjects").model("Sphere01").toon.highlightPercentage = 25

// Javascript
member("3Dobjects").getPropRef("model",2).toon.highlightPercentage = 25;
```

See also

`highlightStrength`, `brightness`

highlightStrength

Usage

```
member(whichCastmember).model(whichModel).toon.highlightStrength
member(whichCastmember).model(whichModel).shader.highlightStrength
member(whichCastmember).shader(whichShader).highlightStrength
```

Description

3D `toon` modifier and `#painter` shader property; indicates the brightness of the area of the model's surface where light creates highlights.

The default value of this property is 1.0.

Example

This example sets the `highlightStrength` property of the `toon` modifier in the model Sphere.

```
-- Lingo syntax
member("3Dobjects").model("Sphere01").toon.highlightStrength = 0.25

// Javascript
member("3Dobjects").getPropRef("model",2).toon.highlightStrength = 0.25;
```

See also

`highlightPercentage`, `brightness`

hilite

Usage

```
-- Lingo syntax
memberObjRef.hilite
```

```
// JavaScript syntax
memberObjRef.hilite;
```

Description

Member property; determines whether a check box or radio button created with the button tool is selected (TRUE) or not (FALSE, default). Read/write.

Example

This statement checks whether the button named Sound on is selected and, if it is, turns sound channel 1 all the way up:

```
-- Lingo syntax
if (member("Sound On").hilite = TRUE) then
    sound(1).volume = 255
end if

// JavaScript syntax
if (member("Sound On").hilite == true) {
    sound(1).volume = 255;
}
```

This statement selects the button cast member powerSwitch by setting the hilite member property for the cast member to TRUE:

```
-- Lingo syntax
member("powerSwitch").hilite = TRUE

// JavaScript syntax
member("powerSwitch").hilite = true;
```

See also

Member

hither

Usage

```
member(whichCastmember).camera(whichCamera).hither
sprite(whichSprite).camera{(index)}.hither
```

Description

3D camera property; indicates the distance in world units from the camera beyond which models are drawn. Objects closer to the camera than hither are not drawn.

The value of this property must be greater than or equal to 1.0 and has a default value of 5.0.

Example

The following statement sets the hither property of camera 1 to 1000. Models closer than 1000 world units from the camera will not be visible.

```
member("SolarSystem").camera[1].hither = 1000
```

See also

yon

hotSpot

Usage

```
-- Lingo syntax
memberObjRef.hotSpot

// JavaScript syntax
memberObjRef.hotSpot;
```

Description

Cursor cast member property; specifies the horizontal and vertical point location of the pixel that represents the hotspot within the animated color cursor cast member `whichCursorCastMember`. Director uses this point to track the cursor's position on the screen (for example, when it returns the values for the Lingo functions `mouseH` and `mouseV`) and to determine where a rollover (signaled by the Lingo message `mouseEnter`) occurs.

The upper left corner of a cursor is point(0,0), which is the default `hotSpot` value. Trying to set a point outside the bounds of the cursor produces an error. For example, setting the hotspot of a 16-by-16-pixel cursor to point(16,16) produces an error (because the starting point is 0,0, not 1,1).

This property can be tested and set.

Example

This handler sets the hotspot of a 32-by-32-pixel cursor (whose cast member number is stored in the variable `cursorNum`) to the middle of the cursor:

```
-- Lingo syntax
on startMovie
    member(cursorNum).hotSpot = point(16,16)
end

// JavaScript syntax
function startMovie() {
    member(cursorNum).hotSpot = point(16,16);
}
```

hotSpotEnterCallback

Usage

```
-- Lingo syntax
spriteObjRef.hotSpotEnterCallback

// JavaScript syntax
spriteObjRef.hotSpotEnterCallback;
```

Description

QuickTime VR sprite property; contains the name of the handler that runs when the cursor enters a QuickTime VR hot spot that is visible on the Stage. The QuickTime VR sprite receives the message first. The message has two arguments: the `me` parameter and the ID of the hot spot that the cursor entered.

To clear the callback, set this property to 0.

To avoid a performance penalty, set a callback property only when necessary.

This property can be tested and set.

Example

This example shows the name of the handler that runs when the cursor enters a QuickTime VR hot spot that is visible on the Stage.

```
-- Lingo syntax
put sprite("multinode").hotSpotEnterCallback

// Javascript
put(sprite("multinode").hotSpotEnterCallback);
```

See also

hotSpotExitCallback, nodeEnterCallback, nodeExitCallback, triggerCallback

hotSpotExitCallback

Usage

```
-- Lingo syntax
spriteObjRef.hotSpotExitCallback

// JavaScript syntax
spriteObjRef.hotSpotExitCallback;
```

Description

QuickTime VR sprite property; contains the name of the handler that runs when the cursor leaves a QuickTime VR hot spot that is visible on the Stage. The QuickTime VR sprite receives the message first. The message has two arguments: the `me` parameter and the ID of the hot spot that the cursor entered.

To clear the callback, set this property to 0.

To avoid a performance penalty, set a callback property only when necessary.

This property can be tested and set.

Example

This example shows the name of the handler that runs when the cursor leaves a QuickTime VR hot spot that is visible on the Stage.

```
-- Lingo syntax
put sprite("multinode").hotSpotExitCallback

// Javascript
put(sprite("multinode").hotSpotExitCallback);
```

See also

hotSpotEnterCallback, nodeEnterCallback, nodeExitCallback, triggerCallback

HTML

Usage

```
-- Lingo syntax
memberObjRef.HTML
```

```
// JavaScript syntax
memberObjRef.HTML;
```

Description

Cast member property; accesses text and tags that control the layout of the text within an HTML-formatted text cast member.

This property can be tested and set.

Example

This statement displays in the message window the HTML formatting information embedded in the text cast member Home Page:

```
--Lingo syntax
put(member("Home Page").HTML)

// JavaScript syntax
trace(member("Home Page").HTML);
```

See also

importFileInto(), RTF

hyperlink

Usage

```
-- Lingo syntax
chunkExpression.hyperlink

// JavaScript syntax
chunkExpression.hyperlink;
```

Description

Text cast member property; returns the hyperlink string for the specified chunk expression in the text cast member.

This property can be both tested and set.

When retrieving this property, the link containing the first character of *chunkExpression* is used.

Hyperlinks may not overlap. Setting a hyperlink over an existing link, even partially over it), replaces the initial link with the new one.

Setting a hyperlink to an empty string removes it.

Example

The following handler creates a hyperlink in the first word of text cast member "MacroLink". The text is linked to Adobe's website.

```
--Lingo syntax
on startMovie
    member("MacroLink").word[1].hyperlink = "http://www.adobe.com"
end

// JavaScript syntax
function startMovie() {
    member("MacroLink").getPropRef("word", 1).hyperlink = "http://www.adobe.com";
```

```
}
```

See also

hyperlinkRange, hyperlinkState

hyperlinkRange

Usage

```
-- Lingo syntax
chunkExpression.hyperlinkRange

// JavaScript syntax
chunkExpression.hyperlinkRange;
```

Description

Text cast member property; returns the range of the hyperlink that contains the first character of the chunk expression.

This property can be tested but not set.

Like `hyperLink` and `hyperLinkState`, the returned range of the link contains the first character of *`chunkExpression`*.

Example

This example displays the range of the hyperlink that contains the first character of the chunk expression.

```
-- Lingo syntax
put member("MyText").hyperlinkRange

// Javascript
put(member("MyText").hyperlinkRange);
```

See also

hyperlink, hyperlinkState

hyperlinks

Usage

```
-- Lingo syntax
chunkExpression.hyperlinks

// JavaScript syntax
chunkExpression.hyperlinks;
```

Description

Text cast member property; returns a linear list containing all the hyperlink ranges for the specified chunk of a text cast member. Each range is given as a linear list with two elements, one for the starting character of the link and one for the ending character.

Example

This statement returns all the links for the text cast member Glossary to the message window:

```
--Lingo syntax
put(member("Glossary").hyperlinks) -- [[3, 8], [10, 16], [41, 54]]

// JavaScript syntax
trace(member("Glossary").hyperlinks); // [[3, 8], [10, 16], [41, 54]]
```

hyperlinkState

Usage

```
-- Lingo syntax
chuckExpression.hyperlinkState

// JavaScript syntax
chuckExpression.hyperlinkState;
```

Description

Text cast member property; contains the current state of the hyperlink. Possible values for the state are: `#normal`, `#active`, and `#visited`.

This property can be tested and set.

Like `hyperLink` and `hyperLinkRange`, the returned range of the link contains the first character of *`chunkExpression`*.

Example

The following handler checks to see if the hyperlink clicked is a web address. If it is, the state of the hyperlink text state is set to `#visited`, and the movie branches to the web address.

```
--Lingo syntax
property spriteNum

on hyperlinkClicked me, data, range
    if data starts "http://" then
        currentMember = sprite(spriteNum).member
        currentMember.word[4].hyperlinkState = #visited
        gotoNetPage(data)
    end if
end

// JavaScript syntax
function hyperlinkClicked(data, range) {
    var st = data.slice(0,7);
    var ht = "http://";
    if (st == ht) {
        currentMember = sprite(spriteNum).member;
        currentMember.getPropRef("word", 4).hyperlinkState = symbol("visited");
        gotoNetPage(data);
    }
}
```

See also

hyperlink, hyperlinkRange

idleHandlerPeriod

Usage

```
-- Lingo syntax
_movie.idleHandlerPeriod

// JavaScript syntax
_movie.idleHandlerPeriod;
```

Description

Movie property; determines the maximum number of ticks that passes until the movie sends an `idle` message. Read/write.

The default value is 1, which tells the movie to send `idle` handler messages no more than 60 times per second.

When the playhead enters a frame, Director starts a timer, repaints the appropriate sprites on the Stage, and issues an `enterFrame` event. Then, if the amount of time set for the tempo has elapsed, Director generates an `exitFrame` event and goes to the next specified frame; if the amount of time set for this frame hasn't elapsed, Director waits until the time runs out and periodically generates an `idle` message. The amount of time between `idle` events is determined by `idleHandlerPeriod`.

Possible settings for `idleHandlerPeriod` are:

- 0—As many idle events as possible
- 1—Up to 60 per second
- 2—Up to 30 per second
- 3—Up to 20 per second
- *n*—Up to 60/*n* per second

The number of `idle` events per frame also depends on the frame rate of the movie and other activity, including whether scripts are executing. If the tempo is 60 frames per second (fps) and the `idleHandlerPeriod` value is 1, one `idle` event per frame occurs. If the tempo is 20 fps, three `idle` events per frame occur. Idle time results when Director doesn't have a current task to perform and cannot generate any events.

In contrast, if the `idleHandlerPeriod` property is set to 0 and the tempo is very low, thousands of `idle` events can be generated.

The default value for this property is 1.

Example

The following statement causes the movie to send an idle message a maximum of once per second:

```
-- Lingo syntax
_movie.idleHandlerPeriod = 60

// JavaScript syntax
_movie.idleHandlerPeriod = 60;
```

See also

`on idle`, `idleLoadMode`, `idleLoadPeriod`, `idleLoadTag`, `idleReadChunkSize`, `Movie`

idleLoadMode

Usage

```
-- Lingo syntax
_movie.idleLoadMode

// JavaScript syntax
_movie.idleLoadMode;
```

Description

Movie property; determines when the `preLoad()` and `preLoadMember()` methods try to load cast members during idle periods. Read/write.

Idle periods can be one of the following values:

- 0—Does not perform idle loading
- 1—Performs idle loading when there is free time between frames
- 2—Performs idle loading during `idle` events
- 3—Performs idle loading as frequently as possible

The `idleLoadMode` property performs no function and works only in conjunction with the `preLoad()` and `preLoadMember()` methods.

Cast members that were loaded using idle loading remain compressed until the movie uses them. When the movie plays back, it may have noticeable pauses while it decompresses the cast members.

Example

This statement causes the movie to try as frequently as possible to load cast members designated for preloading by the `preLoad` and `preLoadMember` commands:

```
-- Lingo syntax
_movie.idleLoadMode = 3

// JavaScript syntax
_movie.idleLoadMode = 3;
```

See also

`on idle`, `Movie`, `preLoad() (Movie)`, `preLoadMember()`

idleLoadPeriod

Usage

```
-- Lingo syntax
_movie.idleLoadPeriod

// JavaScript syntax
_movie.idleLoadPeriod;
```

Description

Movie property; determines the number of ticks that Director waits before trying to load cast members waiting to be loaded. Read/write.

The default value for `idleLoadPeriod` is 0, which instructs Director to service the load queue as frequently as possible.

Example

This statement instructs Director to try loading every 1/2 second (30 ticks) any cast members waiting to be loaded:

```
-- Lingo syntax
_movie.idleLoadPeriod = 30

// JavaScript syntax
_movie.idleLoadPeriod = 30;
```

See also

`on idle`, `Movie`

idleLoadTag

Usage

```
-- Lingo syntax
_movie.idleLoadTag

// JavaScript syntax
_movie.idleLoadTag;
```

Description

Movie property; identifies or tags with a number the cast members that have been queued for loading when the computer is idle. Read/write.

The `idleLoadTag` property is a convenience that identifies the cast members in a group that you want to preload, and can be any number that you choose.

Example

This statement makes the number 10 the idle load tag:

```
-- Lingo syntax
_movie.idleLoadTag = 10

// JavaScript syntax
_movie.idleLoadTag = 10;
```

See also

`on idle`, `Movie`

idleReadChunkSize

Usage

```
-- Lingo syntax
_movie.idleReadChunkSize

// JavaScript syntax
_movie.idleReadChunkSize;
```

Description

Movie property; determines the maximum number of bytes that Director can load when it attempts to load cast members from the load queue. Read/write.

The default value of `idleReadChunkSize` is 32K.

Example

This statement specifies that 500K is the maximum number of bytes that Director can load in one attempt at loading cast members in the load queue:

```
-- Lingo syntax
_movie.idleReadChunkSize = (500 * 1024)

// JavaScript syntax
_movie.idleReadChunkSize = (500 * 1024);
```

See also

`on idle`, `Movie`

image (Image)

Usage

```
-- Lingo syntax
imageObjRef.image

// JavaScript syntax
imageObjRef.image;
```

Description

Image property. Refers to the image object of a bitmap or text cast member, of the Stage, or of a window. Read/write for a cast member's image, read-only for an image of the Stage or a window.

Setting a cast member's `image` property immediately changes the contents of the member. However, when getting the image of a member or window, Director creates a reference to the image of the specified member or window. If you make changes to the windows, the contents of the cast member or window change immediately.

If you plan to make a lot of changes to an item's `image` property, it is faster to copy the item's image property into a new image object using the `duplicate()` method, apply your changes to the new image object, and then set the original item's image to the new image object. For nonbitmap members, it is always faster to use the `duplicate()` method.

Example

This statement puts the image of cast member originalFlower into cast member newFlower:

```
-- Lingo syntax
member("newFlower").image = member("originalFlower").image

// JavaScript syntax
member("newFlower").image = member("originalFlower").image;
```

These statements place a reference to the image of the stage into the variable `myImage` and then put that image into cast member flower:

```
-- Lingo syntax
```

```
myImage = _movie.stage.image
member("flower").image = myImage

// JavaScript syntax
var myImage = _movie.stage.image;
member("flower").image = myImage;
```

See also

copyPixels(), draw(), duplicate() (Image), fill(), image(), setPixel()

image (RealMedia)

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.image

// JavaScript syntax
memberOrSpriteObjRef.image;
```

Description

RealMedia sprite or cast member property; returns a Lingo image object containing the current frame of the RealMedia video stream. You can use this property to map RealVideo® onto a 3D model (see the example below).

Example

This statement copies the current frame of the RealMedia cast member Real to the bitmap cast member Still:

```
-- Lingo syntax
member("Still").image = member("Real").image

// JavaScript syntax
member("Still").image = member("Real").image;
```

image (Window)

Usage

```
-- Lingo syntax
windowObjRef.image

// JavaScript syntax
windowObjRef.image;
```

Description

Window property; refers to the image object of a window. Read-only.

When you get the image of a window, Director creates a reference to the image of the specified window. If you make changes to the image, the contents of the window change immediately.

If you plan to make a lot of changes to the `image` property, it is faster to copy the `image` property into a new image object using the Member object's `duplicate()` method, apply your changes to the new image object, and then set the original item's image to the new image object. For nonbitmap members, it is always faster to use the `duplicate()` method.

Example

These statements place a reference to the image of the Stage into the variable `myImage`, and then put that image into the window named Flower:

```
-- Lingo syntax
myImage = _movie.stage.image
window("Flower").image = myImage

// JavaScript syntax
var myImage = _movie.stage.image;
window("Flower").image = myImage;
```

See also

`duplicate() (Member)`, `Window`

imageCompression

Usage

```
-- Lingo syntax
_movie.imageCompression
memberObjRef.imageCompression

// JavaScript syntax
_movie.imageCompression;
memberObjRef.imageCompression;
```

Description

Movie and bitmap cast member property; indicates the type of compression that Director applies to internal (non-linked) bitmap cast members when saving a movie in Shockwave Player format. Read/write.

Valid values for `imageCompression` include the following:

Value	Meaning
#standard	Use the Director standard internal compression format.
#movieSetting	Use the compression settings of the movie, as stored in the `_movie.imageCompression` property. This is the default value for image formats not restricted to standard compression.
#jpeg	Use JPEG compression. See `imageQuality`.

You normally set this property in the Director Publish Settings dialog box.

Example

This statement displays in the Message window the imageCompression that applies to the currently playing movie:

```
-- Lingo syntax
put(_movie.imageCompression)

// JavaScript syntax
put(_movie.imageCompression);
```

See also

`imageQuality`, `Movie`

imageEnabled

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.imageEnabled

// JavaScript syntax
memberOrSpriteObjRef.imageEnabled;
```

Description

Cast member property and sprite property; controls whether a Flash movie or vector shape's graphics are visible (`TRUE`, default) or invisible (`FALSE`).

This property can be tested and set.

Example

This `beginSprite` script sets up a linked Flash movie sprite to hide its graphics when it first appears on the Stage and begins to stream into memory and saves its sprite number in a global variable called `gStreamingSprite` for use in a frame script later in the Score:

```
-- Lingo syntax
global gStreamingSprite

on beginSprite me
    gStreamingSprite = me.spriteNum
    sprite(gStreamingSprite).imageEnabled = FALSE
end

// JavaScript syntax
function beginSprite() {
    _global.gStreamingSprite = this.spriteNum;
    sprite(_global.gStreamingSprite).imageEnabled = 0;
}
```

In a later frame of the movie, this frame script checks to see if the Flash movie sprite specified by the global variable `gStreamingSprite` has finished streaming into memory. If it has not, the script keeps the playhead looping in the current frame until 100% of the movie has streamed into memory. It then sets the `imageEnabled` property to `TRUE` so that the graphics appear and lets the playhead continue to the next frame in the Score.

```
-- Lingo syntax
global gStreamingSprite

on exitFrame me
    if sprite(gStreamingSprite).member.percentStreamed < 100 then
        _movie.go(_movie.frame)
    else
        sprite(gStreamingSprite).imageEnabled = TRUE
        _movie.updatestage()
    end if
end

// JavaScript syntax
function exitFrame() {
    var stmSp = sprite(_global.gStreamingSprite).member.percentStreamed;
    if (stmSp < 100) {
        _movie.go(_movie.frame);
    } else {
        sprite(_global.gStreamingSprite).imageEnabled = 1;
```

```
        _movie.updatestage();
    }
}
```

imageQuality

Usage

```
-- Lingo syntax
_movie.imageQuality
memberObjRef.imageQuality

// JavaScript syntax
_movie.imageQuality;
memberObjRef.imageQuality;
```

Description

Movie and bitmap cast member property; indicates the level of compression to use when a movie's `imageCompression` property is set to `#jpeg`. Read/write during authoring only.

The range of acceptable values is 0–100. Zero yields the lowest image quality and highest compression; 100 yields the highest image quality and lowest compression.

You can set this property only during authoring and it has no effect until the movie is saved in Shockwave Player format.

Example

This statement displays in the Message window the `imageQuality` that applies to the currently playing movie:

```
-- Lingo syntax
put(_movie.imageQuality)

// JavaScript syntax
put(_movie.imageQuality);
```

See also

`imageCompression`, `Movie`

immovable

Usage

```
member(whichCastmember).model(whichModel).collision.immovable
```

Description

3D `#collision` modifier property; indicates whether a model can be moved as a result of collisions during animations. Specifying `TRUE` makes the model immovable; specifying `FALSE` allows the model to be moved. This property is useful as a way of improving performance during animation, because models that do not move do not need to be checked for collisions by Lingo.

This property has a default value of `FALSE`.

Example

This statement sets the `immovable` property of the `collision` modifier attached to the second model of the cast member named 3Dobjects to `TRUE`.

```
-- Lingo syntax
member("3Dobjects").model[2].collision.immovable = TRUE

// Javascript
member("3Dobjects").getPropRef("model",2).collision.immovable = 1;
```

See also

collision (modifier)

ink

Usage

```
-- Lingo syntax
spriteObjRef.ink

// JavaScript syntax
spriteObjRef.ink;
```

Description

Sprite property; determines the ink effect applied to a sprite. Read/write.

Valid values of `ink` are as follows:

0—Copy	32—Blend
1—Transparent	33—Add pin
2—Reverse	34—Add
3—Ghost	35—Subtract pin
4—Not copy	36—Background transparent
5—Not transparent	37—Lightest
6—Not reverse	38—Subtract
7—Not ghost	39—Darkest
8—Matte	40—Lighten
9—Mask	41—Darken

In the case of 36 (background transparent), you select a sprite in the Score and select a transparency color from the background color box in the Tools window. You can also do this by setting the `backColor` property.

If you set this property within a script while the playhead is not moving, be sure to use the Movie object's `updateStage()` method to redraw the Stage. If you change several sprite properties—or several sprites—use only one `updateStage()` method at the end of all the changes.

Example

This statement changes the variable `currentInk` to the value for the ink effect of sprite (3):

```
-- Lingo syntax
currentInk = sprite(3).ink

// JavaScript syntax
var currentInk = sprite(3).ink;
```

This statement gives sprite (i + 1) a matte ink effect by setting the ink effect of the `sprite` property to 8, which specifies matte ink:

```
-- Lingo syntax
sprite(i + 1).ink = 8

// JavaScript syntax
sprite(i + 1).ink = 8;
```

See also

`backColor`, `Sprite`, `updateStage()`

inker (modifier)

Syntax

```
member(whichCastmember).modelResource(whichModelResource).inker.inkerModifierProperty
modelResourceObjectReference.inker.inkerModifierProperty
```

Description

3D modifier; once you have added the `#inker` modifier to a model resource (using `addModifier`) you can get and set `#inker` modifier properties.

The `#inker` modifier adds silhouettes, creases, and boundary edges to an existing model; the `#inker` properties allow you to control the definition and emphasis of these properties.

When the `#inker` modifier is used in conjunction with the `#toon` modifier, the rendered effect is cumulative and varies depending on which modifier was first applied. The list of modifiers returned by the `modifier` property will list `#inker` or `#toon` (whichever was added first), but not both. The `#inker` modifier can not be used in conjunction with the `#sds` modifier.

The `#inker` modifier has the following properties:

- `lineColor` allows you to get or set the color of lines drawn by the inker.
- `silhouettes` allows you to get or set whether lines are drawn to define the edges along the border of a model, outlining its shape.
- `creases` allows you to get or set whether lines are drawn in creases.
- `creaseAngle` allows you to get or set the sensitivity of crease angle detection for the inker.
- `boundary` allows you to get or set whether lines are drawn around the boundary of the surface.
- `lineOffset` allows you to get or set where lines are drawn relative to the surface being shaded and the camera.
- `useLineOffset` allows you to get or set whether `lineOffset` is on or off.

Note: *For more detailed information about the following properties see the individual property entries.*

Example

This statement adds the `inker` modifier to the second model of the cast member named 3Dobjects. Once you have added the `#inker` modifier to a model resource (using addModifier) you can get and set `#inker` modifier properties.

```
-- Lingo syntax
member("3Dobjects").model[2].addModifier(#inker)

// JavaScript syntax
member("3Dobjects").getPropRef("model",2).addModifier(symbol("inker"));
```

See also

addModifier, modifiers, toon (modifier), shadowPercentage

inlineImeEnabled

Usage

```
-- Lingo syntax
_player.inlineImeEnabled

// JavaScript syntax
_player.inlineImeEnabled;
```

Description

In Director 11, inlineImeEnabled property is alwaysTRUE irrespective of whether it is set to TRUE or FALSE using scripting. This property is retained only for backward compatibility.

Example

This statement sets the `inlineImeEnabled` property of the player to `TRUE`.

```
-- Lingo syntax
_player.inlineImeEnabled=TRUE

// JavaScript syntax
_player.inlineImeEnabled=1;
```

See also

Player

interval

Usage

```
-- Lingo syntax
memberObjRef.interval

// JavaScript syntax
memberObjRef.interval;
```

Description

Cursor cast member property; specifies the interval, in milliseconds (ms), between each frame of the animated color cursor cast member *`whichCursorCastMember`*. The default interval is 100 ms.

The cursor interval is independent of the frame rate set for the movie using the tempo channel or the `puppetTempo` Lingo command.

This property can be tested and set.

Example

In this sprite script, when the animated color cursor stored in the cast member named Butterfly enters the sprite, the interval is set to 50 ms to speed up the animation. When the cursor leaves the sprite, the interval is reset to 100 ms to slow down the animation.

```
-- Lingo syntax
on mouseEnter
    member("Butterfly").interval = 50
end

on mouseLeave
    member("Butterfly").interval = 100
end

// JavaScript syntax
function mouseEnter() {
    member("Butterfly").interval = 50;
}

function mouseLeave() {
    member("Butterfly").interval = 100;
}
```

invertMask

Usage

```
-- Lingo syntax
memberObjRef.invertMask

// JavaScript syntax
memberObjRef.invertMask;
```

Description

QuickTime cast member property; determines whether Director draws QuickTime movies in the white pixels of the movie's mask (`TRUE`) or in the black pixels (`FALSE`, default).

This property can be tested and set.

Example

This handler reverses the current setting of the `invertMask` property of a QuickTime movie named Starburst:

```
-- Lingo syntax
on toggleMask
    member("Starburst").invertMask = not(member("Starburst").invertMask)
end

// JavaScript syntax
function toggleMask() {
    member("Starburst").invertMask = !(member("Starburst").invertMask);
}
```

See also

mask

isVRMovie

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.isVRMovie

// JavaScript syntax
memberOrSpriteObjRef.isVRMovie;
```

Description

QuickTime cast member and sprite property; indicates whether a cast member or sprite is a QuickTime VR movie that has not yet been downloaded (TRUE), or whether the cast member or sprite isn't a QuickTime VR movie (FALSE).

Testing for this property in anything other than an asset whose type is #quickTimeMedia produces an error message.

This property can be tested but not set.

Example

The following handler checks to see if the member of a sprite is a QuickTime movie. If it is, the handler further checks to see if it is a QTVR movie. An alert is posted in any case.

```
-- Lingo syntax
on checkForVR(theSprite)
    if sprite(theSprite).member.type = #quickTimeMedia then
        if sprite(theSprite).isVRMovie then
            _player.alert("This is a QTVR asset.")
        else
            _player.alert("This is not a QTVR asset.")
        end if
    else
        _player.alert("This is not a QuickTime asset.")
    end if
end

// JavaScript syntax
function checkForVR(theSprite) {
    var memType = sprite(theSprite).member.type;
    if (memType == "quickTimeMedia") {
        var isType = sprite(theSprite).isVRMovie;
        if (isType == 1) {
            _player.alert("This is a QTVR asset.");
        } else {
            _player.alert("This is not a QTVR asset.");
        } else {
            _player.alert("This is not a QuickTime asset.");
        }
    }
}
```

itemDelimiter

Usage

```
the itemDelimiter
```

Description

Player property; indicates the special character used to separate items.

You can use the `itemDelimiter` to parse filenames by setting `itemDelimiter` to a backslash (\) in Windows or a colon (:) on the Mac. Restore the `itemDelimiter` character to a comma (,) for normal operation.

This function can be tested and set.

Example

The following handler finds the last component in a Mac pathname. The handler first records the current delimiter and then changes the delimiter to a colon (:). When a colon is the delimiter, Lingo can use `the last item` of to determine the last item in the chunk that makes up a Mac pathname. Before exiting, the delimiter is reset to its original value.

```
on getLastComponent pathName
    save = the itemDelimiter
    the itemDelimiter = ":"
    f = the last item of pathName
    the itemDelimiter = save
    return f
end
```

See also

Player

kerning

Usage

```
-- Lingo syntax
memberObjRef.kerning

// JavaScript syntax
memberObjRef.kerning;
```

Description

Text cast member property; this property specifies whether the text is automatically kerned when the contents of the text cast member are changed.

When set to `TRUE`, kerning is automatic; when set to `FALSE`, kerning is not done.

This property defaults to `TRUE`.

Example

This statement sets the `kerning` property of the Text cast member to `TRUE`.

```
-- Lingo syntax
member("MyText").kerning=TRUE

// JavaScript syntax
```

```
member("MyText").kerning=1;
```

See also

kerningThreshold

kerningThreshold

Usage

```
-- Lingo syntax
memberObjRef.kerningThreshold

// JavaScript syntax
memberObjRef.kerningThreshold;
```

Description

Text cast member property; this setting controls the size at which automatic kerning takes place in a text cast member. This has an effect only when the kerning property of the text cast member is set to TRUE.

The setting itself is an integer indicating the font point size at which kerning takes place.

This property defaults to 14 points.

Example

This statement sets the fontsize of `kerningThreshold` property of the Text cast member to 14.

```
-- Lingo syntax
member("MyText").kerningThreshold=14

// JavaScript syntax
member("MyText").kerningThreshold=14;
```

See also

kerning

key

Usage

```
-- Lingo syntax
_key.key

// JavaScript syntax
_key.key;
```

Description

Key property; returns the value of the last key that was pressed. Read-only.

The returned value is the American National Standards Institute (ANSI) value that is assigned to the key, not the numerical value.

You can use `key` in handlers that perform certain actions when the user presses specific keys as shortcuts and other forms of interactivity. When used in a primary event handler, the actions you specify are the first to be executed.

Note: *The value of* `key` *isn't updated if the user presses a key while Lingo or JavaScript syntax is in a loop.*

Use the sample movie Keyboard Lingo to test which characters correspond to different keys on different keyboards.

Example

The following statements cause the movie to return to the main menu marker when the user presses the q key. Because the `keyDownScript` property is set to `checkKey`, the `on prepareMovie` handler makes the on `checkKey` handler the first event handler executed when a key is pressed. The `on checkKey` handler checks whether the q key is pressed and if it is, navigates to the main menu marker.

```
-- Lingo syntax
on prepareMovie
    keyDownScript = "checkKey"
end

on checkKey
    if (_key.key = "q") then _movie.go("Main Menu")
    end if
end

// JavaScript syntax
function prepareMovie() {
    keyDownScript = checkKey();
}

function checkKey() {
    if (_key.key == "q") {
         _movie.go("Main Menu");
    }
}
```

This on `keyDown` handler checks whether the last key pressed is the z key and if it is, calls the on addNumbers handler:

```
-- Lingo syntax
on keyDown
    if (_key.key = "z") then addNumbers
end

// JavaScript syntax
function keyDown() {
    if (_key.key == "z") {
        addNumbers();
    }
}
```

See also

`commandDown`, `Key`

keyboardFocusSprite

Usage

```
-- Lingo syntax
_movie.keyboardFocusSprite

// JavaScript syntax
```

```
_movie.keyboardFocusSprite;
```

Description

Movie property; lets the user set the focus for keyboard input (without controlling the cursor's insertion point) on a particular text sprite currently on the screen. Read/write.

This is the equivalent to using the Tab key when the `autoTab` property of the cast member is selected.

Setting `keyboardFocusSprite` to -1 returns keyboard focus control to the Score, and setting it to 0 disables keyboard entry into any editable sprite.

Example

This statement disables keyboard entry into any editable sprite.

```
-- Lingo syntax
_movie.keyboardFocusSprite=0

// JavaScript syntax
_movie.keyboardFocusSprite=0;
```

See also

Movie

keyCode

Usage

```
-- Lingo syntax
_key.keyCode

// JavaScript syntax
_key.keyCode;
```

Description

Key property; returns the numerical code for the last key pressed. Read-only.

The returned value is the key's numerical value, not the American National Standards Institute (ANSI) value.

You can use `keyCode` to detect when the user has pressed an arrow or function key, which cannot be specified by the `key` property.

Use the sample movie Keyboard Lingo to test which characters correspond to different keys on different keyboards.

Example

This handler uses the Message window to display the appropriate key code each time a key is pressed:

```
-- Lingo syntax
on enterFrame
    keyDownScript = put(_key.keyCode)
end

// JavaScript syntax
function enterFrame() {
    keyDownScript = put(_key.keyCode);
}
```

This statement checks whether the up arrow (whose key code is 126) was pressed and if it was, goes to the previous marker:

```
-- Lingo syntax
if (_key.keyCode = 126) then
    _movie.goPrevious()
end if

// JavaScript syntax
if (_key.keyCode == 126) {
    _movie.goPrevious();
}
```

This handler checks whether one of the arrow keys was pressed and if one was, responds accordingly:

```
-- Lingo syntax
on keyDown
    case (_key.keyCode) of
        123: TurnLeft
        126: GoForward
        125: BackUp
        124: TurnRight
    end case
end keyDown

// JavaScript syntax
function keyDown() {
    switch (_key.keyCode) {
        case 123: TurnLeft();
            break;
        case 126: GoForward();
            break;
        case 125: BackUp();
            break;
        case 124: TurnRight();
            break;
    }
}
```

See also

Key, key

keyDownScript

Usage

```
the keyDownScript
```

Description

System property; specifies the Lingo that is executed when a key is pressed. The Lingo is written as a string, surrounded by quotation marks, and can be a simple statement or a calling script for a handler.

When a key is pressed and the `keyDownScript` property is defined, Lingo executes the instructions specified for the `keyDownScript` property first. Unless the instructions include the `pass` command so that the `keyDown` message can be passed on to other objects in the movie, no other `on keyDown` handlers are executed.

Setting the keyDownScript property performs the same function as using the when keyDown then command that appeared in earlier versions of Director.

When the instructions you specify for the keyDownScript property are no longer appropriate, turn them off by using the statement set the keyDownScript to EMPTY.

Example

This statement specifies the script that is executed when a key is pressed.

```
-- Lingo syntax
 the keyDownScript = "go to the frame"

// JavaScript syntax
 _system.keyDownScript = "_movie.go(_movie.frame)";
```

See also

on keyDown, keyUpScript, mouseDownScript, mouseUpScript

keyframePlayer (modifier)

Syntax

member(whichCastmember).model(whichModel).keyframePlayer.keyframePlayerModifierProperty

Description

3D modifier; manages the use of motions by models. The motions managed by the keyframePlayer modifier animate the entire model at once, unlike Bones player motions, which animate segments of the model called bones.

Motions and the models that use them must be created in a 3D modeling program, exported as W3D files, and then imported into a movie. Motions cannot be applied to model primitives created within Director.

Adding the keyframePlayer modifier to a model by using the addModifier command allows access to the following keyframePlayer modifier properties:

- playing indicates whether a model is executing a motion.
- playList is a linear list of property lists containing the playback parameters of the motions that are queued for a model.
- currentTime indicates the local time, in milliseconds, of the currently playing or paused motion.
- playRate is a number that is multiplied by the *scale* parameter of the play() or queue() command to determine the playback speed of the motion.
- playlist.count returns the number of motions currently queued in the playlist.
- rootLock indicates whether the translational component of the motion is used or ignored.
- currentLoopState indicates whether the motion plays once or repeats continuously.
- blendTime indicates the length of the transition created by the modifier between motions when the modifier's autoblend property is set to TRUE.
- autoBlend indicates whether the modifier creates a linear transition to the currently playing motion from the motion that preceded it.
- blendFactor indicates the degree of blending between motions when the modifier's autoBlend property is set to FALSE.

- `lockTranslation` indicates whether the model can be displaced from the specified planes.
- `positionReset` indicates whether the model returns to its starting position after the end of a motion or each iteration of a loop.
- `rotationReset` indicates the rotational element of a transition from one motion to the next, or the looping of a single motion.

Note: *For more detailed information about these properties, see the individual property entries.*

The `keyframePlayer` modifier uses the following commands:

- `pause` halts the motion currently being executed by the model.
- `play()` initiates or unpauses the execution of a motion.
- `playNext()` initiates playback of the next motion in the playlist.
- `queue()` adds a motion to the end of the playlist.

The `keyframePlayer` modifier generates the following events, which are used by handlers declared in the `registerForEvent()` and `registerScript()` commands. The call to the declared handler includes three arguments: the event type (either `#animationStarted` or `#animationEnded`), the name of the motion, and the current time of the motion. For detailed information about notification events, see the entry for `registerForEvent()`.

`#animationStarted` is sent when a motion begins playing. If blending is used between motions, the event is sent when the transition begins.

`#animationEnded` is sent when a motion ends. If blending is used between motions, the event is sent when the transition ends.

See also

`addModifier`, `modifiers`, `bonesPlayer (modifier)`, `motion`

keyUpScript

Usage

```
the keyUpScript
```

Description

System property; specifies the Lingo that is executed when a key is released. The Lingo is written as a string, surrounded by quotation marks, and can be a simple statement or a calling script for a handler.

When a key is released and the `keyUpScript` property is defined, Lingo executes the instructions specified for the `keyUpScript` property first. Unless the instructions include the `pass` command so that the `keyUp` message can be passed on to other objects in the movie, no other `on keyUp` handlers are executed.

When the instructions you've specified for the `keyUpScript` property are no longer appropriate, turn them off by using the statement `set the keyUpScript to empty`.

Example

This statement specifies the script that is executed when a key is pressed.

```
-- Lingo syntax
 the keyUpScript = "go to the frame +1 "
```

```
// JavaScript syntax
 _system.keyUpScript = "_movie.go(_movie.frame+1)";
```

See also

on keyUp

labelList

Usage

```
the labelList
```

Description

System property; lists the frame labels in the current movie as a Return-delimited string (not a list) containing one label per line. Labels are listed according to their order in the Score. (Because the entries are Return-delimited, the end of the string is an empty line after the last Return. Be sure to remove this empty line if necessary.)

Example

This statement lists the frame labels in the current movie as a Return-delimited string (not a list) containing one label per line.

```
put the labelList
```

See also

frameLabel, label(), marker()

lastChannel

Usage

```
-- Lingo syntax
_movie.lastChannel

// JavaScript syntax
_movie.lastChannel;
```

Description

Movie property; the number of the last channel in the movie, as entered in the Movie Properties dialog box. Read-only.

To see an example of `lastChannel` used in a completed movie, see the QT and Flash movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement displays the number of the last channel of the movie in the Message window:

```
-- Lingo syntax
put(_movie.lastChannel)

// JavaScript syntax
put(_movie.lastChannel);
```

See also

Movie

lastClick

Usage

```
-- Lingo syntax
_player.lastClick

// JavaScript syntax
_player.lastClick;
```

Description

Player property; returns the time in ticks (1 tick = 1/60 of a second) since the mouse button was last pressed. Read-only.

Example

This statement checks whether 10 seconds have passed since the last mouse click and, if so, sends the playhead to the marker No Click:

```
-- Lingo syntax
if (_player.lastClick > (10 * 60)) then
    _movie.go("No Click")
end if

// JavaScript syntax
if (_player.lastClick > (10 * 60)) {
    _movie.go("No Click");
}
```

See also

lastEvent, lastKey, lastRoll, Player

lastError

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.lastError

// JavaScript syntax
memberOrSpriteObjRef.lastError;
```

Description

RealMedia sprite or cast member property; allows you to get the last error symbol returned by RealPlayer® as a Lingo symbol. The error symbols returned by RealPlayer are strings of simple English and provide a starting point for the troubleshooting process. This property is dynamic during playback and can be tested but not set.

The value `#PNR_OK` indicates that everything is functioning properly.

Example

The following examples show that the last error returned by RealPlayer for the sprite 2 and the cast member Real was #PNR_OUTOFMEMORY:

```
-- Lingo syntax
put(sprite(2).lastError) -- #PNR_OUTOFMEMORY
put(member("Real").lastError) -- #PNR_OUTOFMEMORY

// JavaScript syntax
trace(sprite(2).lastError); // #PNR_OUTOFMEMORY
put(member("Real").lastError); // #PNR_OUTOFMEMORY
```

lastEvent

Usage

```
-- Lingo syntax
_player.lastEvent

// JavaScript syntax
_player.lastEvent;
```

Description

Player property; returns the time in ticks (1 tick = 1/60 of a second) since the last mouse click, rollover, or key press occurred. Read-only.

Example

This statement checks whether 10 seconds have passed since the last mouse click, rollover, or key press and, if so, sends the playhead to the marker Help:

```
-- Lingo syntax
if (_player.lastEvent > (10 * 60)) then
    _movie.go("Help")
end if

// JavaScript syntax
if (_player.lastEvent > (10 * 60)) {
    _movie.go("Help");
}
```

See also

lastClick, lastKey, lastRoll, Player

lastFrame

Usage

```
-- Lingo syntax
_movie.lastFrame

// JavaScript syntax
_movie.lastFrame;
```

Description

Movie property; displays the number of the last frame in the movie. Read-only.

Example

This statement displays the number of the last frame of the movie in the Message window:

```
-- Lingo syntax
put(_movie.lastFrame)

// JavaScript syntax
put(_movie.lastFrame);
```

See also

Movie

lastKey

Usage

```
-- Lingo syntax
_player.lastKey

// JavaScript syntax
_player.lastKey;
```

Description

Player property; gives the time in ticks (1 tick = 1/60 of a second) since the last key was pressed. Read-only.

Example

This statement checks whether 10 seconds have passed since the last key was pressed and, if so, sends the playhead to the marker No Key:

```
-- Lingo syntax
if (_player.lastKey > (10 * 60)) then
    _movie.go("No Key")
end if

// JavaScript syntax
if (_player.lastKey > (10 * 60)) {
    _movie.go("No Key");
}
```

See also

lastClick, lastEvent, lastRoll, Player

lastRoll

Usage

```
-- Lingo syntax
_player.lastRoll

// JavaScript syntax
_player.lastRoll;
```

Description

Player property; gives the time in ticks (1 tick = 1/60 of a second) since the mouse was last moved. Read-only.

Example

This statement checks whether 45 seconds have passed since the mouse was last moved and, if so, sends the playhead to the marker Attract Loop:

```
-- Lingo syntax
if (_player.lastRoll > (45 * 60)) then
    _movie.go("Attract Loop")
end if

// JavaScript syntax
if (_player.lastRoll > (45 * 60)) {
    _movie.go("Attract Loop");
}
```

See also

lastClick, lastEvent, lastKey, Player

left

Usage

```
-- Lingo syntax
spriteObjRef.left

// JavaScript syntax
spriteObjRef.left;
```

Description

Sprite property; identifies the left horizontal coordinate of the bounding rectangle of a sprite. Read/write.

Sprite coordinates are measured in pixels, starting with (0,0) at the upper left corner of the Stage.

Example

The following statement determines whether the sprite's left edge is to the left of the Stage's left edge. If the sprite's left edge is to the Stage's left edge, the script runs the handler `offLeftEdge`:

```
-- Lingo syntax
if (sprite(3).left < 0) then
    offLeftEdge()
end if

// JavaScript syntax
if (sprite(3).left < 0) {
    offLeftEdge();
}
```

This statement measures the left horizontal coordinate of the sprite numbered (i + 1) and assigns the value to the variable named `vLowest`:

```
-- Lingo syntax
vLowest = sprite(i + 1).left

// JavaScript syntax
var vLowest = sprite(i + 1).left
```

See also

bottom, height, locH, locV, right, Sprite, top, width

left (3D)

Usage

```
member(whichCastmember).modelResource(whichModelResource).left
```

Description

3D #box model resource property; indicates whether the side of the box intersected by its -X axis is sealed (TRUE) or open (FALSE).

The default value for this property is TRUE.

Example

This statement sets the left property of the model resource named Crate to FALSE, meaning the left side of this box will be open:

```
member("3D World").modelResource("crate").left = FALSE
```

See also

back, front, bottom (3D), top (3D), right (3D)

leftIndent

Usage

```
chunkExpression.leftIndent
```

Description

Text cast member property; contains the number of pixels the left margin of *chunkExpression* is offset from the left side of the text cast member.

The value is an integer greater than or equal to 0.

This property can be tested and set.

Example

This statement left indents the text cast member "myText" by ten pixels.

```
-- Lingo syntax
member("myText").leftIndent=10

// JavaScript syntax
member("myText").leftIndent=10;
```

See also

firstIndent, rightIndent

length (3D)

Usage

```
member(whichCastmember).modelResource(whichModelResource).length
vectorReference.length
```

Description

3D `#box` model resource, `#plane` model resource, and vector property; indicates the length in world units of the box or plane.

The length of a box is measured along its Z axis. The default length of a box is 50.

The length of a plane is measured along its Y axis. The default length of a plane is 1.

The length of a vector is its distance in world units from `vector(0, 0, 0)`. This is the same as the magnitude of the vector.

Example

This statement sets the variable `myBoxLength` to the length of the model resource named GiftBox.

```
myBoxLength = member("3D World").modelResource("GiftBox").length
```

See also

`height (3D)`, `width (3D)`, `magnitude`

lengthVertices

Usage

```
member(whichCastmember).modelResource(whichModelResource).lengthVertices
```

Description

3D `#box` and `#plane` model resource property; indicates the number of mesh vertices along the length of the box or plane. Increasing this value increases the number of faces, and therefore the fineness, of the mesh.

The length of a box is measured along its Z axis. The length of a plane is measured along its Y axis.

Set the `renderStyle` property of a model's shader to `#wire` to see the faces of the mesh of the model's resource. Set the `renderStyle` property to `#point` to see just the vertices of the mesh.

The value of this property must be greater than or equal to 2. The default value is 4.

Example

The following statement sets the `lengthVertices` property of the model resource named Tower to 10. Nine triangles will be used to define the geometry of the model resource along its Y axis; therefore, there will be ten vertices.

```
member("3D World").modelResource("Tower").lengthVertices = 10
```

See also

`length (3D)`

level

Usage

```
member(whichCastmember).model(whichModel).lod.level
```

Description

3D `lod` modifier property; indicates the amount of detail removed by the modifier when its `auto` property is set to `FALSE`.The range of this property is 0.0 to 100.00.

When the modifier's `auto` property is set to `TRUE`, the value of the `level` property is dynamically updated, but cannot be set.

The `#lod` modifier can only be added to models created outside of Director in 3D modeling programs. The value of the `type` property of the model resources used by these models is `#fromFile`. The modifier cannot be added to primitives created within Director.

Example

The following statement sets the `level` property of the lod modifier of the model Spaceship to 50. If the lod modifier's `auto` property is set to `FALSE`, Spaceship will be drawn at a medium level of detail. If the lod modifier's `auto` property is set to `TRUE`, this code will have no effect.

```
member("3D World").model("Spaceship").lod.level = 50
```

See also

lod (modifier), auto, bias

lifetime

Usage

```
member(whichCastmember).modelResource(modelResource).lifetime
```

Description

3D `#particle` model resource property; for all particles in a particle system, this property indicates the number of milliseconds from the creation of a particle to the end of its existence.

The default value of this property is 10,000.

Example

This example sets the `lifetime` property of a particle to 100 milliseconds in the cast member named 3Dobjects.

```
-- Lingo syntax
member("3Dobjects").modelResource("Particle01").lifetime = 100.0

// JavaScript syntax
member("3Dobjects").getPropRef("modelResource",10).lifetime = 100.0;
```

See also

emitter

light

Usage

```
member(whichCastmember).light(whichLight)
member(whichCastmember).light[index]
member(whichCastmember).light(whichLight).whichLightProperty
member(whichCastmember).light[index].whichLightProperty
```

Description

3D element; an object at a vector position from which light emanates.

Example

This example shows the first light in the cast member named 3Dobjects.

```
-- Lingo syntax
putmember("3Dobjects").light(1)

// JavaScript syntax
put( member("3Dobjects").getPropRef("light",1));
```

See also

newLight, deleteLight

lineColor

Usage

```
member(whichCastmember).model(whichModel).inker.lineColor
member(whichCastmember).model(whichModel).toon.lineColor
```

Description

3D `toon` and `inker` modifier property; indicates the color of the lines drawn on the model by the modifier. For this property to have an effect, either the modifier's `creases`, `silhouettes`, or `boundary` property must be set to `TRUE`.

The default value for this property is `rgb(0, 0, 0)`.

Example

This statement sets the color of all lines drawn by the `toon` modifier on the second model to rgb(255, 0, 0), which is red:

```
-- Lingo syntax
member("3Dobjects").model[2].toon.lineColor = rgb(255, 0, 0)

// JavaScript syntax
member("3Dobjects").getPropRef("model",2).toon.lineColor =color(100,0,0);
```

See also

creases, silhouettes, boundary, lineOffset

lineCount

Usage

```
-- Lingo syntax
memberObjRef.lineCount

// JavaScript syntax
memberObjRef.lineCount;
```

Description

Cast member property; indicates the number of lines that appear in the field cast member on the Stage according to the way the string wraps, not the number of carriage returns in the string.

Example

This statement determines how many lines the field cast member Today's News has when it appears on the Stage and assigns the value to the variable `numberOfLines`:

```
--Lingo syntax
numberOfLines = member("Today's News").lineCount

// JavaScript syntax
var numberOfLines = member("Today's News").lineCount;
```

lineDirection

Usage

```
member(whichCastMember).lineDirection
```

Description

Shape member property; this property contains a 0 or 1 indicating the slope of the line drawn.

If the line is inclined from left to right, the property is set to 1; and if it is declined from left to right, the property is set to 0.

This property can be tested and set.

Example

This example toggles the slope of the line in cast member "theLine", producing a see-saw effect:

```
-- Lingo syntax
member("theLine").lineDirection = not member("theLine").lineDirection

// JavaScript syntax
member("theLine").lineDirection = ! (member("theLine").lineDirection);
```

lineHeight

Usage

```
member(whichCastMember).lineHeight
the lineHeight of member whichCastMember
```

Description

Cast member property; determines the line spacing used to display the specified field cast member. The parameter *whichCastMember* can be either a cast member name or number.

Setting the lineHeight member property temporarily overrides the system's setting until the movie closes. To use the desired line spacing throughout a movie, set the lineHeight member property in an on prepareMovie handler.

This property can be tested and set.

Example

This statement sets the line height in cast member "myText".

```
-- Lingo syntax
member("myText").lineHeight=20

// JavaScript syntax
member("myText").lineHeight=20;
```

See also

text, alignment, font, fontSize, fontStyle

lineOffset

Usage

```
member(whichCastmember).model(whichModel).toon.lineOffset
member(whichCastmember).model(whichModel).inker.lineOffset
```

Description

3D toon and inker modifier property; indicates the apparent distance from the model's surface at which lines are drawn by the modifier. For this property to have an effect, the modifier's useLineOffset property must be set to TRUE, and one or more of its creases, silhouettes, or boundary properties must also be set to TRUE.

This range of this property is -100.00 to +100.00. Its default setting is -2.0.

Example

The following statement sets the lineOffset property of the toon modifier for the model named Teapot to 10. The lines drawn by the toon modifier on the surface of the model will stand out more than they would at the default setting of -2.

```
member("shapes").model("Teapot").toon.lineOffset = 10
```

See also

creases, silhouettes, boundary, useLineOffset, lineColor

lineSize

Usage

```
member(whichCastMember).lineSize
the lineSize of member whichCastMember
sprite whichSprite.lineSize
the lineSize of sprite whichSprite
```

Description

Shape cast member property; determines the thickness, in pixels, of the border of the specified shape cast member displayed on the Stage. For nonrectangular shapes, the border is the edge of the shape, not its bounding rectangle.

The `lineSize` setting `of` the sprite takes precedence over the `lineSize` setting `of` the member. If Lingo changes the member's `lineSize` setting while a sprite is on the Stage, the sprite's `lineSize` setting remains in effect until the sprite is finished.

For the value set by Lingo to last beyond the current sprite, the sprite must be a scripted sprite.

This property can be tested and set.

Example

This statement sets the thickness of the shape cast member myShape to 5 pixels.

```
--Lingo syntax
member("myShape").lineSize=5

// JavaScript syntax
member("myShape").lineSize=5;
```

linked

Usage

```
-- Lingo syntax
memberObjRef.linked

// JavaScript syntax
memberObjRef.linked;
```

Description

Member property; controls whether a script, Flash movie, or animated GIF file is stored in an external file (`TRUE`, default), or inside the Director cast library (`FALSE`). Read/write for script, Flash, and animated GIF members, read-only for all other member types.

When the data is stored externally in a linked file, the cast member's `pathName` property must point to the location where the movie file can be found.

Example

This statement converts Flash cast member "homeBodies" from a linked member to an internally stored member.

```
-- Lingo syntax
member("homeBodies").linked = 0

// JavaScript syntax
member("homeBodies").linked = 0;
```

See also

Member

loaded

Usage

```
-- Lingo syntax
memberObjRef.loaded

// JavaScript syntax
memberObjRef.loaded;
```

Description

Member property; specifies whether a specified cast member is loaded into memory (`TRUE`) or not (`FALSE`). Read-only.

Different cast member types have slightly different behaviors for loading:

- Shape and script cast members are always loaded into memory.
- Movie cast members are never unloaded.
- Digital video cast members can be preloaded and unloaded independent of whether they are being used. (A digital video cast member plays faster from memory than from disk.)

Example

This statement checks whether cast member Demo Movie is loaded in memory and if it isn't, goes to an alternative movie:

```
-- Lingo syntax
if member("Demo Movie").loaded = FALSE then
    _movie.go(1, "Waiting.dir")
end if

// JavaScript syntax
if (member("Demo Movie").loaded == false) {
    _movie.go(1, "Waiting.dir")
}
```

See also

`Member`

loc (backdrop and overlay)

Usage

```
sprite(whichSprite).camera{(index)}.backdrop[index].loc
member(whichCastmember).camera(whichCamera).backdrop[index].loc
sprite(whichSprite).camera{(index)}.overlay[index].loc
member(whichCastmember).camera(whichCamera).overlay[index].loc
```

Description

3D backdrop and overlay property; indicates the 2D location of the backdrop or overlay, as measured from the upper left corner of the sprite.

This property is initially set as a parameter of the `addBackdrop`, `addOverlay`, `insertBackdrop`, or `insertOverlay` command which creates the backdrop or overlay.

Example

This statement positions the first backdrop of the camera of sprite 2:

```
sprite(2).camera.backdrop[1].loc = point(120, 120)
```

See also

bevelDepth, overlay, regPoint (3D)

locH

Usage

```
-- Lingo syntax
spriteObjRef.locH

// JavaScript syntax
spriteObjRef.locH;
```

Description

Sprite property; indicates the horizontal position of a sprite's registration point. Read/write.

Sprite coordinates are relative to the upper left corner of the Stage.

To make the value last beyond the current sprite, make the sprite a scripted sprite.

Example

This statement puts sprite 15 at the same horizontal location as the mouse click:

```
-- Lingo syntax
sprite(15).locH = _mouse.mouseH

// JavaScript syntax
sprite(15).locH = _mouse.mouseH;
```

See also

bottom, height, left, locV, point(), right, Sprite, top, updateStage()

lockTranslation

Usage

```
member(whichCastmember).model(whichModel).bonesPlayer.lockTranslation
member(whichCastmember).model(whichModel).keyframePlayer.lockTranslation
```

Description

3D `#bonesPlayer` and `#keyframePlayer` modifier property; prevents displacement from the specified plane(s) except by the absolute translation of the motion data. Any additional translation introduced either manually or through cumulative error is removed. The possible values of `#none`, `#x`, `#y`, `#z`, `#xy`, `#yz`, `#xz`, and `#all` control which of the three translational components are controlled for each frame. When a lock on an axis is turned on, the current displacement along that axis is stored and used thereafter as the fixed displacement to which the animation is relative. This displacement can be reset by deactivating that axis lock, moving the object, and reactivating that axis lock.

In other words, it defines the axis of translation to ignore when playing back a motion. To keep a model locked to a ground plane with the top pointing along the *z*-axis, set `lockTranslation` to `#z`. The default value for this property is `#none`.

Example

This statement sets the `lockTranslation` property of the model named Walker to `#z`.

```
member("ParkScene").model("Walker").bonesPlayer.lockTranslation = #z
```

See also

`immovable`

locV

Usage

```
-- Lingo syntax
spriteObjRef.locV

// JavaScript syntax
spriteObjRef.locV;
```

Description

Sprite property; indicates the vertical position of a sprite's registration point. Read/write.

Sprite coordinates are relative to the upper left corner of the Stage.

To make the value last beyond the current sprite, make the sprite a scripted sprite.

Example

This statement puts sprite 15 at the same vertical location as the mouse click:

```
-- Lingo syntax
sprite(15).locV = _mouse.mouseV

// JavaScript syntax
sprite(15).locV = _mouse.mouseV;
```

See also

`bottom`, `height`, `left`, `locH`, `point()`, `right`, `Sprite`, `top`, `updateStage()`

locZ

Usage

```
-- Lingo syntax
spriteObjRef.locZ

// JavaScript syntax
spriteObjRef.locZ;
```

Description

Sprite property; specifies the dynamic Z-order of a sprite, to control layering without having to manipulate sprite channels and properties. Read/write.

This property can have an integer value from negative 2 billion to positive 2 billion. Larger numbers cause the sprite to appear in front of sprites with smaller numbers. If two sprites have the same `locZ` value, the channel number then takes precedence for deciding the final display order of those two sprites. This means sprites in lower numbered channels appear behind sprites in higher numbered channels even when the `locZ` values are equal.

By default, each sprite has a `locZ` value equal to its own channel number.

Layer-dependent operations such as hit detection and mouse events obey sprites' `locZ` values, so changing a sprite's `locZ` value can make the sprite partially or completely obscured by other sprites and the user may be unable to click on the sprite.

Other Director functions do not follow the `locZ` ordering of sprites. Generated events still begin with channel 1 and increase consecutively from there, regardless of the sprite's Z-order.

Example

This handler uses a global variable called `gHighestSprite` which has been initialized in the startMovie handler to the number of sprites used. When the sprite is clicked, its locZ is set to gHighestSprite + 1, which moves the sprite to the foreground on the stage. Then gHighestSprite is incremented by 1 to prepare for the next mouseUp call.

```
-- Lingo syntax
on mouseUp me
    global gHighestSprite
    sprite(me.spriteNum).locZ = gHighestSprite + 1
    gHighestSprite = gHighestSprite + 1
end

// JavaScript syntax
function mouseUp() {
    _global.gHighestSprite;
    sprite(this.spriteNum).locZ = _global.gHighestSprite + 1
    _global.gHighestSprite = _global.gHighestSprite + 1
}
```

See also

locH, locV, Sprite

lod (modifier)

Usage

```
member(whichCastmember).model(whichModel).lod.lodModifierProperty
```

Description

3D modifier; dynamically removes detail from models as they move away from the camera.

This modifier can only be added to models created outside of Director in 3D modeling programs. The value of the `type` property of the model resources used by these models is `#fromFile`. All such models use detail reduction whether or not the `lod` modifier is attached. Attaching the modifier allows you to control the properties of detail reduction. The modifier cannot be added to primitives created within Director.

The `lod` modifier data is generated by 3D modeling programs for all models. Setting the `userData` property `"sw3d_no_lod = true"` allows you to specify that the `lod` modifier data and memory be released when streaming is complete.

Be careful when using the `sds` and `lod` modifiers together, because they perform opposite functions (the `sds` modifier adds geometric detail and the `lod` modifier removes geometric detail). Before adding the `sds` modifier, it is recommended that you disable the `lod.auto` modifier property and set the `lod.level` modifier property to maximum resolution, as follows:

```
member("myMember").model("myModel").lod.auto = 0
member("myMember").model("myModel").lod.level = 100
member("myMember").model("myModel").addmodifier(#sds)
```

The `lod` modifier has the following properties:

- `auto` allows the modifier to set the level of detail reduction as the distance between the model and the camera changes. The value of the modifier's `level` property is updated, but setting the `level` property will have no effect when the `auto` property is set to `TRUE`.
- `bias` indicates how aggressively the modifier removes detail from the model when the modifier's `auto` property is set to `TRUE`. The range for this property is from 0.0 (removes all polygons) to 100.0 (removes no polygons). The default setting for this property is 100.0.
- `level` indicates the amount of detail reduction there will be when the modifier's `auto` property is set to `FALSE`. The range of this property is 0.0 to 100.00.

Note: *For more detailed information about these properties, see the individual property entries.*

Example

This example dynamically removes the model Sphere01 from the cast member named 3Dobjects.

```
--Lingo
member("3Dobjects").model("Sphere01").lod.auto=1
member("3Dobjects").model("Sphere01").lod.bias=0

// Javascript
member("3Dobjects").getPropRef("model",2).lod.auto=1;
member("3Dobjects").getPropRef("model",2).lod.bias=0;
```

See also

`sds (modifier)`, `auto`, `bias`, `level`, `addModifier`

loop (3D)

Usage

```
member(whichCastmember).loop
```

Description

3D cast member property; indicates whether motions applied to the first model in the cast member repeat continuously (TRUE) or play once and stop (FALSE).

The default setting for this property is TRUE.

Example

This statement sets the loop property of the cast member named Walkers to TRUE. Motions being executed by the first model in Walker will repeat continuously.

```
member("Walkers").loop = TRUE
```

See also

motion, play() (3D), queue() (3D), animationEnabled

loop (emitter)

Usage

```
member(whichCastmember).modelResource(whichModelResource).emitter.loop
```

Description

3D property; when used with a model resource whose type is #particle, this property allows you to both get and set what happens to particles at the end of their lifetime. A loop value of TRUE causes particles to be reborn at the end of their lifetime at the emitter location defined by the emitter's region property. A value of FALSE causes the particles to die at the end of their lifetime. The default setting for this property is TRUE.

Example

In this example, ThermoSystem is a model resource of the type #particle. This statement sets the emitter.loop property of ThermoSystem to 1, which causes the particles of ThermoSystem to be continuously emitted.

```
member("Fires").modelResource("ThermoSystem").emitter.loop = 1
```

See also

emitter

loop (Member)

Usage

```
-- Lingo syntax
memberObjRef.loop

// JavaScript syntax
memberObjRef.loop;
```

Description

Cast member property; determines whether the specified digital video, sound, or Flash movie cast member is set to loop (TRUE) or not (FALSE).

Example

This statement sets the QuickTime movie cast member Demo to loop:

```
-- Lingo syntax
member("Demo").loop = 1

// JavaScript syntax
member("Demo").loop = 1;
```

loop (Flash)

Usage

```
sprite(whichFlashSprite).loop
the loop of sprite whichFlashSprite
member (whichFlashMember).loop
the loop of member whichFlashMember
```

Description

Flash sprite and member property; controls whether a Flash movie plays in a a continuous loop (`TRUE`) or plays once and then stops (`FALSE`).

The property can be both tested and set.

Example

This frame script checks the streamed status of a linked Flash cast member called flashObj using the `percentStreamed` property. While flashObj is downloading, the movie loops in the current frame. When flashObj finishes downloading, the movie advances to the next frame and the `loop` property of the Flash movie in channel 1 is set to `FALSE` so that it will continue playing through to the end and then stop (imagine that this sprite has been looping while flashObj was downloading).

```
-- Lingo syntax
on exitFrame me
if member("flashObj").percentStreamed = 100 then
    member(1).loop = FALSE
    go the frame + 1
else
    go to the frame
end if

end
// JavaScript syntax
function exitFrame(me) {
if(member("flashObj").percentStreamed == 100)
    {
        member(1).loop = 0;
        _movie.go(_movie.frame+1);
    }
    else
    {
        _movie.go(_movie.frame);
    }
}
```

loop (Windows Media)

Usage

```
-- Lingo syntax
windowsMediaObjRef.loop

// JavaScript syntax
windowsMediaObjRef.loop;
```

Description

Windows Media property. Determines whether a movie loops (`TRUE`, default) or not (`FALSE`) after completion. Read/write.

Example

This statement specifies that the cast member Classical should loop after completion:

```
-- Lingo syntax
member("Classical").loop = TRUE

// JavaScript syntax
member("Classical").loop = true;
```

See also

`Windows Media`

loopBounds

Usage

```
-- Lingo syntax
spriteObjRef.loopBounds

// JavaScript syntax
spriteObjRef.loopBounds;
```

Description

QuickTime sprite property; sets the internal loop points for a QuickTime cast member or sprite. The loop points are specified as a Director list: [*`startTime`*, *`endTime`*].

The *`startTime`* and *`endTime`* parameters must meet these requirements:

- Both parameters must be integers that specify times in Director ticks.
- The values must range from 0 to the duration of the QuickTime cast member.
- The starting time must be less than the ending time.

If any of these requirements is not met, the QuickTime movie loops through its entire duration.

The `loopBounds` property has no effect if the movie's `loop` property is set to `FALSE`. If the `loop` property is set to `TRUE` while the movie is playing, the movie continues to play. Director uses these rules to decide how to loop the movie:

- If the ending time specified by `loopBounds` is reached, the movie loops back to the starting time.
- If the end of the movie is reached, the movie loops back to the start of the movie.

If the loop property is turned off while the movie is playing, the movie continues to play. Director stops when it reaches the end of the movie.

This property can be tested and set. The default setting is [0,0].

Example

This sprite script sets the starting and ending times for looping within a QuickTime sprite. The times are set by specifying seconds, which are then converted to ticks by multiplying by 60.

```
-- Lingo syntax
on beginSprite me
    sprite(me.spriteNum).loopBounds = [(16 * 60),(32 * 60)]
end

// JavaScript syntax
function beginSprite() {
    sprite(me.spriteNum).loopBounds = list((16 * 60),(32 * 60));
}
```

loopCount

Usage

```
-- Lingo syntax
soundChannelObjRef.loopCount

// JavaScript syntax
soundChannelObjRef.loopCount;
```

Description

Sound Channel property; specifies the total number of times the current sound in a sound channel is set to loop. Read-only.

The default value of this property is 1 for sounds that are simply queued with no internal loop.

You can loop a portion of a sound by passing the parameters loopStartTime, loopEndTime, and loopCount with a queue() or setPlayList() method. These are the only methods for setting this property.

If loopCount is set to 0, the loop will repeat forever. If the sound cast member's loop property is set to TRUE, loopCount will return 0.

Example

This handler queues and plays two sounds in sound channel 2. The first sound, cast member introMusic, loops five times between 8 seconds and 8.9 seconds. The second sound, cast member creditsMusic, loops three times. However, no #loopStartTime and #loopEndTime are specified, so these values default to the #startTime and #endTime, respectively.

```
-- Lingo syntax
on playMusic
    sound(2).queue([#member:member("introMusic"), #startTime:3000, #loopCount:5,
#loopStartTime:8000, #loopEndTime:8900])
    sound(2).queue([#member:member("creditsMusic"), #startTime:3000, #endTime:3000,
#loopCount:3])
    sound(2).play()
end playMusic
```

```
// JavaScript syntax
function playMusic() {
    sound(2).queue(propList("member",member("introMusic"), "startTime",3000,"loopCount",5,
"loopStartTime",8000, "loopEndTime",8900));
    sound(2).queue(propList("member",member("creditsMusic"),
"startTime",3000,"endTime",3000, "loopCount",3]);
    sound(2).play();
}
```

See also

loopEndTime, loopStartTime, queue(), setPlayList(), Sound Channel

loopEndTime

Usage

```
-- Lingo syntax
soundChannelObjRef.loopEndTime

// JavaScript syntax
soundChannelObjRef.loopEndTime;
```

Description

Sound Channel property; specifies the end time, in milliseconds, of the loop set in the current sound playing in a sound channel. Read-only.

The value of this property is a floating-point number, allowing you to measure and control sound playback to fractions of a millisecond.

This property can only be set when passed as a property in a `queue()` or `setPlaylist()` command.

Example

This handler plays sound cast member introMusic in sound channel 2. Playback loops five times between the 8 seconds point and the 8.9 second point in the sound.

```
-- Lingo syntax
on playMusic
    sound(2).play([#member:member("introMusic"), #startTime:3000, #loopCount:5 \
#loopStartTime:8000, #loopEndTime:8900])
end playMusic

// JavaScript syntax
function playMusic() {
    sound(2).play(propList("member",member("introMusic"), "startTime",3000,"loopCount",5,
"loopStartTime",8000, "loopEndTime",8900));
}
```

See also

queue(), setPlayList(), Sound Channel

loopsRemaining

Usage

```
-- Lingo syntax
soundChannelObjRef.loopsRemaining

// JavaScript syntax
soundChannelObjRef.loopsRemaining;
```

Description

Sound Channel property; specifies the number of times left to play a loop in the current sound playing in a sound channel. Read-only.

If the sound had no loop specified when it was queued, this property is 0. If this property is tested immediately after a sound starts playing, it returns one less than the number of loops defined with the `#loopCount` property in the `queue()` or `setPlayList()` methods.

Exceptions

This example shows the loops remaining in the sound channel 2.

```
-- Lingo syntax
put sound(2).loopsRemaining

// JavaScript syntax
put(sound(2).loopsRemaining);
```

See also

`loopCount`, `queue()`, `setPlayList()`, `Sound Channel`

loopStartTime

Usage

```
-- Lingo syntax
soundChannelObjRef.loopStartTime

// JavaScript syntax
soundChannelObjRef.loopStartTime;
```

Description

Sound Channel property; specifies the start time, in milliseconds, of the loop for the current sound being played by a sound channel. Read-only.

Its value is a floating-point number, allowing you to measure and control sound playback to fractions of a millisecond. The default is the `startTime` of the sound if no loop has been defined.

This property can only be set when passed as a property in a `queue()` or `setPlaylist()` methods.

Example

This handler plays sound cast member introMusic in sound channel 2. Playback loops five times between two points 8 seconds and 8.9 seconds into the sound.

```
-- Lingo syntax
on playMusic
```

```
    sound(2).play([#member:member("introMusic"), #startTime:3000, #loopCount:5
\#loopStartTime:8000, #loopEndTime:8900])
end playMusic

// JavaScript syntax
function playMusic() {
    sound(2).play(propList("member",member("introMusic"), "startTime",3000,"loopCount",5,
"loopStartTime",8000, "loopEndTime",8900));
}
```

See also

queue(), setPlayList(), Sound Channel, startTime

magnitude

Usage

```
whichVector.magnitude
```

Description

3D property; returns the magnitude of a vector. The value is a floating-point number. The magnitude is the length of a vector and is always greater than or equal to0.0. (`vector (0, 0, 0)` equals 0.)

Example

This statement shows that the magnitude of MyVec1 is 100.

```
-- Lingo syntax
MyVec1 = vector(100, 0, 0)
put MyVec1.magnitude

// JavaScript syntax
MyVec1 = vector(100, 0, 0);
put(MyVec1.magnitude);
```

See also

length (3D), identity()

margin

Usage

```
-- Lingo syntax
memberObjRef.margin

// JavaScript syntax
memberObjRef.margin;
```

Description

Field cast member property; determines the size, in pixels, of the margin inside the field box.

Example

The following statement sets the margin inside the box for the field cast member Today's News to 15 pixels:

```
--Lingo syntax
```

```
member("Today's News").margin = 15

// JavaScript syntax
member("Today's News").margin = 15;
```

markerList

Usage

```
-- Lingo syntax
_movie.markerList

// JavaScript syntax
_movie.markerList;
```

Description

Movie property; contains a script property list of the markers in the Score. Read-only.

The list is of the format:

```
frameNumber: "markerName"
```

Example

This statement displays the list of markers in the Message window:

```
-- Lingo syntax
put(_movie.markerList)

// JavaScript syntax
put(_movie.markerList);
```

See also

Movie

mask

Usage

```
-- Lingo syntax
memberObjRef.mask

// JavaScript syntax
memberObjRef.mask;
```

Description

Cast member property; specifies a black-and-white (1-bit) cast member to be used as a mask for media rendered direct to Stage with media appearing in the areas where the mask's pixels are black. The `mask` property lets you benefit from the performance advantages of a direct-to-Stage digital video while playing a QuickTime movie in a nonrectangular area. The `mask` property has no effect on non–direct-to-Stage cast members.

Director always aligns the registration point of the mask cast member with the upper left of the QuickTime movie sprite. Be sure to reset the registration point of a bitmap to the upper left corner, as it defaults to the center. The registration point of the QuickTime member cannot be reset from the upper left corner. The mask cast member can't be moved and is not affected by the `center` and `crop` properties of its associated cast member.

For best results, set a QuickTime cast member's mask property before any of its sprites appear on the Stage in the `on beginSprite` event handler. Setting or changing the `mask` property while the cast member is on the Stage can have unpredictable results (for example, the mask may appear as a freeze frame of the digital video at the moment the `mask` property took effect).

Masking is an advanced feature; you may need to experiment to achieve your goal.

This property can be tested and set. To remove a mask, set the `mask` property to 0.

Example

This frame script sets a mask for a QuickTime sprite before Director begins to draw the frame:

```
-- Lingo syntax
on prepareFrame
    member("Peeping Tom").mask = member("Keyhole")
end

// JavaScript syntax
function prepareFrame() {
    member("Peeping Tom").mask = member("Keyhole");
}
```

See also

invertMask

maxInteger

Usage

```
the maxInteger
```

Description

System property; returns the largest whole number that is supported by the system. On most personal computers, this is 2,147,483,647 (2 to the thirty-first power, minus 1).

This property can be useful for initializing boundary variables before a loop or for limit testing.

To use numbers larger than the range of addressable integers, use floating-point numbers instead. They aren't processed as quickly as integers, but they support a greater range of values.

Example

This statement generates a table, in the Message window, of the maximum decimal value that can be represented by a certain number of binary digits:

```
on showMaxValues
    b = 31
    v = the maxInteger
    repeat while v > 0
        put b && "-" && v
        b = b-1
        v = v/2
    end repeat
end
```

maxSpeed

Usage

```
member(whichCastmember).modelResource(whichModelResource).emitter.maxSpeed
```

Description

3D property; when used with a model resource whose type is `#particle`, allows you to get and set the maximum speed at which particles are emitted. Each particle's initial velocity is randomly selected between the emitter's `minSpeed` and `maxSpeed` properties.

The value is a floating-point number and must be greater than 0.0.

Example

This example sets the max speed of the particle to 15 in the cast member named 3Dobjects.

```
-- Lingo syntax
member("3Dobjects").modelResource("Particle01").emitter.maxSpeed=15

// JavaScript syntax
member("3Dobjects").getPropRef("modelResource",10).emitter.maxSpeed=15;
```

See also

`minSpeed`, `emitter`

media

Usage

```
-- Lingo syntax
memberObjRef.media

// JavaScript syntax
memberObjRef.media;
```

Description

Member property; identifies the specified cast member as a set of numbers. Read/write.

Because setting this property can use large amounts of memory, it is best used during authoring only.

You can use the `media` property to copy the content of one cast member into another cast member by setting the second member's `media` value to the `media` value for the first member.

For a film loop cast member, the `media` property specifies a selection of frames and channels in the Score.

To swap media in a projector, it is more efficient to set the `member` sprite property.

Example

This statement copies the content of the cast member Sunrise into the cast member Dawn by setting the `media` member property value for Dawn to the `media` member property value for Sunrise:

```
-- Lingo syntax
member("Dawn").media = member("Sunrise").media

// JavaScript syntax
member("Dawn").media = member("Sunrise").media;
```

See also

Member

mediaReady

Usage

```
-- Lingo syntax
memberObjRef.mediaReady

// JavaScript syntax
memberObjRef.mediaReady;
```

Description

Member property; determines whether the contents of a cast member, a movie or cast library file, or a linked cast member is downloaded from the Internet and is available on the local disk (TRUE) or not (FALSE). Read-only.

This property is useful only when streaming a movie or cast library file. Movie streaming is activated by setting the Movie:Playback properties in the Modify menu to Play While Downloading Movie (default setting).

For a demonstration of the mediaReady property, see the sample movie Streaming Shockwave in Director Help.

Example

This statement changes cast members when the desired cast member is downloaded and available locally:

```
-- Lingo syntax
if member("background").mediaReady = TRUE then sprite(2).member =
member("background").number
end if

// JavaScript syntax
if (member("background").mediaReady == true) {
    sprite(2).member = member("background").number;
}
```

See also

Member

mediaStatus (DVD)

Usage

```
-- Lingo syntax
dvdObjRef.mediaStatus

// JavaScript syntax
dvdObjRef.mediaStatus;
```

Description

DVD property; returns a symbol that indicates the current state of the DVD player. Read-only.

Possible symbols include the following:

Symbol	Description
#stopped	The DVD is stopped.
#playing	The DVD is playing.
#paused	The DVD is paused.
#scanning	The DVD is scanning.
#uninitialized	The DVD is not initialized.
#volumeInvalid	The DVD specified is not valid.
#volumeUknown	The DVD does not exist or there is no disc in the drive.
#systemSoftwareMissing	The DVD decoders are not installed.
#systemSoftwareBusy	The system software required to play the DVD is in use by another application.

See also

DVD

mediaStatus (RealMedia, Windows Media)

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.mediaStatus

// JavaScript syntax
memberOrSpriteObjRef.mediaStatus;
```

Description

RealMedia and Windows Media cast member and sprite property; allows you to get a symbol representing the state of the RealMedia or Windows Media stream. Read-only.

The value of this property can change during playback.

Valid values for this property are as follows:

- `#closed` indicates that the RealMedia or Windows Media cast member is not active. The `mediaStatus` value remains `#closed` until playback is initiated.
- `#connecting` indicates that a connection to the RealMedia or Windows Media stream is being established.
- `#opened` indicates that a connection to the RealMedia or Windows Media stream has been established and is open. This is a transitory state that is very quickly followed by `#buffering`.
- `#buffering` indicates that the RealMedia or Windows Media stream is being downloaded into the playback buffer. When buffering is complete (`percentBuffered` equals 100), the stream begins to play if the `pausedAtStart` property is `FALSE`. For more information, see `percentBuffered`.
- `#playing` indicates that the RealMedia or Windows Media stream is currently playing.
- `#seeking` indicates that play was interrupted by the `seek` command.
- `#paused` indicates that play has been interrupted, possibly by the user clicking the Stop button in the RealMedia or Windows Media viewer, or by a script invoking the `pause()` method.

- `#error` indicates that the stream could not be connected, buffered, or played for some reason. The `lastError` property reports the actual error.

Depending on the cast member's `state (RealMedia)` value, a different `mediaStatus` property value is returned. Each `mediaStatus` value corresponds to only one `state` value.

Example

The following examples show that the RealMedia element in sprite 1 and the cast member Real is playing.

```
-- Lingo syntax
put(sprite(1).mediaStatus)
put(member("RealDemo").mediaStatus)

// JavaScript syntax
put(sprite(1).mediaStatus);
put(member("RealDemo").mediaStatus);
```

See also

`state (RealMedia)`, `percentBuffered`, `lastError`

mediaXtraList

Usage

```
-- Lingo syntax
_player.mediaXtraList

// JavaScript syntax
_player.mediaXtraList;
```

Description

Player property; returns a linear list of all media Xtra extensions available to the Director player. Read-only.

Example

This statement displays in the Message window all media Xtra extensions that are available to the Director Player.

```
-- Lingo syntax
put(_player.mediaXtraList)

// JavaScript syntax
put(_player.mediaXtraList);
```

See also

`Media Types`, `Player`, `scriptingXtraList`, `toolXtraList`, `transitionXtraList`, `xtraList (Player)`

member

Usage

```
member(whichCastmember).texture(whichTexture).member
member(whichCastmember).model(whichModel).shader.texture.member
member(whichCastmember).model(whichModel).shaderList[shaderListIndex].textureList[textureL
istIndex].member
```

Description

3D texture property; if the texture's type is `#fromCastMember`, this property indicates the cast member that is used as the source for a texture.

This property can be tested and set.

If the texture's type is `#importedFromFile`, this property value is void and cannot be set. If the texture's type is `#fromImageObject`, this property value is `void`, but it can be set.

Example

This Lingo adds a new texture. The second statement shows that the cast member used to create the texture named gbTexture was member 16 of cast 1.

```
member("scene").newTexture("gbTexture", #fromCastmember, member(16, 1))
put member("scene").texture("gbTexture").member
-- (member 16 of castLib 1)
```

member (Cast)

Usage

```
-- Lingo syntax
castObjRef.member[memberNameOrNum]

// JavaScript syntax
castObjRef.member[memberNameOrNum]
```

Description

Cast library property; provides indexed or named access to the members of a cast library. Read-only.

The *memberNameOrNum* argument can be a string that specifies the cast member by name or an integer that specifies the cast member by number.

Example

The following example provides access to the second cast member in the cast library named Internal.

```
-- Lingo syntax
myMember = castLib("Internal").member[2]

// JavaScript syntax
var myMember = castLib("Internal").member[2];
```

See also

Cast Library

member (Movie)

Usage

```
-- Lingo syntax
_movie.member[memberNameOrNum]

// JavaScript syntax
_movie.member[memberNameOrNum];
```

Description

Movie property; provides indexed or named access to the members of a movie's cast library. Read-only.

The *memberNameOrNum* argument can be a string that specifies the cast member by name or an integer that specifies the cast member by number.

Example

The following statement accesses a cast member using both named and numbered access, and sets the result to the variable `myMember`.

```
-- Lingo syntax
myMember = _movie.member[2] -- using numbered access
myMember = _movie.member["Athlete"] -- using named access

// JavaScript syntax
var myMember = _movie.member[2]; // using numbered access;
var myMember = _movie.member["Athlete"]; // using named access;
```

See also

Movie

member (Sound Channel)

Usage

```
-- Lingo syntax
soundChannelObjRef.member

// JavaScript syntax
soundChannelObjRef.member;
```

Description

Sound Channel property; specifies the sound cast member currently playing in a sound channel. Read-only.

This property returns `VOID` (Lingo) or `null` (JavaScript syntax) if no sound is being played.

Example

This statement displays the name of the member of the sound playing in sound channel 2 in the Message window:

```
-- Lingo syntax
put(sound(2).member)

// JavaScript syntax
put(sound(2).member);
```

See also

Sound Channel

member (Sprite)

Usage

```
-- Lingo syntax
spriteObjRef.member
```

```
// JavaScript syntax
spriteObjRef.member;
```

Description

Sprite property; specifies a sprite's cast member and cast library. Read/write.

The `member` Sprite property differs from the `spriteNum` Sprite property, which specifies only the sprite's number to identify its location in the cast library but doesn't specify the cast library itself. The `member` property also differs from the Mouse object's `mouseMember` property, which does not specify a sprite's cast library.

When assigning a sprite's `member` property, use one of the following formats:

- Specify the full cast member and cast library description (*`spriteObjRef`*`.member = member(`*`intMemberNum`*`{, castLibraryNameOrNum})`).
- Specify the cast member name (*`spriteObjRef`*`.member = member("`*`stringMemberName`*"`)`).
- Specify the unique integer that includes all cast libraries and corresponds to the `mouseMember` property (*`spriteObjRef`*`.member = 132`).

If you use only the cast member name, Director finds the first cast member that has that name in all current cast libraries. If the name is duplicated in two cast libraries, only the first name is used.

To specify a cast member by number when there are multiple casts, use the `memberNum` Sprite property, which changes the member's position in its cast library without affecting the sprite's cast library (*`spriteObjRef`*`.memberNum = 10`).

The cast member assigned to a sprite channel is only one of that sprite's properties; other properties vary by the type of media element in that channel in the Score. For example, if you replace a bitmap with an unfilled shape by setting the `member` Sprite property, the shape sprite's `lineSize` property doesn't automatically change, and you probably won't see the shape.

Similar sprite property mismatches can occur if you change the member of a field sprite to a video. It's generally more useful and predictable to replace cast members with similar cast members. For example, replace bitmap sprites with bitmap cast members.

Example

This statement assigns cast member 3 of cast number 4 to sprite 15:

```
-- Lingo syntax
sprite(15).member = member(3, 4)

// JavaScript syntax
sprite(15).member = member(3, 4);
```

The following handler uses the mouseMember function with the `sprite.member` property to find if the mouse is over a particular sprite:

```
-- Lingo syntax
on exitFrame
    mm = _mouse.mouseMember
    target = sprite(1).member
    if (target = mm) then
        put("Above the hotspot.")
        _movie.go(_movie.frame)
    end if
end
```

```
// JavaScript syntax
function exitFrame() {
    var mm = _mouse.mouseMember;
    var target = sprite(1).member;
    if (target == mm) {
        put("Above the hotspot.");
        _movie.go(_movie.frame);
    }
}
```

See also

lineSize, mouseMember, Sprite, spriteNum

memorySize

Usage

```
the memorySize
```

Description

System property; returns the total amount of memory allocated to the program, whether in use or free memory. This property is useful for checking minimum memory requirements. The value is given in bytes.

In Windows, the value is the total physical memory available; on the Mac, the value is the entire partition assigned to the application.

Example

This statement checks whether the computer allocates more than 500K of memory and, if it does, displays an alert.

```
-- Lingo syntax
if the memorySize > 500 * 1024 then alert "There is enough memory to run this movie."

// JavaScript syntax
if ( _system.memorySize > 500 * 1024)
{
    _player.alert( "There is enough memory to run this movie.");
}
```

See also

freeBlock(), freeBytes(), ramNeeded(), size

meshDeform (modifier)

Usage

```
member(whichCastmember).model(whichModel).meshDeform.propertyName
```

Description

3D modifier; allows control over the various aspects of the referenced model's mesh structure. Once you have added the `#meshDeform` modifier (using the `addModifier` command) to a model you have access to the following properties of the `#meshDeform` modifier:

Note: *For more detailed information about the following properties see the individual property entries referenced in the see also section of this entry.*

- `face.count` returns the total number of faces in the referenced model.
- `mesh.count` returns the number of meshes in the referenced model.
- `mesh[index]` allows access to the properties of the specified mesh.

Example

The following statement displays the number of faces in the model named gbFace:

```
put member("3D World").model("gbFace").meshDeform.face.count
-- 432
```

The following statement displays the number of meshes in the model named gbFace:

```
put member("3D World").model("gbFace").meshDeform.mesh.count
-- 2
```

The following statement displays the number of faces in the second mesh of the model named gbFace:

```
put member("3D World").model("gbFace").meshDeform.mesh[2].face.count
-- 204
```

See also

mesh (property), addModifier

milliseconds

Usage

```
-- Lingo syntax
_system.milliseconds

// JavaScript syntax
_system.milliseconds;
```

Description

System property; returns the current time in milliseconds (1/1000 of a second). Read-only.

Counting begins from the time the computer is started.

Example

This statement converts milliseconds to seconds and minutes by dividing the number of milliseconds by 1000 and dividing that result by 60, and then sets the variable `currentMinutes` to the result:

```
-- Lingo syntax
currentSeconds = _system.milliseconds/1000
currentMinutes = currentSeconds/60

// JavaScript syntax
var currentSeconds = _system.milliseconds/1000;
var currentMinutes = currentSeconds/60;
```

The resolution accuracy of the count is machine and operating system dependent.

This handler counts the milliseconds and posts an alert if you've been working too long:

```
-- Lingo syntax
on idle
    if (_system.milliseconds > (1000 * 60 * 60 * 4)) then
        _player.alert("Take a break")
    end if
end

// JavaScript syntax
function idle() {
    if (_system.milliseconds > (1000 * 60 * 60 * 4)) {
        _player.alert("Take a break");
    }
}
```

See also

System

minSpeed

Usage

```
member(whichCastmember).modelResource(whichModelResource).emitter.minSpeed
```

Description

3D property; when used with a model resource whose type is `#particle`, allows you to get and set the minimum speed at which particles are emitted. Each particle's initial velocity is randomly selected between the emitter's `minSpeed` and `maxSpeed` properties.

The value is a floating-point number and must be greater than 0.0.

Example

This example sets the min speed of the particle to 4 in the cast member named 3Dobjects.

```
-- Lingo syntax
member("3Dobjects").modelResource("Particle01").emitter.minSpeed=4

// JavaScript syntax
member("3Dobjects").getPropRef("modelResource",10).emitter.minSpeed=4;
```

See also

maxSpeed, emitter

missingFonts

Usage

```
member(textCastMember).missingFonts
```

Description

Text cast member property; this property contains a list of the names of the fonts that are referenced in the text, but not currently available on the system.

This allows the developer to determine during run time if a particular font is available or not.

This property can be tested but not set.

Example

This example shows a list of the names of the fonts that are referenced in the text, but not currently available on the system.

```
-- Lingo syntax
put member(4).missingFonts

// JavaScript syntax
put(member(4).missingFonts);
```

See also

substituteFont

mode (emitter)

Usage

```
member(whichCastmember).modelResource(whichModelResource).emitter.mode
```

Description

3D property; when used with a model resource whose type is `#particle`, allows you to both get and set the `mode` property of the resource's particle emitter.

This property can have the value `#burst` or `#stream` (default). A `mode` value of `#burst` causes all particles to be emitted at the same time, while a value of `#stream` causes a group of particles to be emitted at each frame. The number of particles emitted in each frame is determined using the following equation:

```
particlesPerFrame = resourceObject.emitter.numParticles (resourceObject.lifetime x
millisecondsPerRenderedFrame)
```

Example

In this example, ThermoSystem is a model resource of the type `#particle`. This statement sets the `emitter.mode` property of ThermoSystem to `#burst`, which causes the particles of ThermoSystem to appear in bursts. To create a single burst of particles, set `emitter.mode = #burst` and `emitter.loop = 0`.

```
member("Fires").modelResource("ThermoSystem").emitter.mode = #burst
```

See also

emitter

mode (collision)

Usage

```
member(whichCastmember).model(whichModel).collision.mode
```

Description

3D collision modifier property; indicates the geometry to be used in the collision detection algorithm. Using simpler geometry such as the bounding sphere leads to better performance. The possible values for this property are:

- `#mesh` uses the actual mesh geometry of the model's resource. This gives one-triangle precision and is usually slower than `#box` or `#sphere`.
- `#box` uses the bounding box of the model. This is useful for objects that can fit more tightly in a box than in a sphere, such as a wall.
- `#sphere` is the fastest mode, because it uses the bounding sphere of the model. This is the default value for this property.

Example

These statements add the collision modifier to the model named `your Model` and set the `mode` property to `#mesh`:

```
member("3d").model("yourModel").addModifier(#collision)
member("3d").model("yourModel").collision.mode = #mesh
```

model

Usage

```
member(whichCastmember).model(whichModel)
member(whichCastmember).model[index]
member(whichCastmember).model.count
member(whichCastmember).model(whichModel).propertyName
member(whichCastmember).model[index].propertyName
```

Description

3D command; returns the model found within the referenced cast member that has the name specified by *`whichModel`*, or is found at the index position specified by *`index`*. If no model exists for the specified parameter, the command returns `void`. As `model.count`, the command returns the number of models found within the referenced cast member. This command also allows access to the specified model's properties.

Model name comparisons are not case-sensitive. The index position of a particular model may change when objects at lower index positions are deleted.

If no model is found that uses the specified name or no model is found at the specified index position then this command returns void.

Example

This statement stores a reference to the model named Player Avatar in the variable `thismodel`:

```
thismodel = member("3DWorld").model("Player Avatar")
```

This statement stores a reference to the eighth model of the cast member named 3DWorld in the variable `thismodel`.

```
thismodel = member("3DWorld").model[8]
```

This statement shows that there are four models in the member of sprite 1..

```
put sprite(1).member.model.count
-- 4
```

modelA

Usage

```
collisionData.modelA
```

Description

3D `collisionData` property; indicates one of the models involved in a collision, the other model being `modelB`.

The `collisionData` object is sent as an argument with the `#collideWith` and `#collideAny` events to the handler specified in the `registerForEvent`, `registerScript`, and `setCollisionCallback` commands.

The `#collideWith` and `#collideAny` events are sent when a collision occurs between models to which collision modifiers have been added. The `resolve` property of the models' modifiers must be set to `TRUE`.

This property can be tested but not set.

Example

This example has three parts. The first part is the first line of code, which registers the `#putDetails` handler for the `#collideAny` event. The second part is the `#putDetails` handler. When two models in the cast member named MyScene collide, the `#putDetails` handler is called and the `collisionData` argument is sent to it. This handler displays the `modelA` and `modelB` properties of the `collisionData` object in the message window. The third part of the example shows the results from the message window. These show that the model named GreenBall was modelA and the model named YellowBall was `modelB` in the collision.

```
member("MyScene").registerForEvent(#collideAny, #putDetails, 0)
on putDetails me, collisionData
    put collisionData.modelA
    put collisionData.modelB
end
-- model("GreenBall")
-- model("YellowBall")
```

See also

`registerScript()`, `registerForEvent()`, `sendEvent`, `modelB`, `setCollisionCallback()`

modelB

Usage

```
collisionData.modelB
```

Description

3D `collisionData` property; indicates one of the models involved in a collision, the other model being `modelA`.

The `collisionData` object is sent as an argument with the `#collideWith` and `#collideAny` events to the handler specified in the `registerForEvent`, `registerScript`, and `setCollisionCallback` commands.

The `#collideWith` and `#collideAny` events are sent when a collision occurs between models to which collision modifiers have been added. The `resolve` property of the models' modifiers must be set to `TRUE`.

This property can be tested but not set.

Example

This example has three parts. The first part is the first line of code, which registers the #putDetails handler for the #collideAny event. The second part is the #putDetails handler. When two models in the cast member named MyScene collide, the #putDetails handler is called and the collisionData argument is sent to it. This handler displays the modelA and modelB properties of the collisionData object in the message window. The third part of the example shows the results from the message window. These show that the model named GreenBall was modelA and the model named YellowBall was modelB in the collision.

```
member("MyScene").registerForEvent(#collideAny, #putDetails, 0)
on putDetails me, collisionData
    put collisionData.modelA
    put collisionData.modelB
end
-- model("GreenBall")
-- model("YellowBall")
```

See also

registerScript(), registerForEvent(), sendEvent, modelA, collisionNormal, setCollisionCallback()

modelResource

Usage

```
member(whichCastmember).modelResource(whichModelResource)
member(whichCastmember).modelResource[index]
member(whichCastmember).modelResource.count
member(whichCastmember).modelResource(whichModelResource).propertyName
member(whichCastmember).modelResource[index].propertyName
```

Description

3D command; returns the model resource found within the referenced cast member that has the name specified by *whichModelResource*, or is found at the index position specified by the *index* parameter. If no model resource exists for the specified parameter, the command returns void. As modelResource.count, the command returns the number of model resources found within the referenced cast member. This command also allows access to the specified model resource's properties.

Model resource name string comparisons are not case-sensitive. The index position of a particular model resource may change when objects at lower index positions are deleted.

Example

This statement shows first model resource of the cast member named 3Dobjects.

```
-- Lingo syntax
putmember("3Dobjects").modelResource[1]

// JavaScript syntax
put( member("3Dobjects").getPropRef("modelResource",1));
```

modified

Usage

```
-- Lingo syntax
memberObjRef.modified

// JavaScript syntax
memberObjRef.modified;
```

Description

Member property; indicates whether a cast member has been modified since it was read from a movie file. Read-only.

- When the `modified` property is `TRUE` (`1`), the cast member has been modified since it was read from the movie file.
- When the `modified` property is `FALSE` (`0`), the cast member has not been modified since it was read from the movie file.

Example

This statement tests whether the cast member Introduction has been modified since it was read from the movie file:

```
-- Lingo syntax
if (member("Introduction").modified) then
    _player.alert("Introduction has been modified")
else
    _player.alert("Introduction has not been modified")
end if

// JavaScript syntax
if (member("Introduction").modified) {
    _player.alert("Introduction has been modified");
}
else {
    _player.alert("Introduction has not been modified");
}
```

See also

Member

modifiedBy

Usage

```
-- Lingo syntax
memberObjRef.modifiedBy

// JavaScript syntax
memberObjRef.modifiedBy;
```

Description

Member property; records the name of the user who last edited the cast member. Read-only.

The name is taken from the user name information provided during Director installation. You can change this information in the Director General Preferences dialog box.

This property is useful for tracking and coordinating Director projects with more than one author, and may also be viewed in the Property inspector's Member tab.

Example

This statement displays the name of the person who last modified cast member 1:

```
-- Lingo syntax
put(member(1).modifiedBy)

// JavaScript syntax
put(member(1).modifiedBy);
```

See also

Member

modifiedDate

Usage

```
-- Lingo syntax
memberObjRef.modifiedDate

// JavaScript syntax
memberObjRef.modifiedDate;
```

Description

Member property; indicates the date and time that the cast member was last changed, using the system time on the authoring computer. Read-only.

This property is useful for tracking and coordinating Director projects. It can also be viewed in the Property inspector's Member tab and the Cast window list view.

Example

This statement displays the date of the last change to cast member 1:

```
-- Lingo syntax
put(member(1).modifiedDate)

// JavaScript syntax
put(member(1).modifiedDate);
```

See also

Member

modifier

Usage

```
member(whichCastmember).model(whichModel).modifier
member(whichCastmember).model(whichModel).modifier.count
```

Description

3D model property; returns a list of modifiers that are attached to the specified model. As `modifier.count`, the command returns the number of modifiers attached to the model.

If both the `toon` and `inker` modifiers are applied to a model, only the first one that was added to the model is returned.

This property can be tested but not set. Use the `addModifier` and `removeModifier` commands to add and remove modifiers from models.

Example

This statement shows which modifiers are attached to the model named Sphere01 of the cast member named 3Dobjects.

```
-- Lingo syntax
put member("3Dobjects").model("Sphere01").modifier

// JavaScript syntax
put( member("3Dobjects").getPropRef("model",2).modifier);
```

See also

`modifier[]`, `modifiers`, `addModifier`, `removeModifier`

modifier[]

Usage

```
member(whichCastmember).model(whichModel).modifier[index]
```

Description

3D model property; returns the type of the modifier found at the position specified by *`index`* within the model's attached modifier list. The value returned is a symbol.

If no modifier is found at the specified position then this property's value is void.

To obtain information about a model's attached modifier list use the `modifier` property.

Direct access into an attached modifier's properties is not supported through the use of this command.

Example

```
put member("3d world").model("box").modifier[1]
-- #lod
```

See also

`modifier`, `modifiers`, `addModifier`, `removeModifier`

modifiers

Usage

```
getRendererServices().modifiers
```

Description

Global 3D property; returns a list of modifiers available to models within 3D cast members.

Example

This statement returns the list of all currently available modifiers:

```
-- Lingo syntax
put getRendererServices().modifiers

// JavaScript syntax
put(getRendererServices().modifiers);
```

See also

getRendererServices(), addModifier

mostRecentCuePoint

Usage

```
-- Lingo syntax
spriteObjRef.mostRecentCuePoint

// JavaScript syntax
spriteObjRef.mostRecentCuePoint;
```

Description

Sound channel and sprite property; for sound sprites, QuickTime digital video, and Xtra extensions that support cue points, indicates the number that identifies the most recent cue point passed in the sprite or sound. The value is the cue point's ordinal number. If no cue points have been passed, the value is 0.

Shockwave Audio (SWA) sounds can appear as sprites in sprite channels, but they play sound in a sound channel. It is recommended that you refer to SWA sound sprites by their sprite channel number rather than their sound channel number.

Example

This statement tells the Message window to display the number for the most recent cue point passed in the sprite in sprite channel 1:

```
-- Lingo syntax
put sprite(1).mostRecentCuePoint

// JavaScript syntax
put(sprite(1).mostRecentCuePoint);
```

This statement returns the ordinal number of the most recently passed cue point in the currently playing sound in sound channel 2:

```
-- Lingo syntax
put sound(2).mostRecentCuePoint

// JavaScript syntax
put(sound(2).mostRecentCuePoint);
```

See also

cuePointNames, isPastCuePoint(), cuePointTimes, on cuePassed

motion

Usage

```
member(whichCastmember).motion(whichMotion)
member(whichCastmember).motion[index]
member(whichCastmember).motion.count
```

Description

3D command; returns the motion found within the referenced cast member that has the name specified by *whichMotion*, or is found at the index position specified by the *index*. As `motion.count`, this property returns the total number of motions found within the cast member.

Object name string comparisons are not case-sensitive. The index position of a particular motion may change when objects at lower index positions are deleted.

If no motion is found that uses the specified name or no motion is found at the specified index position then this command returns void.

Example

This statement displays the first motion found within the referenced cast member.

```
-- Lingo syntax
put member("3Dobjects").motion[1]

// JavaScript syntax
put(member("3Dobjects").getPropRef("motion",1));
```

See also

duration (3D), map (3D)

motionQuality

Usage

```
-- Lingo syntax
spriteObjRef.motionQuality

// JavaScript syntax
spriteObjRef.motionQuality;
```

Description

QuickTime VR sprite property; the codec quality used when the user clicks and drags the QuickTime VR sprite. The property's value can be `#minQuality`, `#maxQuality`, or `#normalQuality`.

This property can be tested and set.

Example

This statement sets the motionQuality of sprite 1 to `#minQuality`.

```
-- Lingo syntax
sprite(1).motionQuality = #minQuality

// JavaScript syntax
sprite(1).motionQuality = symbol("minQuality");
```

mouseChar

Usage

```
-- Lingo syntax
_mouse.mouseChar

// JavaScript syntax
_mouse.mouseChar;
```

Description

Mouse property; for field sprites, contains the number of the character that is under the pointer when the property is called. Read-only.

The count is from the beginning of the field. If the mouse pointer is not over a field or is in the gutter of a field, the result is -1.

The value of `mouseChar` can change in a handler or loop. If a handler or loop uses this property multiple times, it's usually a good idea to call the property once and assign its value to a local variable.

Example

This statement determines whether the pointer is over a field sprite and changes the content of the field cast member Instructions to "Please point to a character." when it is not:

```
-- Lingo syntax
if (_mouse.mouseChar = -1) then
    member("Instructions").text = "Please point to a character."
end if

// JavaScript syntax
if (_mouse.mouseChar == -1) {
    member("Instructions").text = "Please point to a character.";
}
```

This statement assigns the character under the pointer in the specified field to the variable `currentChar`:

```
-- Lingo syntax
currentChar = member(_mouse.mouseMember).char[_mouse.mouseChar]

// JavaScript syntax
var currentChar = member(_mouse.mouseMember).getProp("char",_mouse.mouseChar);
```

See also

Mouse, mouseItem, mouseLine

mouseDown

Usage

```
-- Lingo syntax
_mouse.mouseDown

// JavaScript syntax
_mouse.mouseDown;
```

Description

Mouse property; indicates whether the mouse button is currently being pressed (TRUE) or not (FALSE). Read-only.

Example

The following mouseEnter handler that is attached to a sprite calls one handler if the mouse is not down when the mouse enters the sprite, and calls a different handler if the mouse is not down when the mouse enters the sprite.

```
-- Lingo syntax
on mouseEnter
    if (_mouse.mouseDown) then
        runMouseDownScript
    else
        runMouseUpScript
    end if
end

// JavaScript syntax
function mouseEnter() {
    if (_mouse.mouseDown) {
        runMouseDownScript();
    }
    else {
        runMouseUpScript();
    }
}
```

See also

Mouse, on mouseDown (event handler), mouseH, mouseUp, on mouseUp (event handler), mouseV

mouseDownScript

Usage

```
the mouseDownScript
```

Description

System property; specifies the Lingo that is executed when the mouse button is pressed. The Lingo is written as a string, surrounded by quotation marks and can be a simple statement or a calling script for a handler. The default value is EMPTY, which means that the mouseDownScript property has no Lingo assigned to it.

When the mouse button is pressed and the mouseDownScript property is defined, Lingo executes the instructions specified for the mouseDownScript property first. No other on mouseDown handlers are executed, unless the instructions include the pass command so that the mouseDown message can be passed to other objects in the movie.

Setting the mouseDownScript property performs the same function as the when keyDown then command in earlier versions of Director.

To turn off the instructions you've specified for the mouseDownScript property, use the statement set the mouseDownScript to empty.

This property can be tested and set.

Example

This statement specifies the script that is executed when the user clicks the mouse button.

```
-- Lingo syntax
the mouseDownScript = "go to the frame"

// JavaScript syntax
_system.mouseDownScript = "_movie.go(_movie.frame)";
```

See also

stopEvent(), mouseUpScript, on mouseDown (event handler), on mouseUp (event handler)

mouseH

Usage

```
-- Lingo syntax
_mouse.mouseH

// JavaScript syntax
_mouse.mouseH;
```

Description

Mouse property; indicates the horizontal position of the mouse pointer. Read-only.

The value of `mouseH` is the number of pixels the cursor is positioned from the left edge of the Stage.

The `mouseH` property is useful for moving sprites to the horizontal position of the mouse pointer and checking whether the pointer is within a region of the Stage. Using the `mouseH` and `mouseV` properties together, you can determine the cursor's exact location.

Example

This handler moves sprite 10 to the mouse pointer location and updates the Stage when the user clicks the mouse button:

```
-- Lingo syntax
on mouseDown
    sprite(10).locH = _mouse.mouseH
    sprite(10).locV = _mouse.mouseV
end

// JavaScript syntax
function mouseDown() {
    sprite(10).locH = _mouse.mouseH;
    sprite(10).locV = _mouse.mouseV;
}
```

This statement tests whether the pointer is more than 10 pixels to the right or left of a starting point and, if it is, sets the variable `Far` to `TRUE`:

```
-- Lingo syntax
startH = 7
if (abs(_mouse.mouseH - startH) > 10) then
    Far = TRUE
end if

// JavaScript syntax
var startH = 7;
if (Math.abs(_mouse.mouseH - startH) > 10) {
    var Far = true;
```

```
}
```

See also

locH, locV, Mouse, mouseLoc, mouseV

mouseItem

Usage

```
-- Lingo syntax
_mouse.mouseItem

// JavaScript syntax
_mouse.mouseItem;
```

Description

Mouse property; contains the number of the item under the pointer when the property is called and the pointer is over a field sprite. Read-only.

An item is any sequence of characters delimited by the current delimiter as set by `the itemDelimiter` property. Counting starts at the beginning of the field. If the mouse pointer is not over a field, the result is -1.

The value of the `mouseItem` property can change in a handler or loop. If a handler or loop relies on the initial value of `mouseItem` when the handler or loop begins, call this property once and assign its value to a local variable.

Example

This statement determines whether the pointer is over a field sprite and changes the content of the field cast member Instructions to "Please point to an item." when it is not:

```
-- Lingo syntax
if (mouse.mouseItem = -1) then
    member("Instructions").text = "Please point to an item."
end if

// JavaScript syntax
if (_mouse.mouseItem == -1) {
    member("Instructions").text = "Please point to an item.";
}
```

This statement assigns the item under the pointer in the specified field to the variable `currentItem`:

```
-- Lingo syntax
currentItem = member(_mouse.mouseMember).item[_mouse.mouseItem]

// JavaScript syntax
var currentItem = member(_mouse.mouseMember).getProp("item",_mouse.mouseItem);
```

See also

itemDelimiter, Mouse, mouseChar, mouseLine, mouseWord

mouseLevel

Usage

```
-- Lingo syntax
spriteObjRef.mouseLevel

// JavaScript syntax
spriteObjRef.mouseLevel;
```

Description

QuickTime sprite property; controls how Director passes mouse clicks on a QuickTime sprite to QuickTime. The ability to pass mouse clicks within the sprite's bounding rectangle can be useful for interactive media such as QuickTime VR. The `mouseLevel` sprite property can have these values:

- `#controller`—Passes clicks only on the movie controller to QuickTime. Director responds only to mouse clicks that occur outside the controller. This is the standard behavior for QuickTime sprites other than QuickTime VR.
- `#all`—Passes all mouse clicks within the sprite's bounding rectangle to QuickTime. No clicks pass to other Lingo handlers.
- `#none`—Does not pass any mouse clicks to QuickTime. Director responds to all mouse clicks.
- `#shared`—Passes all mouse clicks within a QuickTime VR sprite's bounding rectangle to QuickTime and then passes these events to Lingo handlers. This is the default value for QuickTime VR.

This property can be tested and set.

Example

This frame script checks to see if the name of the QuickTime sprite in channel 5 contains the string "QTVR." If it does, this script sets `mouseLevel` to `#all`; otherwise, it sets `mouseLevel` to `#none`.

```
on prepareFrame
    if sprite(5).member.name contains "QTVR" then
        sprite(5).mouseLevel = #all
    else
        sprite(5).mouseLevel = #none
    end if
end

// JavaScript syntax
function prepareFrame() {
    var nm = sprite(5).member.name;
    var nmStr = nm.indexOf("QTVR");
    if (nmStr != -1) {
        sprite(5).mouseLevel = symbol("all");
    } else {
        sprite(5).mouseLevel = symbol("none");
    }
}
```

mouseLine

Usage

```
-- Lingo syntax
_mouse.mouseLine
```

```
// JavaScript syntax
_mouse.mouseLine;
```

Description

Mouse property; contains the number of the line under the pointer when the property is called and the pointer is over a field sprite. Read-only.

Counting starts at the beginning of the field; a line is defined by Return delimiter, not by the wrapping at the edge of the field. When the mouse pointer is not over a field sprite, the result is -1.

The value of the `mouseLine` property can change in a handler or loop. If a handler or loop uses this property multiple times, it's usually a good idea to call the property once and assign its value to a local variable.

Example

This statement determines whether the pointer is over a field sprite and changes the content of the field cast member Instructions to "Please point to a line." when it is not:

```
-- Lingo syntax
if (_mouse.mouseLine = -1) then
    member("Instructions").text = "Please point to a line."
end if

// JavaScript syntax
if (_mouse.mouseLine == -1) {
    member("Instructions").text = "Please point to a line.";
}
```

This statement assigns the contents of the line under the pointer in the specified field to the variable `currentLine`:

```
-- Lingo syntax
currentLine = member(_mouse.mouseMember).line[_mouse.mouseLine]

// JavaScript syntax
var currentLine = member(_mouse.mouseMember).getProp("line",_mouse.mouseLine);
```

See also

Mouse, mouseChar, mouseItem, mouseWord

mouseLoc

Usage

```
-- Lingo syntax
_mouse.mouseLoc

// JavaScript syntax
_mouse.mouseLoc;
```

Description

Mouse property; returns the current position of the mouse as a point(). Read-only.

The point location is given as two coordinates, with the horizontal location first, then the vertical location.

Example

The following statement displays the current position of the mouse.

```
-- Lingo syntax
trace(_mouse.mouseLoc)

// JavaScript syntax
trace(_mouse.mouseLoc);
```

See also

Mouse, mouseH, mouseV

mouseMember

Usage

```
-- Lingo syntax
_mouse.mouseMember

// JavaScript syntax
_mouse.mouseMember;
```

Description

Mouse property; returns the cast member assigned to the sprite that is under the pointer when the property is called. Read-only.

When the pointer is not over a sprite, this property returns the result `VOID` (Lingo) or `null` (JavaScript syntax).

You can use this property to make a movie perform specific actions when the pointer rolls over a sprite and the sprite uses a certain cast member.

The value of the `mouseMember` property can change frequently. To use this property multiple times in a handler with a consistent value, assign the `mouseMember` value to a local variable when the handler starts and use the variable.

Example

The following statement checks whether the cast member Off Limits is the cast member assigned to the sprite under the pointer and displays an alert if it is. This example shows how you can specify an action based on the cast member assigned to the sprite.

```
-- Lingo syntax
if (_mouse.mouseMember = member("Off Limits")) then
    _player.alert("Stay away from there!")
end if

// JavaScript syntax
if (_mouse.mouseMember == member("Off Limits")) {
    _player.alert("Stay away from there!");
}
```

This statement assigns the cast member of the sprite under the pointer to the variable `lastMember`:

```
-- Lingo syntax
lastMember = _mouse.mouseMember

// JavaScript syntax
var lastMember = _mouse.mouseMember;
```

See also

castLibNum, Mouse

mouseOverButton

Usage

```
-- Lingo syntax
spriteObjRef.mouseOverButton
```

```
// JavaScript syntax
spriteObjRef.mouseOverButton;
```

Description

Flash sprite property; indicates whether the mouse pointer is over a button in a Flash movie sprite specified by the *whichFlashSprite* parameter (TRUE), or whether the mouse pointer is outside the bounds of the sprite or the mouse pointer is within the bounds of the sprite but over a nonbutton object, such as the background (FALSE).

This property can be tested but not set.

Example

This frame script checks to see if the mouse pointer is over a navigation button in the Flash movie in sprite 3. If the mouse pointer is over the button, the script updates a text field with an appropriate message; otherwise, the script clears the message.

```
-- Lingo syntax
on enterFrame
    case sprite(3).mouseOverButton of
        TRUE:
        member("Message Line").text = "Click here to go to the next page."
        FALSE:
        member("Message Line").text = " "
    end case
    _movie.updatestage()
end

// JavaScript syntax
function enterFrame() {
    switch(sprite(3).mouseOverButton)
    {
        case 1:
            member("Message Line").text = "Click here to go to the next page.";
            break;
        case 0:
            member("Message Line").text = " ";
            break;
    }
    _movie.updatestage();
}
```

mouseUp

Usage

```
-- Lingo syntax
_mouse.mouseUp
```

```
// JavaScript syntax
_mouse.mouseUp;
```

Description

Mouse property; indicates whether the mouse button is released (TRUE) or is being pressed (FALSE). Read-only.

Example

This handler causes the movie to run as long as the user presses the mouse button. The playhead stops when the user releases the mouse button.

```
-- Lingo syntax
on exitFrame me
    if (_mouse.mouseUp) then
        _movie.go(_movie.frame)
    end if
end

// JavaScript syntax
function exitFrame() {
     if (_mouse.mouseUp) {
        _movie.go(_movie.frame);
    }
}
```

This statement instructs Lingo to exit the repeat loop or handler it is in when the user releases the mouse button:

```
-- Lingo syntax
if (_mouse.mouseUp) then exit

// JavaScript syntax
if (_mouse.mouseUp) {
    return;
}
```

See also

Mouse, mouseDown, mouseH, mouseV

mouseUpScript

Usage

```
the mouseUpScript
```

Description

System property; determines the Lingo that is executed when the mouse button is released. The Lingo is written as a string, surrounded by quotation marks, and can be a simple statement or a calling script for a handler.

When the mouse button is released and the mouseUpScript property is defined, Lingo executes the instructions specified for the mouseUpScript property first. Unless the instructions include the pass command so that the mouseUp message can be passed on to other objects in the movie, no other on mouseUp handlers are executed.

When the instructions you've specified for the mouseUpScript property are no longer appropriate, turn them off by using the statement set the mouseUpScript to empty.

Setting the mouseUpScript property accomplishes the same thing as using the when mouseUp then command that appeared in earlier versions of Director.

This property can be tested and set. The default value is EMPTY.

Example

When this statement is in effect and the movie is paused, the movie always continues whenever the user releases the mouse button:

```
the mouseUpScript = "go to the frame +1"
```

With this statement, when the user releases the mouse button after clicking anywhere on the Stage, the movie beeps:

```
the mouseUpScript = "if the clickOn = 0 then beep"
```

This statement sets `mouseUpScript` to the custom handler *`myCustomHandler`*. A Lingo custom handler must be enclosed in quotation marks when used with the `mouseUpScript` property.

```
the mouseUpScript = "myCustomHandler"
```

See also

`stopEvent()`, `mouseDownScript`, `on mouseDown (event handler)`, `on mouseUp (event handler)`

mouseV

Usage

```
-- Lingo syntax
_mouse.mouseV

// JavaScript syntax
_mouse.mouseV;
```

Description

Mouse property; indicates the vertical position of the mouse cursor, in pixels, from the top of the Stage. Read-only.

The value of this property increases as the cursor moves down and decreases as the cursor moves up.

The `mouseV` property is useful for moving sprites to the vertical position of the mouse cursor and checking whether the cursor is within a region of the Stage. Using the `mouseH` and `mouseV` properties together, you can identify the cursor's exact location.

Example

This handler moves sprite 1 to the mouse pointer location and updates the Stage when the user clicks the mouse button:

```
-- Lingo syntax
on mouseDown
    sprite(1).locH = _mouse.mouseH
    sprite(1).locV = _mouse.mouseV
end

// JavaScript syntax
function mouseDown() {
    sprite(1).locH = _mouse.mouseH;
    sprite(1).locV = _mouse.mouseV;
}
```

This statement tests whether the pointer is more than 10 pixels above or below a starting point and, if it is, sets the variable `vFar` to `TRUE`:

```
-- Lingo syntax
startV = 7
```

```
if (abs(_mouse.mouseV - startV) > 10) then
    vFar = TRUE
end if

// JavaScript syntax
var startV = 7
if (Math.abs(_mouse.mouseV - startV) > 10) {
    var vFar = true;
}
```

See also

locH, locV, Mouse, mouseH, mouseLoc

mouseWord

Usage

```
-- Lingo syntax
_mouse.mouseWord

// JavaScript syntax
_mouse.mouseWord;
```

Description

Mouse property; contains the number of the word under the pointer when the property is called and when the pointer is over a field sprite. Read-only.

Counting starts from the beginning of the field. When the mouse is not over a field, the result is -1.

The value of the `mouseWord` property can change in a handler or loop. If a handler or loop uses this property multiple times, it's usually a good idea to call the function once and assign its value to a local variable.

Example

This statement determines whether the pointer is over a field sprite and changes the content of the field cast member Instructions to "Please point to a word." when it is not:

```
-- Lingo syntax
if (_mouse.mouseWord = -1) then
    member("Instructions").text = "Please point to a word."
else
    member("Instructions").text = "Thank you."
end if

// JavaScript syntax
if (_mouse.mouseWord == -1) {
    member("Instructions").text = "Please point to a word.";
}
else {
    member("Instructions").text = "Thank you.";
}
```

This statement assigns the number of the word under the pointer in the specified field to the variable `currentWord`:

```
-- Lingo syntax
currentWord = member(_mouse.mouseMember).word[_mouse.mouseWord]

// JavaScript syntax
```

```
var currentWord = member(_mouse.mouseMember).getProp("word",_mouse.mouseWord);
```

See also

Mouse, mouseChar, mouseItem

moveableSprite

Usage

```
sprite(whichSprite).moveableSprite
the moveableSprite of sprite whichSprite
```

Description

Sprite property; indicates whether a sprite can be moved by the user (TRUE) or not (FALSE).

You can make a sprite moveable by using the Moveable option in the Score. However, to control whether a sprite is moveable and to turn this condition on and off as needed, use Lingo. For example, to let users drag sprites one at a time and then make the sprites unmoveable after they are positioned, turn the moveableSprite sprite property on and off at the appropriate times.

Note: *For more customized control such as snapping back to the origin or animating while dragging, create a behavior to manage the additional functionality.*

This property can be tested and set.

Example

This handler makes sprites in channel 5 moveable:

```
on spriteMove
     sprite(5).moveableSprite = TRUE
end
```

This statement checks whether a sprite is moveable and, if it is not, displays a message:

```
if sprite(13).moveableSprite = FALSE thenmember("Notice").text = "You can't drag this item
by using the mouse."
```

See also

mouseLoc

movie

Usage

```
-- Lingo syntax
windowObjRef.movie

// JavaScript syntax
windowObjRef.movie;
```

Description

Window property; returns a reference to the movie object that is playing in a specified window. Read-only.

Example

This statement displays in the Message window the movie object that is playing in the window named Empires:

```
-- Lingo syntax
trace(window("Empires").movie)

// JavaScript syntax
trace(window("Empires").movie);
```

See also

Window

multiSound

Usage

```
the multiSound
```

Description

System property; specifies whether the system supports more than one sound channel and requires a Windows computer to have a multichannel sound card (TRUE) or not (FALSE).

Example

This statement plays the sound file Music in sound channel 2 if the computer supports more than one sound channel:

```
if the multiSound then sound playFile 2, "Music.wav"
```

name

Usage

```
-- Lingo syntax
castObjRef.name
memberObjRef.name
_movie.name
windowObjRef.name

// JavaScript syntax
castObjRef.name;
memberObjRef.name;
_movie.name;
windowObjRef.name;
```

Description

Cast, Member, Movie, and Window property; returns or sets the name of an object. Read/write for Cast, Member, and Window objects, read-only for Movie objects.

Example

This statement changes the name of the cast member 1 to Newname.

```
-- Lingo syntax
member(1).name="Newname"

// JavaScript syntax
```

```
member(1).name="Newname";
```

See also

Cast Library, Member, Movie, Window

name (3D)

Usage

```
member(whichCastmember).texture(whichTexture).name
member(whichCastmember).shader(whichShader).name
member(whichCastmember).motion(whichMotion).name
member(whichCastmember).modelResource(whichModelResource).name
member(whichCastmember).model(whichModel).name
member(whichCastmember).light(whichLight).name
member(whichCastmember).camera(whichCamera).name
member(whichCastmember).group(whichGroup).name
node.name
```

Description

3D property; when used with an object reference, allows you to get the name of the referenced object. You can only get the name; the name can't be changed.

All names must be unique. If created through Lingo, the name returned is the name given in the constructor function. If created through a 3D-authoring program the name returned may be the name of the model.

Example

This statement sets the name of the fifth camera in the cast member TableScene to BirdCam:

```
member("TableScene").camera[5].name = "BirdCam"
```

name (menu property)

Usage

```
the name of menu(whichMenu)
the name of menu whichMenu
```

Description

Menu property; returns a string containing the name of the specified menu number.

This property can be tested but not set. Use the `installMenu` command to set up a custom menu bar.

Note: *Menus are not available in Shockwave Player.*

Example

This statement assigns the name of menu number 1 to the variable `firstMenu`:

```
firstMenu = menu(1).name
```

The following handler returns a list of menu names, one per line:

```
on menuList
    theList = []
    repeat with i = 1 to the number of menus
```

```
        theList[i] = the name of menu i
    end repeat
    return theList
end menuList
```

See also

number (menus), name (menu item property)

name (menu item property)

Syntax

```
the name of menuItem(whichItem) of menu(whichMenu)
the name of menuItem whichItem of menu whichMenu
```

Description

Menu property; determines the text that appears in the menu item specified by *whichItem* in the menu specified by *whichMenu*. The *whichItem* argument is either a menu item name or a menu item number; *whichMenu* is either a menu name or a menu number.

This property can be tested and set.

Note: *Menus are not available in Shockwave Player.*

Example

This statement sets the variable `itemName` to the name of the eighth item in the Edit menu:

```
set itemName = the name of menuItem(8) of menu("Edit")
```

This statement causes a specific filename to follow the word *Open* in the File menu:

```
the name of menuItem("Open") of menu("fileMenu") = "Open" && fileName
```

See also

name (menu property), number (menu items)

name (Sprite)

Usage

```
-- Lingo syntax
spriteObjRef.name

// JavaScript syntax
spriteObjRef.name;
```

Description

Sprite property; identifies the name of a sprite. Read/write during a Score recording session only.

Unlike sprite display properties such as `backColor` and `blend`, a sprite `name` cannot be a scripted sprite. This means that the `name` can only be set during a Score recording session—between calls to the Movie object's `beginRecording()` and `endRecording()` methods. You can only set the `name` if `beginRecording()` is called on or before a frame in the Score that contains the sprite.

Note: *Starting a Score recording session using* `beginRecording()` *resets the properties of all scripted sprites and sprite channels.*

If you use script to create a new sprite during a Score recording session and you use `updateFrame()` to apply the sprite data to the session, you cannot set the sprite's name until you go back to the frame in which the sprite was created. Use a method such as `go()` to go back to a specific frame.

Example

This statement sets the name of sprite 5 to Background Sound:

```
-- Lingo syntax
sprite(5).name = "Background Sound"

// JavaScript syntax
sprite(5).name = "Background Sound";
```

See also

`beginRecording()`, `endRecording()`, `go()`, `Sprite`, `updateFrame()`

name (Sprite Channel)

Usage

```
-- Lingo syntax
spriteChannelObjRef.name

// JavaScript syntax
spriteChannelObjRef.name;
```

Description

Sprite Channel property; identifies the name of a sprite channel. Read/write during a Score recording session only.

Set the `name` of a sprite channel during a Score recording session—between calls to the Movie object's `beginRecording()` and `endRecording()` methods.

Note: *Starting a Score recording session using* `beginRecording()` *resets the properties of all scripted sprites and sprite channels.*

Unlike a Sprite object's `name` property, which can only be set on or after a frame in which a sprite appears in the Score, a Sprite Channel object's `name` property can be set on an empty channel. This means that you do not need to call `updateFrame()` before setting the `name` of the sprite channel.

A change to a sprite channel's name using script is not reflected in the Score window.

Example

This statement sets the name of sprite channel 6 to Kite String during a Score recording session:

```
-- Lingo syntax
on mouseDown
    _movie.beginRecording()
    channel(6).name = "Kite string"
    _movie.endRecording()
end

// JavaScript syntax
function mouseDown() {
```

```
    _movie.beginRecording();
    channel(6).name = "Kite string";
    _movie.endRecording();
}
```

See also

beginRecording(), endRecording(), Sprite Channel

name (timeout)

Usage

```
timeoutObject.name
```

Description

This timeout property is the name of the timeout object as defined when the object is created. The `new()` command is used to create timeout objects.

Example

This timeout handler opens an alert with the name of the timeout that sent the event:

```
on handleTimeout timeoutObject
    alert "Timeout:" && timeoutObject.name
end
```

See also

forget() (Timeout), new(), period, persistent, target, time (timeout object), timeout(), timeoutHandler, timeoutList

name (XML)

Usage

```
XMLnode.name
```

Description

XML property; returns the name of the specified XML node.

Example

Beginning with this XML:

```
<?xml version="1.0"?>
 <e1>
<tagName attr1="val1" attr2="val2"/>
<e2> element 2</e2>
<e3>element 3</e3>
 </e1>
```

This Lingo returns the name of the second tag that is nested within the tag `<e1>`:

```
put gParserObject.child[1].child[2].name
-- "e2"
```

See also

attributeName

near (fog)

Usage

```
member(whichCastmember).camera(whichCamera).fog.near
cameraReference.fog.near
member(whichCastmember).camera(whichCamera).fog.far
cameraReference.fog.far
```

Description

3D properties; this property allows you to get or set the distance from the front of the camera to the point where the fogging starts if `fog.enabled` is `TRUE`.

The default value for this property is 0.0.

Example

This following statement sets the `near` property of the fog of the camera Defaultview to 100.

```
-- Lingo syntax
member("3dobjects").camera("defaultview").fog.near = 100.0

// JavaScript syntax
member("3dobjects").getPropRef("camera",1).fog.near = 100.00;
```

See also

fog, far (fog), enabled (fog), decayMode

nearFiltering

Usage

```
member(whichCastmember).texture(whichTexture).nearFiltering
member(whichCastmember).shader(whichShader).texture(whichTexture).nearFiltering
member(whichCastmember).model(whichModel).shader.texture(whichTexture).nearFiltering
member(whichCastmember).model(whichModel).shaderList[shaderListIndex].texture(whichTexture
).nearFiltering
```

Description

3D texture property; allows you to get or set whether bilinear filtering is used when rendering a projected texture map that covers more screen space than the original texture source. Bilinear filtering smooths any errors across the texture and thus improves the texture's appearance. Bilinear filtering smooths errors in two dimensions. Trilinear filtering smooths errors in three dimensions. Filtering improves appearance at the expense of performance, with bilinear being less performance-costly than trilinear.

When the property's value is `TRUE`, bilinear filtering is used. When the value is `FALSE`, bilinear filtering is not used. The default is `TRUE`.

Example

This statement turns off bilinear filtering for the first texture in the cast member 3Dobjects.

```
-- Lingo syntax
member("3dobjects").texture[1].nearFiltering = FALSE

// JavaScript syntax
member("3dobjects").getPropRef("texture",1).nearFiltering = 0;
```

netPresent

Usage

```
-- Lingo syntax
_player.netPresent

// JavaScript syntax
_player.netPresent;
```

Description

Player property; determines whether the Xtra extensions needed to access the Internet are available but does not report whether an Internet connection is currently active. Read-only.

If the Net Support Xtra extensions are not available, `netPresent` will function properly, but `netPresent()` will cause a script error.

Example

This statement sends an alert if the Xtra extensions are not available:

```
-- Lingo syntax
if (not(_player.netPresent)) then
    _player.alert("Sorry, the Network Support Xtras could not be found.")
end if

// JavaScript syntax
if (!(_player.netPresent)) {
    _player.alert("Sorry, the Network Support Xtras could not be found.");
}
```

See also

Player

netThrottleTicks

Usage

```
-- Lingo syntax
_player.netThrottleTicks

// JavaScript syntax
_player.netThrottleTicks;
```

Description

Player property; in the Mac authoring environment, allows you to control the frequency of servicing to a network operation. Read/write.

The default value is 15. The higher the value is set, the smoother the movie playback and animation is, but less time is spent servicing any network activity. A low setting allows more time to be spent on network operations, but will adversely affect playback and animation performance.

This property only affects the authoring environment and projectors on the Mac. It is ignored on Windows or Shockwave Player on the Mac.

Example

This statement shows the value of `netthrottleticks` in the player.

```
-- Lingo syntax
put _player.netThrottleTicks

// JavaScript syntax
 put( _player.netThrottleTicks);
```

See also

Player

node

Usage

```
-- Lingo syntax
spriteObjRef.node

// JavaScript syntax
spriteObjRef.node;
```

Description

QuickTime VR sprite property; the current node ID displayed by the sprite.

This property can be tested and set.

Example

This statement shows the current node ID displayed by the sprite.

```
-- Lingo syntax
put sprite(3).node

// JavaScript syntax
put( sprite(3).node);
```

nodeEnterCallback

Usage

```
-- Lingo syntax
spriteObjRef.nodeEnterCallback

// JavaScript syntax
spriteObjRef.nodeEnterCallback;
```

Description

QuickTime VR sprite property; contains the name of the handler that runs after the QuickTime VR movie switches to a new active node on the Stage. The message has two arguments: the `me` parameter and the ID of the node that is being displayed.

The QuickTime VR sprite receives the message first.

To clear the callback, set this property to 0.

To avoid a performance penalty, set a callback property only when necessary.

This property can be tested and set.

Example

This statement shows the name of the handler that runs after the QuickTime VR movie switches to a new active node on the Stage.

```
-- Lingo syntax
put sprite(3).nodeEnterCallback

// JavaScript syntax
put( sprite(3).nodeEnterCallback);
```

nodeExitCallback

Usage

```
-- Lingo syntax
spriteObjRef.nodeExitCallback

// JavaScript syntax
spriteObjRef.nodeExitCallback;
```

Description

QuickTime VR sprite property; contains the name of the handler that runs when the QuickTime VR movie is about to switch to a new active node on the Stage. The message has three arguments: the `me` parameter, the ID of the node that the movie is about to leave, and the ID of the node that the movie is about to switch to.

The value that the handler returns determines whether the movie goes on to the next node. If the handler returns `#continue`, the QuickTime VR sprite continues with a normal node transition. If the handler returns #cancel, the transition doesn't occur and the movie stays in the original node.

Set this property to 0 to clear the callback.

The QuickTime VR sprite receives the message first.

To avoid a performance penalty, set a callback property only when necessary.

This property can be tested and set.

Example

This statement shows the name of the handler that runs when the QuickTime VR movie is about to switch to a new active node on the Stage.

```
-- Lingo syntax
put sprite(3).nodeExitCallback
```

```
// JavaScript syntax
put( sprite(3).nodeExitCallback);
```

nodeType

Usage

```
-- Lingo syntax
spriteObjRef.nodeType

// JavaScript syntax
spriteObjRef.nodeType;
```

Description

QuickTime VR sprite property; gives the type of node that is currently on the Stage for the specified sprite. Possible values are `#object`, `#panorama`, or `#unknown`. (`#unknown` is the value for a sprite that isn't a QuickTime VR sprite.)

This property can be tested but not set.

Example

This statement shows the type of node that is currently on the Stage for the specified sprite.

```
-- Lingo syntax
put sprite(3).nodeType

// JavaScript syntax
put( sprite(3).nodeType);
```

normalList

Usage

```
member(whichCastmember).modelResource(whichModelResource).normalList
model.meshDeform.mesh[index].normalList
```

Description

3D property; when used with a model resource whose type is `#mesh`, this property allows you to get or set the `normalList` property of the model resource.

The `normalList` property is a linear list of vectors from which you may specify vertex normals when building the faces of your mesh.

This property must be set to a list of exactly the number of vectors specified in the `newMesh()` call.

Alternately, the `normalList` property may be generated for you by the `generateNormals()` method of mesh model resources.

In the context of the `meshDeform` modifier, the `normalList` property is similarly a linear list of vectors from which you may specify vertex normals when deforming your mesh.

For more information on face normals and vertex normals, see the `normals` entry.

Example

This statement shows the `normalList` property of the model resource named pyramid of the cast member named 3Dobjects.

```
-- Lingo syntax
put member("3Dobjects").modelResource("pyramid").normalList[2]

// JavaScript syntax
put(member("3Dobjects").getPropRef("modelResource",10).normalList[2]);
```

See also

face, meshDeform (modifier)

normals

Usage

```
member(whichCastmember).modelResource(whichModelResource).face[index].normals
```

Usage

3D face property; for model resources whose type is `#mesh` (created using the `newMesh` command) this property allows you to get and set the list of normal vectors used by the face specified by the *index* parameter.

Set this property to a linear list of integers corresponding to the index position of each vertex's normal in the model resource's `normalList` property.

This property must be set to the same length as the `face[index].verticies` list, or it can be an empty list [ ].

Do not set any value for this property if you are going to generate normal vectors using the `generateNormals()` command.

If you make changes to this property, or a use the `generateNormals()` command, you need to call the `build()` command in order to rebuild the mesh.

Example

This statement shows the `normals` property of the fifth face of the model resource named pyramid of the cast member named 3Dobjects.

```
-- Lingo syntax
put member("3Dobjects").modelResource("pyramid").face[5].normals

// JavaScript syntax
put(member("3Dobjects").getPropRef("modelResource",10).face[5].normals);
```

See also

face, normalList, vertices

number (Cast)

Usage

```
-- Lingo syntax
castObjRef.number
```

```
// JavaScript syntax
castObjRef.number;
```

Description

Cast library property; returns the number of a specified cast library. Read-only.

Example

This repeat loop uses the Message window to display the number of cast members that are in each of the movie's casts:

```
-- Lingo syntax
repeat with n = 1 to _movie.castLib.count
    put(castLib(n).name && "contains" && castLib(n).member.count && "cast members.")
end repeat

// JavaScript syntax
for (var n=1; n<=_movie.castLib.count; n++) {
    put(castLib(n).name + " contains " + castLib(n).member.count + " cast members.")
}
```

See also

Cast Library

number (characters)

Usage

```
the number of chars in chunkExpression
```

Description

Chunk expression; returns a count of the characters in a chunk expression.

Chunk expressions are any character (including spaces and control characters such as tabs and carriage returns), word, item, or line in any container of characters. Containers include field cast members and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

Note: *The* `count()` *function provides a more efficient alternative for determining the number of characters in a chunk expression.*

Example

This statement displays the number of characters in the string "Adobe, the Multimedia Company" in the Message window:

```
put the number of chars in "Adobe, the Multimedia Company"
```

The result is 29.

This statement sets the variable `charCounter` to the number of characters in the word i located in the string Names:

```
charCounter = the number of chars in member("Names").word[i]
```

You can accomplish the same thing with text cast members using the syntax:

```
charCounter = member("Names").word[i].char.count
```

See also

length(), char...of, count(), number (items), number (lines), number (words)

number (items)

Usage

```
the number of items in chunkExpression
```

Description

Chunk expression; returns a count of the items in a chunk expression. An item chunk is any sequence of characters delimited by commas.

Chunk expressions are any character, word, item, or line in any container of characters. Containers include fields (field cast members) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

Note: *The `count()` function provides a more efficient alternative for determining the number of items in a chunk expression.*

Example

This statement displays the number of items in the string "Adobe, the Multimedia Company" in the Message window:

```
put the number of items in "Adobe, the Multimedia Company"
```

The result is 2.

This statement sets the variable `itemCounter` to the number of items in the field Names:

```
itemCounter = the number of items in member("Names").text
```

You can accomplish the same thing with text cast members using the syntax:

```
itemCounter = member("Names").item.count
```

See also

item...of, count(), number (characters), number (lines), number (words)

number (lines)

Usage

```
the number of lines in chunkExpression
```

Description

Chunk expression; returns a count of the lines in a chunk expression. (Lines refers to lines delimited by carriage returns, not lines formed by line wrapping.)

Chunk expressions are any character, word, item, or line in any container of characters. Containers include field cast members and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

Note: *The `count()` function provides a more efficient alternative for determining the number of lines in a chunk expression.*

Example

This statement displays the number of lines in the string "Adobe, the Multimedia Company" in the Message window:

```
put the number of lines in "Adobe, the Multimedia Company"
```

The result is 1.

This statement sets the variable `lineCounter` to the number of lines in the field Names:

```
lineCounter = the number of lines in member("Names").text
```

You can accomplish the same thing with text cast members with the syntax:

```
lineCounter = member("Names").line.count
```

See also

line...of, count(), number (characters), number (items), number (words)

number (Member)

Usage

```
-- Lingo syntax
memberObjRef.number

// JavaScript syntax
memberObjRef.number;
```

Description

Member property; indicates the cast library number of a specified cast member. Read-only.

The value of this property is a unique identifier for the cast member that is a single integer describing its location in and position in the cast library.

Example

This statement assigns the cast number of the cast member Power Switch to the variable *whichCastMember*:

```
-- Lingo syntax
whichCastMember = member("Power Switch").number

// JavaScript syntax
var whichCastMember = member("Power Switch").number;
```

This statement assigns the cast member Red Balloon to sprite 1:

```
-- Lingo syntax
sprite(1).member = member("Red Balloon").number

// JavaScript syntax
sprite(1).member = member("Red Balloon").number;
```

This verifies that a cast member actually exists before trying to switch the cast member in the sprite:

```
-- Lingo syntax
property spriteNum

on mouseUp me
    if (member("Mike's face").number > 0) then
        sprite(spriteNum).member = "Mike's face"
```

```
    end if
end

// JavaScript syntax
function mouseUp() {
    if (member("Mike's face").number > 0) {
        sprite(this.spriteNum).member = "Mike's face"
    }
}
```

See also

castLib(), Member

number (menus)

Usage

```
the number of menus
```

Description

Menu property; indicates the number of menus installed in the current movie.

This menu property can be tested but not set. Use the `installMenu` command to set up a custom menu bar.

Note: *Menus are not available in Shockwave Player.*

Example

This statement determines whether any custom menus are installed in the movie and, if no menus are already installed, installs the menu Menubar:

```
if the number of menus = 0 then installMenu "Menubar"
```

This statement displays in the Message window the number of menus that are in the current movie:

```
put the number of menus
```

See also

installMenu, number (menu items)

number (menu items)

Usage

```
the number of menuItems of menu whichMenu
```

Description

Menu property; indicates the number of menu items in the custom menu specified by *whichMenu*. The *whichMenu* parameter can be a menu name or menu number.

This menu property can be tested but not set. Use the `installMenu` command to set up a custom menu bar.

Note: *Menus are not available in Shockwave Player.*

Example

This statement sets the variable `fileItems` to the number of menu items in the custom File menu:

```
fileItems = the number of menuItems of menu "File"
```

This statement sets the variable `itemCount` to the number of menu items in the custom menu whose menu number is equal to the variable i:

```
itemCount = the number of menuItems of menu i
```

See also

installMenu, number (menus)

number (Sprite Channel)

Usage

```
-- Lingo syntax
spriteChannelObjRef.number

// JavaScript syntax
spriteChannelObjRef.number;
```

Description

Sprite Channel property; returns the number of a sprite channel. Read-only.

Example

This statement displays in the Message window the number of a named sprite channel:

```
-- Lingo syntax
put(channel("Kite String").number)

// JavaScript syntax
put(channel("Kite String").number);
```

See also

Sprite

number (system)

Usage

```
the number of castLibs
```

Description

System property; returns the number of casts that are in the current movie.

This property can be tested but not set.

Example

This repeat loop uses the Message window to display the number of cast members that are in each of the movie's casts:

```
repeat with n = 1 to the number of castLibs
```

```
    put castLib(n).name && "contains" && the number of members of castLib(n) && "cast
members."
end repeat
```

number (words)

Usage

```
the number of words in chunkExpression
```

Description

Chunk expression; returns the number of words in the chunk expression specified by *chunkExpression*.

Chunk expressions are any character, word, item, or line in any container of characters. Containers include field cast members and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

To accomplish this functionality with text cast members, see `count`.

Note: *The `count()` function provides a more efficient alternative for determining the number of words in a chunk expression.*

Example

This statement displays in the Message window the number of words in the string "Adobe, the multimedia company":

```
put the number of words in "Adobe, the multimedia company"
```

The result is 4.

This handler reverses the order of words in the string specified by the argument `wordList`:

```
on reverse wordList
    theList = EMPTY
    repeat with i = 1 to the number of words in wordList
        put word i of wordList & " " before theList
    end repeat
    delete theList.char[thelist.char.count]
    return theList
end
```

See also

`count()`, `number (characters)`, `number (items)`, `number (lines)`, `word...of`

number of members

Usage

```
the number of members of castLib whichCast
```

Description

Cast member property; indicates the number of the last cast member in the specified cast.

This property can be tested but not set.

Example

The following statement displays in the Message window the type of each cast member in the cast Central Casting. The number of `members of castLib` property is used to determine how many times the loop repeats.

```
repeat with i = 1 to the number of members of castLib("Central Casting")
    put "Cast member" && i && "is a" && member(i, "Central Casting").type
end repeat
```

number of xtras

Usage

```
the number of xtras
```

Description

System property; returns the number of scripting Xtra extensions available to the movie. The Xtra extensions may be either those opened by the `openxlib` command or those present in the Configuration\Xtras folder.

This property can be tested but not set.

Example

This statement displays in the Message window the number of scripting Xtra extensions that are available to the movie:

```
put the number of xtras
```

numChannels

Usage

```
-- Lingo syntax
memberObjRef.numChannels

// JavaScript syntax
memberObjRef.numChannels;
```

Description

Shockwave Audio (SWA) cast member property; returns the number of channels within the specified SWA streaming cast member. The value can be either 1 for monaural or 2 for stereo.

This property is available only after the SWA streaming cast member begins playing or after the file has been preloaded using the `preLoadBuffer` command.

This property can be tested but not set.

Example

This example assigns the number of sound channels of the SWA streaming cast member Duke Ellington to the field cast member Channel Display:

```
-- Lingo syntax
myVariable = member("Duke Ellington").numChannels
if myVariable = 1 then
    member("Channel Display").text = "Mono"
else
```

```
    member("Channel Display").text = "Stereo"
end if

// JavaScript syntax
var myVariable = member("Duke Ellington").numChannels;
if (myVariable == 1) {
    member("Channel Display").text = "Mono";
} else {
    member("Channel Display").text = "Stereo";
}
```

numParticles

Usage

```
member(whichCastmember).modelResource(whichModelResource).emitter.numParticles
modelResourceObjectReference.emitter.numParticles
```

Description

3D property; when used with a model resource whose type is `#particle`, allows you to get or set the `numParticles` property of the resource's particle emitter. The value must be greater than 0 and no more than 100000. The default setting is 1000.

Example

This statement sets the number of particles to 50000 in the cast member named 3Dobjects.

```
-- Lingo syntax
member("3Dobjects").modelResource("Particle01").emitter.numParticles = 50000
// JavaScript syntax
member("3Dobjects").getPropRef("modelResource",10).emitter.numParticles = 50000;
```

See also

emitter

numSegments

Usage

```
member(whichCastmember).modelResource(whichModelResource).numSegments
```

Description

3D property; when used with a model resource whose type is `#cylinder`, allows you to get or set the `numSegments` property of the model resource.

The `numSegments` property determines the number of segments running from the top cap of the cylinder to the bottom cap. This property must be greater than or equal to the default value of 2.

The smoothness of the cylinder's surface depends upon the value specified for this property. The greater the property value the smoother the cylinder's surface will appear.

Example

This statement sets the `numSegments` property of the model resource named Cylinder01 to 10.

```
-- Lingo syntax
member("3Dobjects").modelResource("Cylinder01").numSegments = 10

// JavaScript syntax
member("3Dobjects").getPropRef("modelResource",11).numSegments = 10;
```

obeyScoreRotation

Usage

```
member(flashMember).obeyScoreRotation
```

Description

Flash cast member property; set to TRUE or FALSE to determine if a Flash movie sprite uses the rotation information from the Score, or the older rotation property of Flash assets.

This property is automatically set to FALSE for all movies created in Director prior to version 7 in order to preserve old functionality of using the member rotation property for all sprites containing that Flash member.

New assets created in version 7 or later will have this property automatically set to TRUE.

If set to TRUE, the rotation property of the member is ignored and the Score rotation settings are obeyed instead.

Example

The following script sets the obeyScoreRotation property of cast member "FlashObj" to 1 (TRUE), then rotates the sprite which contains the cast member 180°.

```
-- Lingo syntax
member("FlashObj").obeyScoreRotation = 1
sprite(1).rotation = sprite(1).rotation + 180

// JavaScript syntax
member("FlashObj").obeyScoreRotation = 1;
sprite(1).rotation = sprite(1).rotation + 180;
```

See also

rotation

optionDown

Usage

```
-- Lingo syntax
_key.optionDown

// JavaScript syntax
_key.optionDown;
```

Description

Key property; determines whether the user is pressing the Alt key (Windows) or the Option key (Mac). Read-only.

This property returns TRUE if the user is pressing the Alt or Option key; otherwise, it returns FALSE.

In Windows, optionDown does not work in projectors if Alt is pressed without another nonmodifier key. Avoid using optionDown if you intend to distribute a movie as a Windows projector and need to detect only the modifier key press; use controlDown or shiftDown instead.

On the Mac, pressing the Option key changes the key value, so use keyCode instead.

Example

This handler checks whether the user is pressing the Alt or the Option key and, if so, calls the handler named doOptionKey:

```
-- Lingo syntax
on keyDown
    if (_key.optionDown) then
        doOptionKey(_key.key)
    end if
end

// JavaScript syntax
function keyDown() {
    if (_key.optionDown) {
        doOptionKey(_key.key);
    }
}
```

See also

controlDown, Key, key, keyCode, shiftDown

organizationName

Usage

```
-- Lingo syntax
_player.organizationName

// JavaScript syntax
_player.organizationName;
```

Description

Player property; contains the company name entered during installation of Director. Read-only.

This property is available in the authoring environment only. It can be used in a movie in a window tool that is personalized to show the user's information.

Example

The following handler would be located in a movie script of a movie in a window (MIAW). It places the user's name and serial number into a display field when the window is opened:

```
-- Lingo syntax
on prepareMovie
    displayString = _player.userName & RETURN & _player.organizationName & RETURN &
_player.serialNumber
    member("User Info").text = displayString
end

// JavaScript syntax
function prepareMovie() {
```

```
    var displayString = _player.userName + "\n" + _player.organizationName+ "\n" +
_player.serialNumber;
    member("User Info").text = displayString;
}
```

See also

Player

originalFont

Usage

```
-- Lingo syntax
memberObjRef.originalFont

// JavaScript syntax
memberObjRef.originalFont;
```

Description

Font cast member property; returns the exact name of the original font that was imported when the given cast member was created.

Example

This statement displays the name of the font that was imported when cast member 11 was created:

```
-- Lingo syntax
put(member(11).originalFont)

// JavaScript syntax
put(member(11).originalFont);
```

See also

recordFont, bitmapSizes, characterSet

originH

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.originH

// JavaScript syntax
memberOrSpriteObjRef.originH;
```

Description

Cast member and sprite property; controls the horizontal coordinate of a Flash movie or vector shape's origin point, in pixels. The value can be a floating-point value.

The origin point is the coordinate in a Flash movie or vector shape around which scaling and rotation occurs. The origin point can be set with floating-point precision using the separate `originH` and `originV` properties, or it can be set with integer precision using the single `originPoint` property.

You can set the `originH` property only if the `originMode` property is set to `#point`.

This property can be tested and set. The default value is 0.

Note: *This property must be set to the default value if the scaleMode property is set to #autoSize, or the sprite does not display correctly.*

Example

This sprite script uses the `originMode` property to set up a Flash movie sprite so it's origin point can be set to a specific point. It then sets the horizontal and vertical origin points.

```
-- Lingo syntax
property spriteNum

on beginSprite me
    sprite(spriteNum).originMode = #point
    sprite(spriteNum).originH = 100
    sprite(spriteNum).originV = 80
end

// JavaScript syntax
function beginSprite() {
    sprite(this.spriteNum).originMode = symbol("point");
    sprite(this.spriteNum).originH = 100;
    sprite(this.spriteNum).originV = 80;
}
```

See also

originV, originMode, originPoint, scaleMode

originMode

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.originMode

// JavaScript syntax
memberOrSpriteObjRef.originMode;
```

Description

Cast member property and sprite property; sets the origin point around which scaling and rotation occurs, as follows:

- `#center` (default)—The origin point is at the center of the Flash movie.
- `#topleft`—The origin point is at the top left of the Flash movie.
- `#point`—The origin point is at a point specified by the `originPoint`, `originH`, and `originV` properties.

This property can be tested and set.

Note: *This property must be set to the default value if the scaleMode property is set to #autoSize, or the sprite will not display correctly.*

Example

This sprite script uses the `originMode` property to set up a Flash movie sprite so its origin point can be set to a specific point. It then sets the horizontal and vertical origin points.

```
-- Lingo syntax
property spriteNum

on beginSprite me
    sprite(spriteNum).originMode = #point
    sprite(spriteNum).originH = 100
    sprite(spriteNum).originV = 80
end

// JavaScript syntax
function beginSprite() {
    sprite(this.spriteNum).originMode = symbol("point");
    sprite(this.spriteNum).originH = 100;
    sprite(this.spriteNum).originV = 80;
}
```

See also

originH, originV, originPoint, scaleMode

originPoint

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.originPoint

// JavaScript syntax
memberOrSpriteObjRef.originPoint;
```

Description

Cast member and sprite property; controls the origin point around which scaling and rotation occurs of a Flash movie or vector shape.

The `originPoint` property is specified as a Director point value: for example, point(100,200). Setting a Flash movie or vector shape's origin point with the `originPoint` property is the same as setting the `originH` and `originV` properties separately. For example, setting the `originPoint` property to point(50,75) is the same as setting the `originH` property to 50 and the `originV` property to 75.

Director point values specified for the `originPoint` property are restricted to integers, whereas `originH` and `originV` can be specified with floating-point numbers. When you test the `originPoint` property, the point values are truncated to integers. As a rule of thumb, use the `originH` and `originV` properties for precision; use the `originPoint` property for speed and convenience.

You can set the `originPoint` property only if the `originMode` property is set to `#point`.

This property can be tested and set. The default value is 0.

Note: *This property must be set to the default value if the scaleMode property is set to #autoSize, or the sprite will not display correctly.*

Example

This sprite script uses the `originMode` property to set up a Flash movie sprite so its origin point can be set to a specific point. It then sets the origin points.

```
-- Lingo syntax
property spriteNum
```

```
on beginSprite me
    sprite(spriteNum).scaleMode = #showAll
    sprite(spriteNum).originMode = #point
    sprite(spriteNum).originPoint = point(100, 80)
end

// JavaScript syntax
function beginSprite() {
    sprite(this.spriteNum).scaleMode = symbol("showAll");
    sprite(this.spriteNum).originMode = symbol("point");
    sprite(this.spriteNum).originPoint = point(100, 80);
}
```

See also

originH, originV, scaleMode

originV

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.originv

// JavaScript syntax
memberOrSpriteObjRef.originV;
```

Description

Cast member and sprite property; controls the vertical coordinate of a Flash movie or vector shape's origin point around which scaling and rotation occurs, in pixels. The value can be a floating-point value.

The origin point can be set with floating-point precision using the separate `originH` and `originV` properties, or it can be set with integer precision using the single `originPoint` property.

You can set the `originV` property only if the `originMode` property is set to `#point`.

This property can be tested and set. The default value is 0.

Note: *This property must be set to the default value if the scaleMode property is set to #autoSize, or the sprite does not display correctly.*

Example

This sprite script uses the `originMode` property to set up a Flash movie sprite so its origin point can be set to a specific point. It then sets the horizontal and vertical origin points.

```
-- Lingo syntax
property spriteNum

on beginSprite me
    sprite(spriteNum).scaleMode = #showAll
    sprite(spriteNum).originMode = #point
    sprite(spriteNum).originH = 100
    sprite(spriteNum).originV = 80
end

// JavaScript syntax
function beginSprite() {
```

```
    sprite(this.spriteNum).scaleMode = symbol("showAll");
    sprite(this.spriteNum).originMode = symbol("point");
    sprite(this.spriteNum).originH = 100;
    sprite(this.spriteNum).originV = 80;
}
```

See also

originH, originPoint, scaleMode

orthoHeight

Usage

```
member(whichCastmember).camera(whichCamera).orthoHeight
member(whichCastmember).camera[cameraindex].orthoHeight
sprite(whichSprite).camera.orthoHeight
```

Description

3D property; when `camera.projection` is set to `#orthographic`, the value `camera.orthoHeight` gives the number of perpendicular world units that fit vertically in the sprite. World units are the measuring units for the particular 3D world. They are internally consistent but arbitrarily chosen, and they can vary from one 3D world to another.

You do not need to specify the camera index (*whichCamera*) to access the first camera of the sprite.

The default value of this property is 200.0

Example

The following statement sets the `orthoHeight` of the camera of sprite 1 to 200. This means 200 world units will fit vertically within the sprite.

```
-- Lingo syntax
sprite(1).camera.orthoheight = 200.0

// JavaScript syntax
sprite(1).camera.orthoheight = 200.0;
```

See also

projection

overlay

Usage

```
member(whichCastmember).camera(whichCamera).overlay[overlayIndex].propertyName
member(whichCastmember).camera(whichCamera).overlay.count
```

Description

3D camera property; allows both get and set access to the properties of overlays contained in the camera's list of overlays to be displayed. When used as `overlay.count` this property returns the total number of overlays contained in the camera's list of overlays to be displayed.

Overlays are textures displayed in front of all models appearing in a given camera's view frustum. The overlays are drawn in the order that they appear in the camera's overlay list, the first item in the list appears behind all other overlays and the last item in the list in front of all other overlays.

Each overlay in the camera's list of overlays list has the following properties:

- `loc` allows you to get or set the specific position of the overlay's `regPoint`, relative to the camera rect's upper left corner.
- `source` allows you to get or set the texture to use as the source image for the overlay.
- `scale` allows you to get or set the scale value used by the overlay. The scale determines the magnification of the overlay; this property defaults to a value of 1.0.
- `rotation` allows you to get or set the rotation, in degrees, of the overlay.
- `regPoint` allows you to get or set the registration point of the overlay relative to the texture's upper left corner.
- `blend` allows you to get or set the blending of the overlay to an integer between 0 and 100, indicating how transparent (0) or opaque (100) the overlay is.

Example

This statement displays the `scale` property of the first overlay in the sprite camera's overlay.

```
-- Lingo syntax
put sprite(2).camera.overlay[1].scale
```

See also

`addOverlay`, `removeOverlay`, `bevelDepth`

pageHeight

Usage

```
-- Lingo syntax
memberObjRef.pageHeight

// JavaScript syntax
memberObjRef.pageHeight;
```

Description

Field cast member property; returns the height, in pixels, of the area of the field cast member that is visible on the Stage.

This property can be tested but not set.

Example

This statement returns the height of the visible portion of the field cast member Today's News:

```
--Lingo syntax
trace(member("Today's News").pageHeight)

// JavaScript syntax
trace(member("Today's News").pageHeight);
```

palette

Usage

```
-- Lingo syntax
memberObjRef.palette

// JavaScript syntax
memberObjRef.palette;
```

Description

Cast member property; for bitmap cast members only, determines which palette is associated with the cast member specified by *whichCastMember*.

This property can be tested and set.

Example

This statement displays the palette assigned to the cast member Leaves in the Message window:

```
-- Lingo syntax
put(member("Leaves").palette)

// JavaScript syntax
put(member("Leaves").palette);
```

paletteMapping

Usage

```
-- Lingo syntax
_movie.paletteMapping

// JavaScript syntax
_movie.paletteMapping;
```

Description

Movie property; determines whether the movie remaps (`TRUE`) or does not remap (`FALSE`, default) palettes for cast members whose palettes are different from the current movie palette. Read/write.

The effect of this property is similar to that of the Remap Palettes When Needed check box in the Movie Properties dialog box.

To display different bitmaps with different palettes simultaneously, set `paletteMapping` to `TRUE`. Director looks at each onscreen cast member's reference palette (the palette assigned in its Cast Member Properties dialog box) and, if it is different from the current palette, finds the closest match for each pixel in the new palette.

The colors of the nonmatching bitmap will be close to the original colors.

Remapping consumes processor time, and it's usually better to adjust the bitmap's palette in advance.

Remapping can also produce undesirable results. If the palette changes in the middle of a sprite span, the bitmap immediately remaps to the new palette and appears in the wrong colors. However, if anything refreshes the screen—a transition or a sprite moving across the Stage—then the affected rectangle on the screen appears in remapped colors.

Example

This statement tells the movie to remap the movie's palette whenever necessary:

```
-- Lingo syntax
_movie.paletteMapping = TRUE

// JavaScript syntax
_movie.paletteMapping = true;
```

See also

Movie

paletteRef

Usage

```
member(whichCastMember). paletteRef
the paletteRef
```

Description

Bitmap cast member property; determines the palette associated with a bitmap cast member. Built-in Director palettes are indicated by symbols (`#systemMac`, `#rainbow`, and so on). Palettes that are cast members are treated as cast member references. This behavior differs from that of the `palette` member property, which returns a positive number for cast palettes and negative numbers for built-in Director palettes.

This property can be tested and set.

Example

This statement assigns the Mac system palette to the bitmap cast member Shell:

```
member("Shell").paletteRef = #systemMac
```

pan

Usage

```
-- Lingo syntax
soundChannelObjRef.pan

// JavaScript syntax
soundChannelObjRef.pan;
```

Description

Sound Channel property; indicates the left/right balance of the sound playing in a sound channel. Read/write.

The range of values is from -100 to 100. -100 indicates only the left channel is heard. 100 indicate only the right channel is being heard. A value of 0 indicates even left/right balance, causing the sound source to appear to be centered. For mono sounds, `pan` affects which speaker (left or right) the sound plays through.

You can change the pan of a sound object at any time, but if the sound channel is currently performing a fade, the new pan setting doesn't take effect until the fade is complete.

To see an example of `pan` used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

These statements pan the sound in sound channel 2 from the left channel to the right channel:

```
-- Lingo syntax
repeat with x = -100 to 100
 sound(2).pan = x
end repeat

// JavaScript syntax
for (var x = -100; x <= 100; x++) {
 sound(2).pan = x;
}
```

See also

Sound Channel

pan (QTVR property)

Usage

```
-- Lingo syntax
spriteObjRef.pan

// JavaScript syntax
spriteObjRef.pan;
```

Description

QuickTime VR sprite property; the current pan of the QuickTime VR movie. The value is in degrees.

This property can be tested and set.

paragraph

Usage

```
chunkExpression.paragraph[whichParagraph]
chunkExpression.paragraph[firstParagraph..lastParagraph]
```

Description

Text cast member property; this chunk expression allows access to different paragraphs within a text cast member.

The paragraph is delimited by a carriage return.

```
put member("AnimText").paragraph[3]
```

Example

This statement returns the second paragraph from the Text cast member myText.

```
-- Lingo syntax
put member("myText").paragraph[2]
```

See also

line...of

parent

Usage

```
member(whichCastmember).model(whichModel).parent
member(whichCastmember).camera(whichCamera).parent
member(whichCastmember).light(whichLight).parent
member(whichCastmember).group(whichGroup).parent
```

Description

3D property; when used with a model, camera, light or group reference, this property allows you to get or set the parent node of the referenced object. The parent node can be any other model, camera, light or group object.

An object's `transform` property defines its scale, position and orientation relative to its parent object.

Setting an object's parent property to `VOID` is the same as removing the object from the world using the `removeFromWorld()` command.

Setting an object's parent property to the World group object (`group("World")`) is the same as adding an object to the world using the `addToWorld()` command.

You can also alter the value of this property by using the `addChild` command.

Example

The following statement sets the `parent` property of the model named Tire. Its parent is set to the model named Car.

```
-- Lingo syntax
member("3Dobjects").model("Tire").parent = member("3Dobjects").model("Car")

// JavaScript syntax
member("3Dobjects").getPropRef("model",2).parent =
member("3Dobjects").getPropRef("model",3);
```

See also

child (3D), addChild

password

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.password

// JavaScript syntax
memberOrSpriteObjRef.password;
```

Description

RealMedia sprite and cast member property; allows you to set the password required to access a protected RealMedia stream. For security reasons, you cannot use this property to retrieve a password previously specified for this property. If a password has been set previously, the value of this property is the string `"********"`. If no password has been set, the value of this property is an empty string.

Example

The following examples show that the password has been set for the RealMedia stream in the cast member Real or sprite 2.

```
-- Lingo syntax
put(sprite(2).password) -- "********"
put(member("Real").password) -- "********"

// JavaScript syntax
put(sprite(2).password); // "********"
put(member("Real").password); // "********"
```

The following examples show that the password has never been set for the RealMedia stream in the cast member Real or sprite 2.

```
-- Lingo syntax
put(sprite(2).password) -- ""
put(member("Real").password) -- ""

// JavaScript syntax
put(sprite(2).password); // ""
put(member("Real").password); // ""
```

The following examples set the password for the RealMedia stream in sprite 2 and the cast member Real to `"abracadabra"`.

```
-- Lingo syntax
sprite(2).password = "abracadabra"
member("Real").password = "abracadabra"

// JavaScript syntax
sprite(2).password = "abracadabra";
member("Real").password = "abracadabra";
```

See also

userName (RealMedia)

path (Movie)

Usage

```
-- Lingo syntax
_movie.path

// JavaScript syntax
_movie.path;
```

Description

Movie property; indicates the pathname of the folder in which the current movie is located. Read-only.

For pathnames that work on both Windows and Mac computers, use the `@` pathname operator.

To see an example of `path` used in a completed movie, see the Read and Write Text movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement displays the pathname for the folder containing the current movie:

```
-- Lingo syntax
trace(_movie.path)

// JavaScript syntax
trace(_movie.path);
```

This statement plays the sound file Crash.aif stored in the Sounds subfolder of the current movie's folder:

```
-- Lingo syntax
sound(1).playFile(_movie.path & "Sounds\Crash.aif")

// JavaScript syntax
sound(1).playFile(_movie.path + "Sounds\\Crash.aif");
```

See also

Movie

path (3D)

Usage

```
member(whichCastmember).modelResource(whichModelResource).emitter.path
```

Description

3D property; when used with a model resource whose type is `#particle`, allows you to get or set the `path` property of the resource's particle emitter.

This property is a list of vectors that define the path particles follow over their lifetime. The default value of this property is an empty list `[]`.

Example

In this example, ThermoSystem is a model resource of the type `#particle`. This statement specifies that the particles of ThermoSystem will follow the path outlined by the list of vectors.

```
member("Fires").modelResource("ThermoSystem").emitter.path = [vector(0,0,0),
vector(15,0,0), vector(30,30,-10)]
```

See also

pathStrength, emitter

pathName (Flash member)

Usage

```
member(whichFlashMember).pathName
the pathName of member whichFlashMember
```

Description

Cast member property; controls the location of an external file that stores the assets of a Flash movie cast member are stored. You can link a Flash movie to any path on a local or network drive or to a URL.

Setting the path of an unlinked cast member converts it to a linked cast member.

This property can be tested and set. The `pathName` property of an unlinked member is an empty string.

This property is the same as the `fileName` property for other member types, and you can use `fileName` instead of `pathName`.

Example

The following statement changes cast member's `pathName` property to the location of a Flash movie on the World Wide Web.

```
-- Lingo syntax
member("FlashObj").pathName = "http://www.someURL.com/myFlash.swf"

// JavaScript syntax
member("FlashObj").pathName = "http://www.someURL.com/myFlash.swf";
```

See also

`fileName (Member)`, `linked`

pathStrength

Usage

```
member(whichCastmember).modelResource(whichModelResource).emitter.pathStrength
```

Description

3D property; when used with a model resource whose type is `#particle`, determines how closely the particles follow the path specified by the `path` property of the emitter. Its range starts at 0.0 (no strength - so the particles won't be attracted to the path) and continues to infinity. Its default value is 0.1. Setting `pathStrength` to 0.0 is useful for turning the path off temporarily.

As the value of `pathStrength` gets larger, the entire particle system will get more and more stiff. Large `pathStrength` values will tend to make the particles bounce around very quickly, unless some dampening force is also used, such as the particle `drag` property.

This property can be tested and set.

Example

In this example, ThermoSystem is a model resource of the type `#particle`. This statement sets the `pathStrength` property of ThermoSystem to 0.97. If a path is outlined by ThermoSystem's `emitter.path` property, the particles follow that path very closely.

```
member("Fires").modelResource("ThermoSystem").emitter.pathStrength = 0.97
```

See also

`path (3D)`, `emitter`

pattern

Usage

```
member(whichCastMember).pattern
the pattern of member whichCastMember
```

Description

Cast member property; determines the pattern associated with the specified shape. Possible values are the numbers that correspond to the swatches in the Tools window's patterns palette. If the shape cast member is unfilled, the pattern is applied to the cast member's outer edge.

This property can be useful in movies with Shockwave content to change images by changing the tiling applied to a shape, allowing you to save memory required by larger bitmaps.

This property can be tested and set.

Example

The following statements make the shape cast member myShape a filled shape and assign it pattern 1, which is a solid color.

```
-- Lingo syntax
member("myShape").filled = TRUE
member("myShape").pattern = 1

// JavaScript syntax
member("myShape").filled = 1;
member("myShape").pattern = 1;
```

pausedAtStart (Flash, Digital video)

Usage

```
member(whichFlashOrDigitalVideoMember).pausedAtStart
the pausedAtStart of member whichFlashOrDigitalVideoMember
```

Description

Cast member property; controls whether the digital video or Flash movie plays when it appears on the Stage. If this property is `TRUE`, the digital video or Flash movie does not play when it appears. If this property is `FALSE`, it plays immediately when it appears.

For a digital video cast member, the property specifies whether the Paused at Start check box in the Digital Video Cast Member Properties dialog box is selected or not.

This property can be tested and set.

Example

This statement turns on the Paused at Start check box in the Digital Video Cast Member Info dialog box for the QuickTime movie Rotating Chair:

```
member("Rotating Chair").pausedAtStart = TRUE
```

pausedAtStart (RealMedia, Windows Media)

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.pausedAtStart

// JavaScript syntax
memberOrSpriteObjRef.pausedAtStart;
```

Description

RealMedia and Windows Media sprite or cast member property; allows you to get or set whether a RealMedia or Windows Media stream on the Stage automatically begins to play when buffering is complete (FALSE or 0) or remains paused (TRUE or 1). Read/write.

This property can be set to an expression that evaluates to TRUE or FALSE. Integer values other than 1 or 0 are treated as TRUE. The default setting for this property is FALSE. You can set this property to TRUE by selecting Paused in the graphical view of the Property inspector.

If this property is set to FALSE, the user must click the Play button in the RealMedia or Windows Media viewer (or a button you have created for this purpose in your movie), or you must call the play() method to play the sprite when buffering is complete.

This property only affects score-based playback and does not affect playback in the RealMedia or Windows Media viewer.

Example

The following examples show that the pausedAtStart property of sprite 2 and the cast member Real is set to FALSE, which means that the RealMedia stream will automatically begin to play once buffering is complete.

```
-- Lingo syntax
put(sprite(2).pausedAtStart) -- 0
put(member("Real").pausedAtStart) -- 0

// JavaScript syntax
put(sprite(2).pausedAtStart); // 0
put(member("Real").pausedAtStart); // 0
```

The following examples set the pausedAtStart property for sprite 2 and the cast member Real to TRUE, which means the RealMedia stream will not begin to play unless the play command is called.

```
-- Lingo syntax
sprite(2).pausedAtStart = TRUE
member("Real").pausedAtStart = TRUE

// JavaScript syntax
sprite(2).pausedAtStart = 1;
member("Real").pausedAtStart = 1;
```

The following example uses the pausedAtStart property to buffer a RealMedia sprite off the Stage, and then play it on the Stage once the buffering is complete. In this example, the RealMedia member has its pausedAtStart property set to TRUE. An instance of this member is positioned off the Stage, in sprite channel 1. The following frame script should be placed in the sprite span:

```
-- Lingo syntax
on exitFrame me
    if sprite(1).state > 3 then -- check to see if buffering is complete
        sprite(1).locH = 162
```

```
        sprite(1).locV = 118
        sprite(1).play() -- position and play the sprite
    end if
end

// JavaScript syntax
function exitFrame() {
    var st = sprite(1).state;
    if (st > 3) { // check to see if buffering is complete
        sprite(1).locH = 162;
        sprite(1).locV = 118;
        sprite(1).play(); // position and play the sprite
    }
}
```

The RealMedia sprite will buffer off the Stage and then appear on the Stage and play immediately when the buffering is complete.

percentBuffered

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.percentBuffered

// JavaScript syntax
memberOrSpriteObjRef.percentBuffered;
```

Description

RealMedia sprite or cast member property; returns the percentage of the buffer that has been filled with the RealMedia stream that is loading from a local file or the server. When this property reaches 100, the buffer is full, and the RealMedia stream begins to play if the `pausedAtStart` property is not set to `TRUE`. This property is dynamic during playback and cannot be set.

The buffer is a type of memory cache that contains the portion of the movie that is about to play, usually just a few seconds. The stream enters the buffer as it streams to RealPlayer and leaves the buffer as RealPlayer plays the clip. The buffer allows viewers to view content without downloading the entire file, and prevents network congestion or lapses in bandwidth availability from disrupting the playback stream.

The buffering process is initiated by the `play` command, and once the buffer is full (100%), the portion of the stream that is in the buffer begins to play. Because the initial buffering process takes a few seconds, there is a delay between the time when the `play` command is called and when the stream actually begins to play. You can work around this using the `pausedAtStart` command, starting to play the stream off the Stage during the buffering process, and then displaying the stream on the Stage as it actually begins to play. (For more information, see the example in the "pausedAtStart (RealMedia, Windows Media)" on page 927 entry.)

Example

The following examples show that 56% of the RealMedia stream in sprite 2 and the cast member Real has been buffered.

```
-- Lingo syntax
put(sprite(2).percentBuffered) -- 56
put(member("Real").percentBuffered) -- 56

// JavaScript syntax
```

```
put(sprite(2).percentBuffered); // 56
put(member("Real").percentBuffered); // 56
```

See also

mediaStatus (RealMedia, Windows Media), pausedAtStart (RealMedia, Windows Media), state (RealMedia)

percentPlayed

Usage

```
member(whichCastMember).percentPlayed
the percentPlayed of member whichCastMember
```

Description

Shockwave Audio (SWA) cast member property; returns the percentage of the specified SWA file that has actually played.

This property can be tested only after the SWA sound starts playing or has been preloaded by means of the `preLoadBuffer` command. This property cannot be set.

Example

This handler displays the percentage of the SWA streaming cast member Frank Sinatra that has played and puts the value in the field cast member Percent Played:

```
on exitFrame
    whatState = member("Frank Sinatra").state
    if whatState > 1 AND whatState < 9 then
        member("Percent Played").text = string(member("Frank Sinatra").percentPlayed)
    end if
end
```

See also

percentStreamed (Member)

percentStreamed (3D)

Usage

```
member(whichCastMember).percentStreamed
```

Description

3D property; allows you to get the percentage of a 3D cast member that has been streamed. This property refers to either the initial file import or to the last file load requested. The value returned is an integer and has a range from 0 to 100. There is no default value for this property.

Example

This statement shows that the cast member PartyScene has finished loading.

```
put member("PartyScene").percentStreamed
-- 100
```

percentStreamed (Member)

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.percentStreamed

// JavaScript syntax
memberOrSpriteObjRef.percentStreamed;
```

Description

Shockwave Audio (SWA) and Flash cast member property, and QuickTime sprite property.

For SWA streaming sounds, gets the percent of a SWA file already streamed from an HTTP or FTP server. For SWA, this property differs from the `percentPlayed` property in that it includes the amount of the file that has been buffered but not yet played. This property can be tested only after the SWA sound starts playing or has been preloaded by means of the `preLoadBuffer` command.

For Flash movie cast members, this property gets the percent of a Flash movie that has streamed into memory.

For QuickTime sprites, this property gets the percent of the QuickTime file that has played.

This property can have a value from 0 to 100%. For a file on a local disk, the value is 100. For files being streamed from the Internet, the `percentStreamed` value increases as more bytes are received. This property cannot be set.

Example

This example displays the percentage of the SWA streaming cast member Ray Charles that has streamed and puts the value in a field:

```
-- Lingo syntax
on exitFrame
    whatState = member("Ray Charles").state
    if whatState > 1 AND whatState < 9 then
        member("Percent Streamed Displayer").text = string(member("Ray
Charles").percentStreamed)
    end if
end

// JavaScript syntax
function exitFrame() {
    var whatState = member("Ray Charles").state;
    var pcStm = new String(member("Ray Charles").percentStreamed);
    if (whatState > 1 && whatState < 9) {
        member("Percent Streamed Displayer").text = pcStm;
    }
}
```

This frame script keeps the playhead looping in the current frame so long as less than 60 percent of a Flash movie called Splash Screen has streamed into memory:

```
-- Lingo syntax
on exitFrame
    if member("Splash Screen").percentStreamed < 60 then
        _movie.go(_movie.frame)
    end if
end

// JavaScript syntax
function exitFrame() {
```

```
    var ssStrm = member("Splash Screen").percentStreamed;
    if (ssStrm < 60) {
        _movie.go(_movie.frame);
    }
}
```

See also

percentPlayed

period

Usage

```
timeoutObject.period
```

Description

Object property; the number of milliseconds between timeout events sent by the `timeoutObject` to the timeout handler.

This property can be tested and set.

Example

This `timeout` handler decreases the timeout's `period` by one second each time it's invoked, until a minimum period of 2 seconds (2000 milliseconds) is reached:

```
on handleTimeout timeoutObject
    if timeoutObject.period > 2000 then
        timeoutObject.period = timeoutObject.period - 1000
    end if
end handleTimeout
```

See also

name (timeout), persistent, target, time (timeout object), timeout(), timeoutHandler, timeoutList

persistent

Usage

```
timeoutObject.persistent
```

Description

Object property; determines whether the given *timeoutObject* is removed from `the timeoutList` when the current movie stops playing. If `TRUE`, *timeoutObject* remains active. If `FALSE`, the timeout object is deleted when the movie stops playing. The default value is `FALSE`.

Setting this property to `TRUE` allows a timeout object to continue generating timeout events in other movies. This is useful when one movie branches to another with the `go to movie` command.

Example

The following statement creates a timeout object and making it as persistent.

```
-- Lingo syntax
```

```
gTO = timeout().new("test",50000,"sampleTimeout",0)
gTO.persistent = TRUE

// JavaScript syntax
_global.gTO = new timeout("test",50000,"sampleTimeout",0)
_global.gTO.persistent = 1;
```

See also

name (timeout), period, target, time (timeout object), timeout(), timeoutHandler, timeoutList

picture (Member)

Usage

```
-- Lingo syntax
memberObjRef.picture

// JavaScript syntax
memberObjRef.picture;
```

Description

Cast member property; determines which image is associated with a bitmap, text, or PICT cast member. To update changes to a cast member's registration point or update changes to an image after relinking it using the `fileName` property, use the following statement:

```
member(whichCastMember).picture = member(whichCastMember).picture
```

where you replace *`whichCastMember`* with the name or number of the affected cast member.

Because changes to cast members are stored in RAM, this property is best used during authoring. Avoid setting it in projectors.

The property can be tested and set.

Example

This statement sets the variable named `pictHolder` to the image in the cast member named Sunset:

```
-- Lingo syntax
pictHolder = member("Sunset").picture

// JavaScript syntax
var pictHolder = member("Sunset").picture;
```

See also

type (sprite)

picture (Window)

Usage

```
-- Lingo syntax
windowObjRef.picture
```

```
// JavaScript syntax
windowObjRef.picture;
```

Description

Window property; provides a way to get a picture of the current contents of a window—either the Stage window or a movie in a window (MIAW). Read-only.

You can apply the resulting bitmap data to an existing bitmap or use it to create a new one.

If no picture exists, this property returns `VOID` (Lingo) or `null` (JavaScript syntax).

Example

This statement grabs the current content of the Stage and places it into a bitmap cast member:

```
-- Lingo syntax
member("Stage image").picture = _movie.stage.picture

// JavaScript syntax
member("Stage image").picture = _movie.stage.picture;
```

See also

`Window`

platform

Usage

```
the platform
```

Description

System property; indicates the platform type for which the projector was created.

This property can be tested but not set.

Possible values are the following:

Possible value	Corresponding platform
`Mac,PowerPC`	PowerPC Mac
`Windows,32`	Windows 95 or Windows NT

For forward compatibility and to allow for addition of values, it is better to test the platform by using `contains`.

Example

This statement returns the platform which the movie has been created.

```
-- Lingo syntax
if the platform contains "Windows,32" then
    alert ("This movie has been created using Windows")
end if
```

See also

`runMode`

playBackMode

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.playBackMode

// JavaScript syntax
memberOrSpriteObjRef.playBackMode;
```

Description

Cast member and sprite property; controls the tempo of a Flash movie or animated GIF cast member with the following values:

- `#normal` (*default*)—Plays the Flash movie or GIF file as close to the original tempo as possible.
- `#lockStep`—Plays the Flash movie or GIF file frame for frame with the Director movie.
- `#fixed`—Plays the Flash movie or GIF file at the rate specified by the `fixedRate` property.

This property can be tested and set.

Example

This sprite script sets the frame rate of a Flash movie sprite to match the frame rate of the Director movie:

```
-- Lingo syntax
property spriteNum

on beginSprite(me)
    sprite(spriteNum).playBackMode = #lockStep
end

// JavaScript syntax
function beginSprite() {
    sprite(this.spriteNum).playBackMode = symbol("lockStep");
}
```

See also

`fixedRate`

playing

Usage

```
-- Lingo syntax
spriteObjRef.playing

// JavaScript syntax
spriteObjRef.playing;
```

Description

Flash sprite property; indicates whether a Flash movie is playing (`TRUE`) or stopped (`FALSE`).

This property can be tested but not set.

Example

This frame script checks to see if the Flash movie sprite in channel 5 is playing and, if it is not, starts the movie:

```
-- Lingo syntax
on enterFrame
    if not sprite(5).playing then
        sprite(5).play()
    end if
end

// JavaScript syntax
function enterFrame() {
    var plg = sprite(5).playing;
    if (plg == 0) {
        sprite(5).play();
    }
}
```

playing (3D)

Usage

```
member(whichCastmember).model(whichModel).keyframePlayer.playing
member(whichCastmember).model(whichModel).bonesPlayer.playing
```

Description

3D `#keyframePlayer` and `#bonesPlayer` modifier property; indicates whether the modifier's animation playback engine is running (`TRUE`) or if it's paused (`FALSE`).

This property can be tested but not set.

Example

This statement shows that the `#keyframePlayer` animation playback engine for the model named Alien3 is currently running.

```
put member("newaliens").model("Alien3").keyframePlayer.playing
-- 1
```

See also

`play() (3D)`, `pause() (3D)`, `playlist`, `queue() (3D)`

playlist

Usage

```
member(whichCastmember).model(whichModel).keyframePlayer.playlist
member(whichCastmember).model(whichModel).bonesPlayer.playlist
```

Description

3D `#keyframePlayer` and `#bonesPlayer` modifier property; returns a linear list of property lists, each representing a motion queued for playback by the modifier.

Each property list will have the following properties:

- `#name` is the name of the motion to be played.
- `#loop` indicates whether the motion's playback should be looped.

- `#startTime` is the time, in milliseconds at which playback of the animation should begin.
- `#endTime` is the time, in milliseconds at which playback of the animation ends or when the motion should be looped. A negative value indicates that the motion should be played to the end.
- `#scale` is rate of play for the motion that is to be multiplied by the modifier's playRate property to determine the actual speed of the motion's playback.

The `playlist` property can be tested but not set. Use the `queue()`, `play()`, `playNext()`, and `removeLast()` commands to manipulate it.

Example

The following statement displays the currently queued motions for the model Stroller in the Message window.

```
-- Lingo syntax
put member("3Dobjects").model("Stroller").bonesPlayer.playList
// JavaScript syntax
put(member("3Dobjects").getPropRef("model",2).bonesPlayer.playList);
```

See also

`play() (3D)`, `playNext() (3D)`, `removeLast()`, `queue() (3D)`

playRate (3D)

Usage

```
member(whichCastmember).model(whichModel).bonesPlayer.playRate
member(whichCastmember).model(whichModel).keyframePlayer.playRate
```

Description

3D `#keyframePlayer` and `#bonesPlayer` modifier property; scale multiplier for the local time of motions being played. This property only partially determines the speed at which motions are executed by the model.

The playback of a motion by a model is the result of either a `play()` or `queue()` command. The `scale` parameter of the `play()` or `queue()` command is multiplied by the modifier's `playRate` property, and the resulting value is the speed at which the particular motion will be played back.

Example

This statement sets the `playRate` property of the `keyframePlayer` modifier for the model named GreenAlien to 3:

```
member("newAliens").model("GreenAlien").keyframePlayer.playRate = 3
```

See also

`play() (3D)`, `queue() (3D)`, `playlist`, `currentTime (3D)`

playRate (DVD)

Usage

```
-- Lingo syntax
dvdObjRef.playRate

// JavaScript syntax
dvdObjRef.playRate;
```

Description

DVD property; specifies the rate at which a DVD plays forward or backward from the current location. Read/write.

A negative value plays the DVD backward, and a positive value plays the DVD forward.

See also

DVD

playRate

Usage

```
-- Lingo syntax
spriteObjRef.playRate

// JavaScript syntax
spriteObjRef.playRate;
```

Description

Digital video sprite property; controls the rate at which a digital video in a specific channel plays. The movie rate is a value specifying the playback of the digital video. A value of 1 specifies normal forward play, -1 specifies reverse, and 0 specifies stop. Higher and lower values are possible. For example, a value of 0.5 causes the digital video to play slower than normal. However, frames may be dropped when the `playRate` sprite property exceeds 1. The severity of frame dropping depends on factors such as the performance of the computer the movie is playing on and whether the digital video sprite is stretched.

This property can be tested and set.

Example

This statement sets the rate for a digital video in sprite channel 9 to normal playback speed:

```
-- Lingo syntax
sprite(9).playRate = 1

// JavaScript syntax
sprite(9).playRate = 1;
```

This statement causes the digital video in sprite channel 9 to play in reverse:

```
-- Lingo syntax
sprite(9).playRate = -1

// JavaScript syntax
sprite(9).playRate = -1;
```

See also

duration (Member), currentTime (QuickTime, AVI)

playRate (Windows Media)

Usage

```
-- Lingo syntax
windowsMediaObjRef.playRate
```

```
// JavaScript syntax
windowsMediaObjRef.playRate;
```

Description

Windows Media property. Determines the playback rate of a Windows Media cast member. Read only.

Example

This statement displays in the Message window the playback rate of cast member 10:

```
-- Lingo syntax
trace(member(10).playRate)

// JavaScript syntax
trace(member(10).playRate);
```

See also

Windows Media

pointAtOrientation

Usage

```
member(whichCastmember).model(whichModel).pointAtOrientation
member(whichCastmember).group(whichGroup).pointAtOrientation
member(whichCastmember).light(whichLight).pointAtOrientation
member(whichCastmember).camera(whichCamera).pointAtOrientation
```

Description

3D model, light, group and camera property; allows you to get or set how the referenced object responds to the `pointAt` command. This property is a linear list of two object-relative vectors, the first vector in the list defines which direction is considered the object's front direction, the second defines which direction is considered the object's up direction.

The object's front and up directions do not need to be perpendicular to each other, but they should not be parallel to each other.

Example

This statement displays the object-relative front direction and up direction vectors of the model named bip01:

```
put member("scene").model("bip01").pointAtOrientation
-- [vector(0.0000, 0.0000, 1.0000), vector(0.0000, 1.0000, 0.0000)]
```

See also

pointAt

pointOfContact

Usage

```
collisionData.pointOfContact
```

Description

3D `collisionData` property; returns a vector describing the point of contact in a collision between two models.

The `collisionData` object is sent as an argument with the `#collideWith` and `#collideAny` events to the handler specified in the `registerForEvent`, `registerScript`, and `setCollisionCallback` commands.

The `#collideWith` and `#collideAny` events are sent when a collision occurs between models to which collision modifiers have been added. The `resolve` property of the models' modifiers must be set to `TRUE`.

This property can be tested but not set.

Example

This example has two parts. The first part is the first line of code, which registers the #explode handler for the `#collideAny` event. The second part is the `#explode` handler. When two models in the cast member MyScene collide, the `#explode` handler is called and the `collisionData` argument is sent to it. The first nine lines of the `#explode` handler create the model resource named SparkSource and set its properties. This model resource is a single burst of particles. The tenth line of the handler creates a model named SparksModel using the model resource named SparkSource. The last line of the handler sets the position of SparksModel to the position where the collision occurred. The overall effect is a burst of sparks caused by a collision.

```
member("MyScene").registerForEvent(#collideAny, #explode, 0)

on explode me, collisionData
    nmr = member("MyScene").newModelResource("SparkSource",#particle)
    nmr.emitter.mode = #burst
    nmr.emitter.loop = 0
    nmr.emitter.minSpeed = 30
    nmr.emitter.maxSpeed = 50
    nmr.emitter.direction = vector(0, 0, 1)
    nmr.colorRange.start = rgb(0, 0, 255)
    nmr.colorRange.end = rgb(255, 0, 0)
    nmr.lifetime = 5000
    nm = member("MyScene").newModel("SparksModel", nmr)
    nm.transform.position = collisionData.pointOfContact
end
```

See also

`modelA`, `modelB`

position (transform)

Usage

```
member(whichCastmember).node(whichNode).transform.position
member(whichCastmember).node(whichNode).getWorldTransform().position
transform.position
```

Description

3D property; allows you to get or set the positional component of a transform. A transform defines a scale, position and rotation within a given frame of reference. The default value of this property is `vector(0,0,0)`.

A node can be a camera, group, light or model object. Setting the `position` of a node's transform defines that object's position within the transform's frame of reference. Setting the position property of an object's world relative transform using `getWorldTransform().position` defines the object's position relative to the world origin. Setting the position property of an object's parent relative transform using `transform.position` defines the object's position relative to its parent node.

The `worldPosition` property of a model, light, camera or group object is a shortcut to the `getWorldTransform().position` version of this property for that object.

Example

The following statement displays the parent-relative position of the model named Sphere01.

```
-- Lingo syntax
    put(member("3Dobjects").model("Sphere01").transform.position)

// JavaScript syntax
    put(member("3Dobjects").getPropRef("model",2).transform.position);
```

See also

`transform (property)`, `getWorldTransform()`, `rotation (transform)`, `scale (transform)`

positionReset

Usage

```
member(whichCastmember).model(whichModel).bonesPlayer.positionReset
member(whichCastmember).model(whichModel).keyframePlayer.positionReset
```

Description

3D `keyframePlayer` and `bonesPlayer` modifier property; indicates whether the model returns to its starting position after the end of a motion (`TRUE`) or not (`FALSE`).

The default value for this property is `TRUE`.

Example

This statement prevents the model Monster from returning to its original position when it finishes the execution of a motion:

```
member("NewAlien").model("Monster").keyframePlayer.positionReset = FALSE
```

See also

`currentLoopState`

posterFrame

Usage

```
-- Lingo syntax
memberObjRef.posterFrame

// JavaScript syntax
memberObjRef.posterFrame;
```

Description

Flash cast member property; controls which frame of a Flash movie cast member is used for its thumbnail image. This property specifies an integer corresponding to a frame number in the Flash movie.

This property can be tested and set. The default value is 1.

Example

This handler accepts a reference to a Flash movie cast member and a frame number as parameters, and it then sets the thumbnail of the specified movie to the specified frame number:

```
-- Lingo syntax
on resetThumbnail(whichFlashMovie, whichFrame)
    member(whichFlashMovie).posterFrame = whichFrame
end

// JavaScript syntax
function resetThumbnail(whichFlashMovie, whichFrame) {
    member(whichFlashMovie).posterFrame = whichFrame;
}
```

preferred3dRenderer

Usage

```
-- Lingo syntax
_movie.preferred3dRenderer

// JavaScript syntax
_movie.preferred3dRenderer;
```

Description

Movie property; allows you to get or set the default renderer used to draw 3D sprites within a particular movie if that renderer is available on the client machine. Read/write.

If the specified renderer is not available on the client machine, the movie selects the most suitable available renderer.

The possible values for this property are as follows:

- `#openGL` specifies the openGL drivers for a hardware acceleration that work with both Mac and Windows platforms.
- `#directX9` specifies the DirectX® 9 drivers for hardware acceleration that work only with Windows platforms. #auto sets the renderer to DirectX 9 on Windows. In Mac® -Intel®, only #OpenGL renderer is available. DirectX 9 has been added to the Preferred 3D rendered menu in the movie tab of the Property Inspector. There is no change in the existing functionality when you use DirectX 9. You may however, experience enhanced performance.
- `#directX7_0` specifies the DirectX 7 drivers for hardware acceleration that work only with Windows platforms.
- `#directX5_2` specifies the DirectX 5.2 drivers for hardware acceleration that work only with Windows platforms.
- `#software` specifies the Director built-in software renderer that works with both Mac and Windows platforms.
- `#auto` specifies that the most suitable renderer should be chosen. This is the default value for this property.

The value set for this property is used as the default for the Renderer Services object's `renderer` property.

This property differs from the `getRendererServices()` object's `renderer` property in that the `preferred3dRenderer` specifies the preferred renderer to use, whereas the `getRendererServices()` object's `renderer` property indicates what renderer is actually being used by the movie.

Shockwave Player users have the option of specifying the renderer of their choice using the 3D Renderer context menu in Shockwave Player. If the user selects the "Obey content settings" option, the renderer specified by the `renderer` or `preferred3DRenderer` property is used to draw the movie (if available on the user's system), otherwise, the renderer selected by the user is used.

Example

This statement allows the movie to pick the best 3D renderer available on the user's system:

```
-- Lingo syntax
_movie.preferred3dRenderer = #auto

// JavaScript syntax
_movie.preferred3dRenderer = "auto";
```

See also

`getRendererServices()`, `Movie`, `renderer`

preLoad (3D)

Usage

```
member(whichCastmember).preload
memberReference.preload
```

Description

3D property; allows you to get or set whether data is preloaded before playing (`TRUE`), or is streamed while playing (`FALSE`). This property can be used only with linked files. The default value is `FALSE`.

In Director, setting the `preLoad` property to `TRUE` causes the cast member to load completely before playback starts. In Shockwave Player, setting the `preLoad` property to `TRUE` causes the cast member to begin streaming when the movie starts playing. Before perfoming any Lingo operations on a 3D cast member that is streaming, be sure to check that the cast member's `state` property has a value greater than or equal to 2.

Example

This statement sets the `preload` property of the cast member PartyScene to `FALSE`, which allows externally linked media to stream into PartyScene during playback:

```
member("PartyScene").preload = FALSE
member("3D world").preload
```

See also

`state (3D)`

preLoad (Member)

Usage

```
-- Lingo syntax
```

```
memberObjRef.preLoad

// JavaScript syntax
memberObjRef.preLoad;
```

Description

Cast member property; determines whether the digital video cast member specified by *whichCastMember* can be preloaded into memory (TRUE) or not (FALSE, default). The TRUE status has the same effect as selecting Enable Preload in the Digital Video Cast Member Properties dialog box.

For Flash movie cast members, this property controls whether a Flash movie must load entirely into RAM before the first frame of a sprite is displayed (TRUE), or whether the movie can stream into memory as it plays (FALSE, default). This property works only for linked Flash movies whose assets are stored in an external file; it has no effect on members whose assets are stored in the cast. The streamMode and bufferSize properties determine how the cast member is streamed into memory.

This property can be tested and set.

Example

This statement reports in the Message window whether the QuickTime movie Rotating Chair can be preloaded into memory:

```
-- Lingo syntax
put(member("Rotating Chair").preload)

// JavaScript syntax
put(member("Rotating Chair").preload);
```

See also

bufferSize, streamMode

preLoadEventAbort

Usage

```
-- Lingo syntax
_movie.preLoadEventAbort

// JavaScript syntax
_movie.preLoadEventAbort;
```

Description

Movie property; specifies whether pressing keys or clicking the mouse can stop the preloading of cast members (TRUE) or not (FALSE, default). Read/write.

Setting this property affects the current movie.

Example

This statement lets the user stop the preloading of cast members by pressing keys or clicking the mouse button:

```
-- Lingo syntax
_movie.preLoadEventAbort = TRUE

// JavaScript syntax
_movie.preLoadEventAbort = true;
```

See also

Movie

preLoadMode

Usage

```
-- Lingo syntax
castObjRef.preLoadMode

// JavaScript syntax
castObjRef.preLoadMode;
```

Description

Cast library property; determines the preload mode of a specified cast library. Read/write.

Valid values of `preLoadMode` are:

- 0. Load the cast library when needed. This is the default value.
- 1. Load the cast library before frame 1.
- 2. Load the cast library after frame 1.

Setting this property has the same effect as setting Load Cast in the Cast Properties dialog box.

Example

The following statement tells Director to load the members of the cast named Buttons before the movie enters frame 1:

```
-- Lingo syntax
castLib("Buttons").preLoadMode = 1

// JavaScript syntax
castLib("Buttons").preLoadMode = 1;
```

See also

Cast Library

preLoadRAM

Usage

```
the preLoadRAM
```

Description

System property; specifies the amount of RAM that can be used for preloading a digital video. This property can be set and tested.

This property is useful for managing memory, limiting digital video cast members to a certain amount of memory, so that other types of cast members can still be preloaded. When `preLoadRAM` is `FALSE`, all available memory can be used for preloading digital video cast members.

However, it's not possible to reliably predict how much RAM a digital video will require once it is preloaded, because memory requirements are affected by the content of the movie, how much compression was performed, the number of keyframes, changing imagery, and so on.

It is usually safe to preload the first couple of seconds of a video and then continue streaming from that point on.

If you know the data rate of your movie, you can estimate the setting for `preLoadRAM`. For example, if your movie has a data rate of 300K per second, set `preLoadRAM` to 600K if you want to preload the first 2 seconds of the video file. This is only an estimate, but it works in most situations.

Example

This statement sets `preLoadRAM` to 600K, to preload the first 2 seconds of a movie with a data rate of 300K per second.

```
--Lingo
set the preLoadRAM = 600

// Javascript
_system.preLoadRAM = 600;
```

See also

`loop (keyword)`, `next`

preLoadTime

Usage

```
-- Lingo syntax
memberObjRef.preLoadTime

// JavaScript syntax
memberObjRef.preLoadTime;
```

Description

Cast member and sound channel property; for cast members, specifies the amount of the Shockwave Audio (SWA) streaming cast member to download, in seconds, before playback begins or when a `preLoadBuffer` command is used. The default value is 5 seconds.

This property can be set only when the SWA streaming cast member is stopped.

For sound channels, the value is for the given sound in the queue or the currently playing sound if none is specified.

Example

The following handler sets the preload download time for the SWA streaming cast member Louis Armstrong to 6 seconds. The actual preload occurs when a `preLoadBuffer` or `play` command is issued.

```
-- Lingo syntax
on mouseDown
    member("Louis Armstrong").stop()
    member("Louis Armstrong").preLoadTime = 6
end

// JavaScript syntax
function mouseDown() {
    member("Louis Armstrong").stop();
    member("Louis Armstrong").preLoadTime = 6;
```

```
}
```

This statement returns the preLoadTime of the currently playing sound in sound channel 1:

```
-- Lingo syntax
put sound(1).preLoadTime

// JavaScript syntax
trace(sound(1).preLoadTime);
```

See also

preLoadBuffer()

primitives

Usage

```
getRendererServices().primitives
```

Description

3D function; returns a list of the primitive types that can be used to create new model resources.

Example

This statement display the available primitive types:

```
--Lingo
put getRendererServices().primitives

// Javascript
put( getRendererServices().primitives);
```

See also

getRendererServices(), newModelResource

productName

Usage

```
-- Lingo syntax
_player.productName

// JavaScript syntax
_player.productName;
```

Description

Player property; returns the name of the Director application. Read-only.

Example

This statement displays in the Message window the name of the Director application.

```
-- Lingo syntax
trace(_player.productName)

// JavaScript syntax
```

```
trace(_player.productName);
```

See also

Player

productVersion

Usage

```
-- Lingo syntax
_player.productVersion

// JavaScript syntax
_player.productVersion;
```

Description

Player property; returns the version number of the Director application. Read-only.

Example

This statement displays in the Message window the version of the Director application.

```
-- Lingo syntax
trace(_player.productVersion)

// JavaScript syntax
trace(_player.productVersion);
```

See also

Player

projection

Usage

```
sprite(whichSprite).camera.projection
camera(whichCamera).projection
member(whichCastmember).camera(whichCamera).projection
```

Description

3D property; allows you to get or set the projection style of the camera. Possible values are `#perspective` (the default) and `#orthographic`.

When projection is `#perspective`, objects closer to the camera appear larger than objects farther from the camera, and the `projectionAngle` or `fieldOfView` properties specify the vertical projection angle (which determines how much of the world you see). The horizontal projection angle is determined by the aspect ratio of the camera's `rect` property.

When projection is `#orthographic`, the apparent size of objects does not depend on distance from the camera, and the `orthoHeight` property specifies how many world units fit vertically into the sprite (which determines how much of the world you see). The orthographic projection width is determined by the aspect ratio of the camera's `rect` property.

Example

This statement sets the `projection` property of the camera of sprite 5 to `#orthographic`:

```
sprite(5).camera.projection = #orthographic
```

See also

`fieldOfView (3D)`, `orthoHeight`, `fieldOfView (3D)`

purgePriority

Usage

```
-- Lingo syntax
memberObjRef.purgePriority

// JavaScript syntax
memberObjRef.purgePriority;
```

Description

Member property; specifies the purge priority of a cast member. Read/write.

A cast member's purge priorities determine the priority that Director follows to choose which cast members to delete from memory when memory is full. The higher the purge priority, the more likely that the cast member will be deleted. The following `purgePriority` settings are available:

- 0—Never
- 1—Last
- 2—Next
- 3—Normal (default)

The Normal setting lets Director purge cast members from memory at random. The Next, Last, and Never settings allow some control over purging, but Last or Never may cause your movie to run out of memory if several cast members are set to these values.

Setting `purgePriority` for cast members is useful for managing memory when the size of the movie's cast library exceeds the available memory. As a general rule, you can minimize pauses while the movie loads cast members and reduce the number of times Director reloads a cast member by assigning a low purge priority to cast members that are used frequently in the course of the movie.

Example

This statement sets the purge priority of cast member Background to 3, which makes it one of the first cast members to be purged when memory is needed:

```
-- Lingo syntax
member("Background").purgePriority = 3

// JavaScript syntax
member("Background").purgePriority = 3;
```

See also

`Member`

quad

Usage

```
-- Lingo syntax
spriteObjRef.quad

// JavaScript syntax
spriteObjRef.quad;
```

Description

Sprite property; contains a list of four points, which are floating point values that describe the corner points of a sprite on the Stage. Read/write.

The points of the quad are organized in the following order: upper left, upper right, lower right, and lower left.

The points themselves can be manipulated to create perspective and other image distortions.

After you manipulate the quad of a sprite, you can reset it to the Score values by turning off the scripted sprite with `puppetSprite(intSpriteNum, FALSE)`. When the quad of a sprite is disabled, you cannot rotate or skew the sprite.

Example

This statement displays a typical list describing a sprite:

```
-- Lingo syntax
put(sprite(1).quad)

// JavaScript syntax
put(sprite(1).quad);
```

When modifying the `quad` sprite property, you must reset the list of points after changing any of the values. This is because when you set a variable to the value of a property, you are placing a copy of the list, not the list itself, in the variable. To effect a change, use syntax like this (applies to Lingo only):

```
-- Lingo syntax
currQuadList = sprite(5).quad
currQuadList[1] = currQuadList[1] + point(50, 50)
sprite(5).quad = currQuadList
```

See also

`point()`, `puppetSprite()`, `Sprite`

quality

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.quality

// JavaScript syntax
memberOrSpriteObjRef.quality;
```

Description

Flash cast member and sprite property; controls whether Director uses anti-aliasing to render a Flash movie sprite, producing high-quality rendering but possibly slower movie playback. The `quality` property can have these values:

• `#autoHigh`—Director starts by rendering the sprite with anti-aliasing. If the actual frame rate falls below the movie's specified frame rate, Director turns off anti-aliasing. This setting gives precedence to playback speed over visual quality.

• `#autoLow`—Director starts by rendering the movie without anti-aliasing. If the Flash player determines that the computer processor can handle it, anti-aliasing is turned on. This setting gives precedence to visual quality whenever possible.

• `#high` (default)—The movie always plays with anti-aliasing.

• `#low`—The movie always plays without anti-aliasing.

The `quality` property can be tested and set.

Example

The following sprite script checks the color depth of the computer on which the movie is playing. If the color depth is set to 8 bits or less (256 colors), the script sets the quality of the sprite in channel 5 to `#low`.

```
-- Lingo syntax
on beginSprite me
    if _system.colorDepth <= 8 then
        sprite(1).quality = #low
    end if
end

// JavaScript syntax
function beginSprite() {
    var clrDp = _system.colorDepth;
    if (clrDp <= 8) {
        sprite(1).quality = symbol("low");
    }
}
```

quality (3D)

Usage

```
member(whichCastmember).texture(whichTexture).quality
member(whichCastmember).shader(whichShader).texture(whichTexture).quality
member(whichCastmember).model(whichModel).shader.texture(whichTexture).quality
member( whichCastmember ).model( whichModel ).shader.texturelist[TextureListIndex].quality
member(whichCastmember).model(whichModel).shaderList[shaderListIndex].
texture(whichTexture).quality
member( whichCastmember ).model( whichModel ).shaderList[ shaderListIndex ]. texturelist[
TextureListIndex ].quality
```

Description

3D texture property; lets you get or set the image quality of a texture by controlling the level of mipmapping applied to the texture. Mipmapping is a process by which additional versions of the texture image are created in several sizes that are smaller than the original image. The 3D Xtra extension then uses whichever version of the image is most appropriate to the current size of the model on the screen and changes the version of the image that is being used when needed. Trilinear mipmapping is higher in quality and uses more memory than bilinear mipmapping.

Mipmapping is not the same as filtering, although both improve texture appearance. Filtering spreads errors out across the texture's area so that errors are less concentrated. Mipmapping resamples the image to make it the appropriate size.

This property can have the following values:

- `#low` is the same as off, and mipmapping is not used for the texture.
- `#medium` enables a low-quality (bilinear) mipmapping for the texture.
- `#high` enables a high-quality (trilinear) mipmapping for the texture.

The default is `#low`.

Example

This statement sets the `quality` property of the texture Marsmap to `#medium`:

```
member("scene").texture("Marsmap").quality = #medium
```

See also

`nearFiltering`

radius

Usage

```
modelResourceObjectReference.radius
member(whichCastmember).modelResource(whichModelResource).radius
```

Description

3D model property; when used with model resource of type `#sphere` or `#cylinder`, allows you to get or set the radius of the model.

The `radius` property determines the sweep radius used to generate the model resource. This property's value must always be set to greater than 0.0, and has a default value of 25.0.

Example

This statement shows the radius of the model resource Cylinder01.

```
--Lingo
put member("3Dobjects").modelResource("Cylinder01").radius

// Javascript
put(member("3Dobjects").getPropRef("modelResource",11).radius);
```

randomSeed

Usage

```
the randomSeed
```

Description

System property; specifies the seed value used for generating random numbers accessed through the `random()` function.

Using the same seed produces the same sequence of random numbers. This property can be useful for debugging during development. Using the `ticks` property is an easy way to produce a unique random seed since the `ticks` value is highly unlikely to be duplicated on subsequent uses.

This property can be tested and set.

Example

This statement displays the random seed number in the Message window.

```
--Lingo
put the randomSeed

// Javascript
put(_system.randomSeed);
```

See also

random(), milliseconds

recordFont

Usage

```
recordFont(whichCastMember, font {[,face]} {,[bitmapSizes]} {,characterSubset} {,
userFontName})
```

Description

Command; embeds a TrueType or Type 1 font as a cast member. Once embedded, these fonts are available to the author just like other fonts installed in the system.

You must create an empty font cast member with the `new()` command before using `recordFont`.

- *font*—Name of original font to be recorded.
- *face*—List of symbols indicating the face of the original font; possible values are `#plain`, `#bold`, `#italic`. If you do not provide a value for this argument, `#plain` is used.
- *bitmapSizes*—List of integers specifying the sizes for which bitmaps are to be recorded. This argument can be empty. If you omit this argument, no bitmaps are generated. These bitmaps typically look better at smaller point sizes (below 14 points) but take up more memory.
- *characterSubset*—String of characters to be encoded. Only the specified characters will be available in the font. If this argument is, all characters are encoded. If only certain characters are encoded but an unencoded character is used, that character is displayed as an empty box.
- *userFontName*—A string to use as the name of the newly recorded font cast member.

The command creates a Shock Font in *whichCastMember* using the font named in the *font* argument. The value returned from the command reports whether the operation was successful. Zero indicates success.

Example

This statement creates a simple Shock Font using only the two arguments for the cast member and the font to record:

```
myNewFontMember = new(#font)
recordFont(myNewFontMember, "Lunar Lander")
```

This statement specifies the bitmap sizes to be generated and the characters for which the font data should be created:

```
myNewFontMember = new(#font)
recordfont(mynewmember,"lunar lander",[],[14, 18, 45], "Lunar Lander Game High Score First
Last Name")
```

Note: *Since recordFont resynthesizes the font data rather than using it directly, there are no legal restrictions on Shock Font distribution.*

See also

`new()`

rect (camera)

Usage

```
sprite(whichSprite).camera(whichCamera).rect
```

Description

3D camera property; allows you to get or set the rectangle that controls the size and position of the camera. This rectangle is analogous to the rectangle you see through the eyepiece of a real camera.

The default value for the rect property for all cameras rect(0,0,1,1) which makes them invisible until you change the setting. However, when `sprite.camera(1)` is rendered, its rect is reset to `rect(0, 0, sprite(whichSprite).width, sprite(whichSprite).height)` so that the camera fills the screen. All camera rect coordinates are given relative to the top left corner of the sprite.

If *`whichCamera`* is greater than 1, the `rect` is not scaled when the sprite is scaled, so it will be necessary to manage that with script if desired.

When *`whichCamera`* is greater than 1, the `rect.top` and `rect.left` properties must be greater than or equal to the `rect.top` and `rect.left` settings for `sprite.camera(1)`.

Example

This statement displays the coordinates of bitmap cast member 1.

```
-- Lingo syntax
put(member(1).rect)

// JavaScript syntax
put(member(1).rect);
```

See also

`cameraPosition`, `cameraRotation`

rect (Image)

Usage

```
-- Lingo syntax
imageObjRef.rect

// JavaScript syntax
imageObjRef.rect;
```

Description

Image property. Returns a rectangle describing the size of a given image. Read-only.

The returned rectangle coordinates are given relative to the top left corner of the image. Therefore, the left and top values of the rectangle are 0, and the right and bottom values are the width and height of the cast member.

Example

This statement returns the rectangle of the 300 x 400 pixel member Sunrise in the message window:

```
-- Lingo syntax
member("Sunrise").image.rect -- rect(0, 0, 300, 400)

// JavaScript syntax
member("Sunrise").image.rect; // rect(0, 0, 300, 400)
```

This Lingo looks at the first 50 cast members and displays the rectangle and name of each cast member that is a bitmap:

```
-- Lingo syntax
on showAllRects
    repeat with x = 1 to 50
    if member(x).type = #bitmap then
        put member(x).image.rect && "-" && member(x).name
    end if
    end repeat
end

// JavaScript syntax
function showAllRects() {
    var x = 1;
    while(x < 51) {
        var tp = member(x).type;
        if (tp == "bitmap") {
            trace(member(x).image.rect + " - " + member(x).name);
            i++;
        }
    }
}
```

See also

height, image(), width

rect (Member)

Usage

```
-- Lingo syntax
memberObjRef.rect

// JavaScript syntax
memberObjRef.rect;
```

Description

Member property; specifies the left, top, right, and bottom coordinates, returned as a rectangle, for the rectangle of any graphic cast member, such as a bitmap, shape, movie, or digital video. Read-only for all cast members, read/write for field cast members only.

For a bitmap, the `rect` property is measured from the upper left corner of the bitmap, instead of from the upper left corner of the easel in the Paint window.

For an Xtra extension cast member, the `rect` property is a rectangle whose upper left corner is at (0,0).

Example

This statement displays the coordinates of bitmap cast member 20:

```
-- Lingo syntax
put(member(20).rect)

// JavaScript syntax
put(member(20).rect);
```

This statement sets the coordinates of bitmap cast member Banner:

```
-- Lingo syntax
member("Banner").rect = rect(100, 150, 300, 400)

// JavaScript syntax
member("Banner").rect = rect(100, 150, 300, 400);
```

See also

Member

rect (Sprite)

Usage

```
-- Lingo syntax
spriteObjRef.rect

// JavaScript syntax
spriteObjRef.rect;
```

Description

Sprite property; specifies the left, top, right, and bottom coordinates, as a rectangle, for the rectangle of any graphic sprite such as a bitmap, shape, movie, or digital video. Read/write.

Example

This statement displays the coordinates of bitmap sprite 20:

```
-- Lingo syntax
put(sprite(20).rect)

// JavaScript syntax
put(sprite(20).rect);
```

See also

rect(), Sprite

rect (Window)

Usage

```
-- Lingo syntax
windowObjRef.rect
```

```
// JavaScript syntax
windowObjRef.rect;
```

Description

Window property; specifies the left, top, right, and bottom coordinates, as a rectangle, of a window. Read/write.

If the size of the rectangle specified is less than that of the Stage where the movie was created, the movie is cropped in the window, not resized.

To pan or scale the movie playing in the window, set the `drawRect` or `sourceRect` property of the window.

Example

This statement displays the coordinates of the window named Control_panel:

```
-- Lingo syntax
put(window("Control_panel").rect)

// JavaScript syntax
put(window("Control_panel").rect);
```

See also

drawRect, sourceRect, Window

ref

Usage

```
chunkExpression.ref
```

Description

Text chunk expression property; this provides a convenient way to refer to a chunk expression within a text cast member.

Example

Without references, you would need statements like these:

```
member(whichTextMember).line[whichLine].word[firstWord..lastWord].font = "Palatino"
member(whichTextMember).line[whichLine].word[firstWord..lastWord].fontSize = 36
member(whichTextMember).line[whichLine].word[firstWord..lastWord].fontStyle = [#bold]
```

But with a `ref` property, you can refer to the same chunk as follows:

```
myRef = member(whichTextMember).line[whichLine].word[firstWord..lastWord].ref
```

The variable `myRef` is now shorthand for the entire chunk expression. This allows something like the following:

```
put myRef.font
-- "Palatino"
```

Or you can set a property of the chunk as follows:

```
myRef.fontSize = 18
myRef.fontStyle = [#italic]
```

You can get access to the string referred to by the reference using the `text` property of the reference:

```
put myRef.text
```

This would result in the actual string data, not information about the string.

reflectionMap

Usage

```
member(whichCastmember).shader(whichShader).reflectionMap
```

Description

3D shader property; allows you to get and set the texture used to create reflections on the surface of a model. This texture is applied to the third texture layer of the shader. This property is ignored if the `toon` modifier is applied to the model resource.

This helper property provides a simple interface for setting up a common use of reflection mapping. The same effect can be achieved by setting the following properties:

```
shader.textureModeList[3] = #reflection
shader.blendFunctionList[3] = #blend
shader.blendSourceList[3] = #constant
shader.blendConstantList[3] = 50.0
```

When tested, this property returns the texture associated with the model's third texture layer. The default is `void`.

Example

This statement sets the model named Sphere01 to appear to reflect the default texture of its surface.

```
-- Lingo syntax
member("3Dobjects").model("Sphere01").shader.reflectionMap=member("3Dobjects").texture[1]

// JavaScript syntax
member("3Dobjects").getPropRef("model",2).shader.reflectionMap=member("3Dobjects").getProp
Ref("texture",1);
```

See also

`textureModeList`, `blendFunctionList`, `blendConstantList`

reflectivity

Usage

```
member(whichCastmember).reflectivity
```

Description

3D shader property; allows you to get or set the shininess of the referenced member's default shader. The value is a floating point value representing the percentage of light to be reflected off the surface of a model using the default shader, from 0.0 to 100.00. The default value is 0.0.

Example

This statement sets the shininess of the default shader in the cast member named Scene to 50%.

```
-- Lingo syntax
member("Scene").reflectivity = 50

// JavaScript syntax
```

```
member("Scene").reflectivity = 50;
```

region

Usage

```
member(whichCastmember).modelResource(whichModelResource).emitter.region
modelResourceObjectReference.emitter.region
```

Description

3D emitter property; when used with a model resource whose type is `#particle`, allows you to both get and set the `region` property of the resource's particle emitter.

The region property defines the location from which particles are emitted. If its value is a single vector, then that vector is used to define a point in the 3D world from which particles will be emitted.

If its value is a list of two vectors, then those vectors are used to define the end points of a line segment from which particles will be emitted.

If its value is a list of four vectors, then those vectors are used to define the vertices of a quadrilateral from which the particles will be emitted.

The default value for this property is [vector(0,0,0)].

Example

In this example, ThermoSystem is a model resource of the type #particle. This statement specifies the four corners of a rectangle from which the particles of ThermoSystem originate.

```
member("Fires").modelResource("ThermoSystem").emitter.region = [vector(20,90,100),
vector(30,90,100), vector(30,100,100), vector(20,100,100)]
```

See also

emitter

regPoint

Usage

```
-- Lingo syntax
memberObjRef.regPoint

// JavaScript syntax
memberObjRef.regPoint;
```

Description

Member property; specifies the registration point of a cast member. Read/write.

The registration point is listed as the horizontal and vertical coordinates of a point in the form `point(`*`horizontal, vertical`*`)`. Nonvisual members such as sounds do not have a useful `regPoint` property.

You can use the `regPoint` property to animate individual graphics in a film loop, changing the film loop's position in relation to other objects on the Stage.

You can also use `regPoint` to adjust the position of a mask being used on a sprite.

When a Flash movie cast member is first inserted into the cast library, its registration point is its center and its `centerRegPoint` property is set to `TRUE`. If you subsequently use the `regPoint` property to reposition the registration point, the `centerRegPoint` property is automatically set to `FALSE`.

Example

This statement displays the registration point of the bitmap cast member Desk in the Message window:

```
-- Lingo syntax
put(member("Desk").regPoint)

// JavaScript syntax
put(member("Desk").regPoint);
```

This statement changes the registration point of the bitmap cast member Desk to the values in the list:

```
-- Lingo syntax
member("Desk").regPoint = point(300, 400)

// JavaScript syntax
member("Desk").regPoint = point(300, 400);
```

See also

`Member`, `Sprite`

regPoint (3D)

Usage

```
sprite(whichSprite).camera.backdrop[backdropIndex].regPoint
member(whichCastmember).camera(whichCamera).backdrop
[backdropIndex].regPoint
```

Description

3D backdrop and overlay property; allows you to get or set the registration point of the backdrop or overlay. The registration point represents the *x*, *y*, and *z* coordinates of the center of the backdrop or overlay in 3D space. The default value for this property is point(0,0).

Example

The following statement changes the registration point of the first backdrop of the camera of sprite 13. The backdrop's registration point will be the point (50, 0), measured from the upper left corner of the backdrop.

```
sprite(13).camera.backdrop[1].regPoint = point(50, 0)
```

See also

`loc (backdrop and overlay)`

regPointVertex

Usage

```
-- Lingo syntax
memberObjRef.regPointVertex
```

```
// JavaScript syntax
memberObjRef.regPointVertex;
```

Description

Cast member property; indicates whether a vertex of *vectorCastMember* is used as the registration point for that cast member. If the value is zero, the registration point is determined normally, using the `centerRegPoint` and `regPoint` properties. If the value is nonzero, it indicates the position in the `vertextList` of the vertex being used as the registration point. The `centerRegPoint` is set to `FALSE` and the `regPoint` is set to the location of that vertex.

Example

This statement makes the registration point for the vector shape cast member Squiggle correspond to the location of the third vertex:

```
-- Lingo syntax
member("squiggle").regPointVertex=3

// JavaScript syntax
member("squiggle").regPointVertex=3;
```

See also

`centerRegPoint`, `regPoint`

renderer

Usage

```
getRendererServices().renderer
```

Description

3D property; allows you to get or set the current renderer in use by a movie. The range of values for this property is determined by the list of available renderers returned by the Renderer Services object's `rendererDeviceList` property.

Shockwave Player users have the option of specifying the renderer of their choice using the 3D Renderer context menu in Shockwave Player. If the user selects the "Obey content settings" option, the renderer specified by the `renderer` or `preferred3DRenderer` properties is used to draw the movie (if available on the users system), otherwise the renderer selected by the user is used.

The default value for this property is determined by the `preferred3DRenderer` property.

This property returns the same value as returned by the movie property `the active3dRenderer`.

Example

This statement shows that the renderer currently being used by the user's system.

```
-- Lingo syntax
put getRendererServices().renderer

// JavaScript syntax
put( getRendererServices().renderer);
```

See also

`getRendererServices()`, `preferred3dRenderer`, `rendererDeviceList`, `active3dRenderer`

rendererDeviceList

Usage

```
getRendererServices().rendererDeviceList
```

Description

3D renderer property; returns a list of symbols identifying the renderers that are available for use on the client machine. The contents of this list determine the range of values that can be specified for the `renderer` and `preferred3DRenderer` properties. This property can be tested but not set.

This property is a list that can contain the following possible values:

- `#openGL` specifies the openGL drivers for a hardware acceleration which work with both Mac and Windows platforms.
- `#directX9` specifies the DirectX 9 drivers for hardware acceleration that work only with Windows platforms. #auto sets the renderer to DirectX 9. In Mac-Intel, only #OpenGL renderer is available.
- `#directX7_0` specifies the DirectX 7 drivers for hardware acceleration which work with Windows platforms only.
- `#directX5_2` specifies the DirectX 5.2 drivers for hardware acceleration which work with Windows platforms only.
- `#software` specifies the Director built-in software renderer which works with both Mac and Windows platforms.

Example

This statement shows the renderers available on the current system.

```
-- Lingo syntax
put getRendererServices().rendererDeviceList

// JavaScript syntax
put( getRendererServices().rendererDeviceList);
```

See also

`getRendererServices()`, `renderer`, `preferred3dRenderer`, `active3dRenderer`

renderFormat

Usage

```
member(whichCastmember).texture(whichTexture).renderFormat
member(whichCastmember).texture[index].renderFormat
member(whichCastmember).shader(whichShader).texture.renderFormat
member(whichCastmember).model(whichModel).shader.texture.renderFormat
member(whichCastmember).model(whichModel).shader.textureList[index].renderFormat
member(whichCastmember).model(whichModel).shaderList[index].texture(whichTexture).renderFo
rmat
member(whichCastmember).model(whichModel).shaderList[index].textureList[index].renderForma
t
```

Description

3D property; allows you to get or set the `textureRenderFormat` for a specific texture by specifying one of the following values:

`#default` uses the value returned by `getRendererServices().textureRenderFormat`.

```
#rgba8888
#rgba8880
#rgba5650
#rgba5550
#rgba5551
#rgba4444
```

See `textureRenderFormat` for information on these values.

Setting this property for an individual texture overrides the global setting set using `textureRenderFormat`.

The `renderFormat` property determines the pixel format the renderer uses when rendering the specified texture. Each pixel format has a number of digits, with each digit indicating the color depth being used for red, green, blue, and alpha. The value you choose determines the accuracy of the color fidelity (including the precision of the optional alpha channel) and thus the amount of memory used on the video card. You can choose a value that improves color fidelity or a value that allows you to fit more textures into memory on the video card. You can fit roughly twice as many 16-bit textures as 32-bit textures in the same space.

Example

The following statement sets the `renderFormat` property of the texture TexPic to #rgba4444. The red, blue, green, and alpha components of the texture will each be drawn using 4 bits of information.

```
-- Lingo syntax
member("3Dobjects").texture[1].renderFormat = #rgba4444
put(member("3Dobjects").texture[1].renderFormat )
```

See also

`textureRenderFormat`, `getHardwareInfo()`

renderStyle

Usage

```
member(whichCastmember).shader(whichShader).renderStyle
```

Description

3D standard shader property; allows you to get or set the `renderStyle` for a shader, as determined by the geometry of the underlying model resource. This property has the following values:

#fill specifies that the shader is drawn to completely fill the surface area of the model resource.

#wire specifies that the shader is drawn only on the edges of the faces of the model resource.

#point specifies that the shader is drawn only on the vertices of the model resource.

All shaders have access to the `#standard` shader properties; in addition to these standard shader properties shaders of the types `#engraver`, `#newsprint`, and `#painter` have properties unique to their type. For more information, see `newShader`.

Example

This statement causes the first shader to be rendered only where it lies on top of a vertex of the underlying model resource.

```
-- Lingo syntax
member("3Dobjects").shader[1].renderStyle = #point

// JavaScript syntax
 member("3Dobjects").getPropRef("shader",1).renderStyle = symbol("point");
```

resizable

Usage

```
-- Lingo syntax
windowObjRef.resizable

// JavaScript syntax
windowObjRef.resizable;
```

Description

Window property; specifies whether the window is resizable (`TRUE`, default) or not (`FALSE`).
Read/write.

Example

These statements maximize the window named Empire if the window is resizable.

```
-- Lingo syntax
if (window("Empire").resizable = TRUE) then
    window("Empire").maximize()
end if

// JavaScript syntax
if (window("Empire").resizable == true) {
    window("Empire").maximize();
}
```

See also

Window

resolution (3D)

Usage

```
member(whichCastmember).modelResource(whichModelResource).resolution
```

Description

3D property; allows you to get or set the resolution property of a model resource whose type is either `#sphere` or `#cylinder`.

Resolution controls the number of polygons used to generate the geometry of the model resource. A larger value generates more polygons and thus results in a smoother surface. The default value of this property is 20.

Example

This statement sets the resolution of the model resource named sphere01 to 10.0:

```
member("3D World").modelResource("sphere01").resolution = 10.0
```

resolution (DVD)

Usage

```
-- Lingo syntax
dvdObjRef.resolution

// JavaScript syntax
dvdObjRef.resolution;
```

Description

DVD property. Returns a property list that contains the x-axis (width) and y-axis (height) source resolution. Read-only.

Example

This statement returns a sample property list of resolutions:

```
-- Lingo syntax
trace(member(1).resolution) -- [#width: 720, #height: 480]

// JavaScript syntax
trace(member(1).resolution); // ["width": 720, "height": 480]
```

See also

DVD

resolve

Usage

```
member(whichCastmember).model(whichModel).collision.resolve
```

Description

3D collision property; allows you to get or set whether collisions are resolved when two models collide. If this property is set to TRUE for both models involved in a collision, both models come to a stop at the point of collision. If only one of the models has the resolve property set to TRUE, that model comes to a stop, and the model with the property not set, or set to FALSE, continues to move. The default value for this property is TRUE.

Example

The following statement sets the resolve property of the collision modifier applied to the model named Box to TRUE. When the model named Box collides with another model that has the #collision modifier attached, it will stop moving.

```
member("3d world").model("Box").collision.resolve = TRUE
```

See also

collisionData, collisionNormal, modelA, modelB, pointOfContact

resource

Usage

```
member(whichCastmember).model(whichModel).resource
```

Description

3D property; allows you to get or set the resource property that defines the geometry of the referenced model resource. This property also allows access to the referenced model's resource object and its associated properties.

Example

This statement displays the `radius` property of the model resource used by the second model.

```
-- Lingo syntax
put member("3Dobjects").model[2].resource.radius

// JavaScript syntax
put(member("3Dobjects").getPropRef("model",2).resource.radius);
```

right

Usage

```
-- Lingo syntax
spriteObjRef.right

// JavaScript syntax
spriteObjRef.right;
```

Description

Sprite property; indicates the distance, in pixels, of a sprite's right edge from the left edge of the Stage. Read/write.

Sprite coordinates are expressed relative to the upper left corner of the Stage.

Example

This statement returns the distance of a sprite's right edge:

```
-- Lingo syntax
put(sprite(6).right)

// JavaScript syntax
put(sprite(6).right);
```

See also

bottom, height, left, locH, locV, Sprite, top, width

right (3D)

Usage

```
member(whichCastmember).modelResource
(whichModelResource).right
modelResourceObjectReference.right
```

Description

3D property; allows you to get or set the `right` property of a model resource whose type is `#box`.

The `right` property determines whether the right of the box is sealed (`TRUE`) or open (`FALSE`). The default value is `TRUE`.

Example

This statement sets the right property of the model resource Crate to `TRUE`, meaning the right side of this box will be closed:

```
member("3D World").modelResource("crate").right = TRUE
```

See also

`bottom (3D)`, `left (3D)`, `top (3D)`

rightIndent

Usage

```
chunkExpression.rightIndent
```

Description

This statement right indents the text cast member "myText" by ten pixels.

```
-- Lingo syntax
member("myText").rightIndent=10

// JavaScript syntax
member("myText").rightIndent=10;
```

See also

`firstIndent`, `leftIndent`

rightMouseDown

Usage

```
-- Lingo syntax
_mouse.rightMouseDown

// JavaScript syntax
_mouse.rightMouseDown;
```

Description

Mouse property; indicates whether the right mouse button (Windows) or the Control+Mouse button (Mac) are being pressed (`TRUE`) or not (`FALSE`). Read-only.

On the Mac, `rightMouseDown` is `TRUE` only if the `emulateMultiButtonMouse` property is `TRUE`.

Example

This statement checks whether the right mouse button in Windows is being pressed and plays the sound Oops in sound channel 2 if it is:

```
-- Lingo syntax
if (_mouse.rightMouseDown) then
    sound(2).play(member("Oops"))
end if

// JavaScript syntax
if (_mouse.rightMouseDown) {
    sound(2).play(member("Oops"));
}
```

See also

emulateMultibuttonMouse, Mouse

rightMouseUp

Usage

```
-- Lingo syntax
_mouse.rightMouseUp

// JavaScript syntax
_mouse.rightMouseUp;
```

Description

Mouse property; indicates whether the right mouse button (Windows) or the mouse button and Control key (Mac) are currently not being pressed (TRUE) or are currently being pressed (FALSE). Read-only.

On the Mac, rightMouseUp is TRUE only if the emulateMultiButtonMouse property is TRUE.

Example

This statement checks whether the right mouse button in Windows is released and plays the sound Click Me if it is:

```
-- Lingo syntax
if (_mouse.rightMouseUp) then
    sound(2).play(member("Click Me"))
end if

// JavaScript syntax
if (_mouse.rightMouseUp) {
    sound(2).play(member("Click Me"));
}
```

See also

emulateMultibuttonMouse, Mouse

romanLingo

Usage

```
the romanLingo
```

Description

System property; specifies whether Lingo uses a single-byte (TRUE) or double-byte interpreter (FALSE).

The Lingo interpreter is faster with single-byte character sets. Some versions of Mac system software—Japanese, for example—use a double-byte character set. U.S. system software uses a single-byte character set. Normally, `romanLingo` is set when Director is first started and is determined by the local version of the system software.

If you are using a non-Roman script system but don't use any double-byte characters in your script, set this property to `TRUE` for faster execution of your Lingo scripts.

Example

This statement sets `romanLingo` to `TRUE`, which causes Lingo to use a single-byte character set:

```
-- Lingo
set the romanLingo to TRUE
```

See also

`inlineImeEnabled`

rootLock

Usage

```
member(whichCastmember).model(whichModel).keyframePlayer.rootLock
member(whichCastmember).model(whichModel).bonesPlayer.rootLock
```

Description

3D `#keyframePlayer` and `#bonesPlayer` modifier property; indicates whether the translational components of a motion are used (`FALSE`) or ignored (`TRUE`).

The default value of this property is `FALSE`.

Example

This statement forces the model named Alien3, which is the first model, to remain at its starting position while executing its motions, resulting in a character that will walk in place:

```
-- Lingo
member("newalien").model("Alien3").keyframePlayer.rootLock = 1

// Javascript
member("newalien").getPropRef("model",1).keyframePlayer.rootLock=1
```

rootNode

Usage

```
member(whichCastmember).camera(whichCamera).rootNode
sprite(whichSprite).camera.rootNode
```

Description

3D property; allows you to get or set which objects are visible within a sprite. When a camera is first created, it shows all nodes within the world. The `rootNode` property allows you to modify this by creating a different default view that limits what's shown to a particular node and its children.

For example, light C is a child of model A., if you set the `rootNode` property to `camera("defaultView").rootNode=model(A)`, the sprite will show only model A as illuminated by light C. The default is `group("world")`, meaning that all nodes are used.

Example

The following statement sets the `rootNode` of the camera of sprite 5 to the model Pluto. Only Pluto and its children will be visible in sprite 5.

```
-- Lingo
sprite(5).camera.rootNode = member("Scene").model("Pluto")

// Javascript
sprite(5).getPropRef("camera",1).rootNode=member("Scene").getPropRef("model",1);
```

rotation

Usage

```
-- Lingo syntax
spriteObjRef.rotation

// JavaScript syntax
spriteObjRef.rotation;
```

Description

Sprite property; controls the rotation of a QuickTime movie, animated GIF, Flash movie, or bitmap sprite within a sprite's bounding rectangle, without rotating that rectangle or the sprite's controller (in the case of QuickTime). Read/write.

In effect, the sprite's bounding rectangle acts as a window through which you can see the Flash or QuickTime movie. The bounding rectangles of bitmaps and animated GIFs change to accommodate the rotating image.

Score rotation works for a Flash movie only if `obeyScoreRotation` is set to `TRUE`.

A Flash movie rotates around its origin point as specified by its `originMode` property. A QuickTime movie rotates around the center of the bounding rectangle of the sprite. A bitmap rotates around the registration point of the image.

For QuickTime media, if the sprite's `crop` property is set to `TRUE`, rotating the sprite frequently moves part of the image out of the viewable area; when the sprite's `crop` property is set to `FALSE`, the image is scaled to fit within the bounding rectangle (which may cause image distortion).

You specify the rotation in degrees as a floating-point number.

The Score can retain information for rotating an image from +21,474,836.47 to -21,474,836.48 , allowing 59,652 full rotations in either direction.

When the rotation limit is reached (slightly past the 59,652th rotation), the rotation resets to +116.47 or -116.48 — not 0.00 . This is because +21,474,836.47 is equal to +116.47 , and -21,474,836.48 is equal to -116.48 (or +243.12). To avoid this reset condition, when you use script to perform continuous rotation, constrain the angles to ±360 .

The default value of this property is 0.

Example

This behavior causes a sprite to rotate continuously by 2° every time the playhead advances, limiting the angle to 360°:

```
-- Lingo syntax
property spriteNum

on prepareFrame me
    sprite(spriteNum).rotation = integer(sprite(spriteNum).rotation + 2) mod 360
end

// JavaScript syntax
function prepareFrame() {
    sprite(this.spriteNum).rotation = parseInt(sprite(this.spriteNum).rotation + 2) % 360;
}
```

The following frame script keeps the playhead looping in the current frame while it rotates a QuickTime sprite in channel 5 a full 360° in 16° increments. When the sprite has been rotated 360°, the playhead continues to the next frame.

```
-- Lingo syntax
on rotateMovie(whichSprite)
    repeat with i = 1 to 36
        sprite(whichSprite).rotation = i * 10
        _movie.updateStage()
    end repeat
end

// JavaScript syntax
function rotateMovie(whichSprite) {
    for (var i = 1; i <= 36; i++) {
        sprite(whichSprite).rotation = i * 10;
        _movie.updateStage();
    }
}
```

See also

obeyScoreRotation, originMode, Sprite

rotation (backdrop and overlay)

Usage

```
sprite(whichSprite).camera.backdrop[backdropIndex].rotation
member(whichCastmember).camera(whichCamera).backdrop
[backdropIndex].rotation
sprite(whichSprite).camera.overlay[overlayIndex].rotation
member(whichCastmember).camera[cameraIndex].overlay
[overlayIndex].rotation
```

Description

3D property; allows you to get or set the rotation of the backdrop or overlay toward the default camera. The default value of this property is 0.0.

Example

This statement rotates a backdrop 60° around its registration point:

```
sprite(4).camera.backdrop[1].rotation = 60.0
```

See also

bevelDepth, transform (property)

rotation (engraver shader)

Usage

```
member(whichCastmember).shader(whichShader).rotation
member(whichCastmember).model(whichModel).shader.rotation
member(whichCastmember).model(whichModel).shaderList[index].rotation
```

Description

3D shader engraver property; allows you to get or set an angle in degrees (as a floating-point number) that describes a 2D rotational offset for engraved lines. The default value for this property is 0.0.

Example

This statement rotates the lines used to draw the engraver shader for the model gbCyl3 by 1°:

```
member("scene").model("gbCyl3").shader.rotation = \
member("scene").model("gbCyl3").shader.rotation + 1
```

See also

transform (property)

rotation (transform)

Usage

```
member(whichCastmember).node(whichNode).transform.rotation
member(whichCastmember).node(whichNode).getWorldTransform().rotation
transform.rotation
```

Description

3D property; allows you to get or set the rotational component of a transform. A transform defines a scale, position and rotation within a given frame of reference. The default value of this property is `vector(0,0,0)`.

A node can be a camera, group, light or model object. Setting the `rotation` of a node's transform defines that object's rotation within the transform's frame of reference. Setting the `rotation` property of an object's world relative transform using `getWorldTransform().rotation` defines the object's rotation relative to the world origin. Setting the `rotation` property of an object's parent relative transform using `transform.rotation` defines the object's rotation relative to its parent node.

If you wish to modify the orientation of a transform it is recommended that you use the `rotate` and `prerotate` methods instead of setting this property.

Example

This statement sets the parent-relative rotation of the first camera in the member to `vector(0,0,0)`:

```
member("Space").camera[1].transform.rotation = vector(0, 0, 0)
```

This example displays the parent-relative rotation of the model named Moon, then it adjusts the model's orientation using the rotate command, and finally it displays the resulting world-relative rotation of the model:

```
put member("SolarSys").model("Moon").transform.rotation
-- vector( 0.0000, 0.0000, 45.0000)
member("SolarSys").model("Moon").rotate(15,15,15)
put member("SolarSys").model("Moon").getWorldTransform().rotation
--vector( 51.3810, 16.5191, 65.8771 )
```

See also

getWorldTransform(), preRotate, rotate, transform (property), position (transform), scale (transform)

rotationReset

Usage

```
member(whichCastmember).model(whichModel).bonesPlayer.rotationReset
member(whichCastmember).model(whichModel).keyframePlayer.rotationReset
```

Description

3D `keyframePlayer` and `bonesPlayer` modifier property; indicates the axes around which rotational changes are maintained from the end of one motion to the beginning of the next, or from the end of one iteration of a looped motion to the beginning of the next iteration.

Possible values of this property include `#none`, `#x`, `#y`, `#z`, `#xy`, `#yz`, `#xz`, and `#all`. The default value is `#all`.

Example

This statement sets the `rotationReset` property of the model named Monster to the *z*-axis. The model maintains rotation around its *z*-axis when the currently playing motion or loop ends.

```
member("NewAlien").model("Monster").bonesPlayer.rotationReset = #z
```

See also

positionReset, bonesPlayer (modifier)

RTF

Usage

```
-- Lingo syntax
memberObjRef.RTF

// JavaScript syntax
memberObjRef.RTF;
```

Description

Cast member property; allows access to the text and tags that control the layout of the text within a text cast member containing text in rich text format.

This property can be tested and set.

Example

This statement displays in the Message window the RTF formatting information embedded in the text cast member Resume:

```
--Lingo syntax
put(member("Resume").RTF)

// JavaScript syntax
trace(member("Resume").RTF);
```

See also

HTML, importFileInto()

safePlayer

Usage

```
-- Lingo syntax
_player.safePlayer

// JavaScript syntax
_player.safePlayer;
```

Description

Player property; controls whether or not safety features in Director are turned on. Read/write.

In a movie with Shockwave content, this property can be tested but not set. It is always TRUE in Shockwave Player.

In the authoring environment and in projectors, the default value is FALSE. This property may be returned, but it may only be set to TRUE. Once it has been set to TRUE, it cannot be set back to FALSE without restarting Director or the projector.

When safePlayer is TRUE, the following safety features are in effect:

- Only safe Xtra extensions may be used.
- The safePlayer property cannot be reset.
- Pasting content from the Clipboard by using the pasteClipBoardInto() method generates a warning dialog box that allows the user to cancel the operation.
- Saving a movie or cast by using script is disabled.
- Printing by using the printFrom() method is disabled.
- Opening an application by using the open() method is disabled.
- The ability to stop an application or the user's computer by using the restart() or shutDown() methods is disabled.
- Opening a file that is outside the DSWMedia folder is disabled.
- Discovering a local filename is disabled.
- Using getNetText() or postNetText(), or otherwise accessing a URL that does not have the same domain as the movie, generates a security dialog box.

Example

The following statement checks for the value set for the safePlayer property in the authoring environment.

```
-- Lingo
put( _player.safePlayer)
--0

// Javascript
trace(_player.safePlayer)
// 0
```

See also

Player

sampleCount

Usage

```
-- Lingo syntax
soundChannelObjRef.sampleCount

// JavaScript syntax
soundChannelObjRef.sampleCount;
```

Description

Sound Channel property; specifies the number of sound samples in the currently playing sound in a sound channel. Read-only.

This is the total number of samples, and depends on the `sampleRate` and `duration` of the sound. It does not depend on the `channelCount` of the sound.

A 1-second, 44.1 KHz sound contains 44,100 samples.

Example

This statement displays the name and `sampleCount` of the cast member currently playing in sound channel 1 in the Message window:

```
-- Lingo syntax
put("Sound cast member" && sound(1).member.name && "contains" && sound(1).sampleCount &&
"samples.")

// JavaScript syntax
put("Sound cast member " + sound(1).member.name + " contains " +sound(1).sampleCount + "
samples.");
```

See also

sampleRate, Sound Channel

sampleRate

Usage

```
-- Lingo syntax
soundChannelObjRef.sampleRate

// JavaScript syntax
soundChannelObjRef.sampleRate;
```

Description

Sound Channel property; returns, in samples per second, the sample rate of the sound cast member or in the case of SWA sound, the original file that has been Shockwave Audio–encoded. Read-only.

This property is available only after the SWA sound begins playing or after the file has been preloaded using the `preLoadBuffer()` method. When a sound channel is given, the result is the sample rate of the currently playing sound cast member in the given sound channel.

Typical values of this property are 8000, 11025, 16000, 22050, and 44100.

When multiple sounds are queued in a sound channel, Director plays them all with the `channelCount`, `sampleRate`, and `sampleSize` of the first sound queued, resampling the rest for smooth playback. Director resets these properties only after the channel's sound queue is exhausted or a `stop()` method is issued. The next sound to be queued or played then determines the new settings.

Example

This statement assigns the original sample rate of the file used in SWA streaming cast member Paul Robeson to the field cast member Sound Quality:

```
-- Lingo syntax
member("Sound Quality").text = string(member("Paul Robeson").sampleRate)

// JavaScript syntax
member("Sound Quality").text = member("Paul Robeson").sampleRate.toString();
```

This statement displays the sample rate of the sound playing in sound channel 1 in the Message window:

```
-- Lingo syntax
trace(sound(1).sampleRate)

// JavaScript syntax
trace(sound(1).sampleRate);
```

See also

`channelCount`, `sampleSize`, `preLoadBuffer()`, `Sound Channel`, `stop() (Sound Channel)`

sampleSize

Usage

```
-- Lingo syntax
memberObjRef.sampleSize

// JavaScript syntax
memberObjRef.sampleSize;
```

Description

Cast member property; determines the sample size of the specified cast member. The result is usually a size of 8 or 16 bits. If a sound channel is given, the value if for the sound member currently playing in the given sound channel.

This property can be tested but not set.

Example

This statement checks the sample size of the sound cast member Voice Over and assigns the value to the variable `soundSize`:

```
-- Lingo syntax
soundSize = member("Voice Over").sampleSize

// JavaScript syntax
var soundSize = member("Voice Over").sampleSize;
```

This statement displays the sample size of the sound playing in sound channel 1 in the Message window:

```
-- Lingo syntax
put(sound(1).sampleSize)

// JavaScript syntax
put(sound(1).sampleSize);
```

savew3d

Usage

```
--Lingo Syntax
member(whichcastmember).savew3d(Absolute path (optional argument))

// JavaScript syntax
member(whichcastmember).savew3d(Absolute path (optional argument));
```

Description

Saves the 3D world from a projector.

The behavior of this method is as follows:

- If the absolute path is provided, this method saves the 3D scene at that instant to the file whose path name is specified (provided the path is correct). This will happen in both Authoring and Projector mode.
- If the absolute path is not provided that is, the call is membername.savew3d(), the behavior is as follows:
  - If the w3d member is externally linked and savew3d is called in projector mode, then the 3D scene at that instant is saved to the externally linked w3d file overwriting the file.
  - If the w3d member is internal and savew3d is called in projector mode, then this method call is ignored.

In authoring mode, savew3d method has the same behavior as the saveWorld method. The member is marked for writing and 3D save happens after you save the Director file.

Example

This statement saves the changes made to the member (“3dworld”) to a separate file named savedworld.w3d.

```
-- Lingo syntax
member(“3dworld”).savew3d(the moviepath & “savedworld.w3d“)

// JavaScript syntax
member(“3dworld”).savew3d(the moviepath & “savedworld.w3d“);
```

saveWorld

Usage

```
--Lingo Syntax
member(whichcastmember).saveWorld()
```

```
// JavaScript syntax
member(whichcastmember).saveWorld();
```

Description

Saves the existing 3D world to the 3D member after you save the movie.

Example

This statement saves the changes made to the member (“3dworld”) to a separate file named savedworld.w3d.

```
-- Lingo syntax
member(“3dworld“).saveWorld()

// JavaScript syntax
member(“3dworld“).saveWorld();
```

scale (3D)

Usage

```
member(whichCastmember).camera(whichCamera).backdrop[backdropIndex].scale
member(whichCastmember).camera(whichCamera).overlay[overlayIndex].scale
```

Description

3D property; allows you to get or set the scale value used by a specific overlay or backdrop in the referenced camera’s list of overlays or backdrops to display. The width and height of the backdrop or overlay are multiplied by the scale value.The default value for this property is 1.0.

Example

This statement doubles the size of a backdrop:

```
-- Lingo
sprite(25).camera.backdrop[1].scale = 2.0

// Javascript
sprite(25).getPropRef(“camera”,1).getProp(“backDrop”,1).scale=2.0;
```

See also

bevelDepth, overlay

scale (backdrop and overlay)

Usage

```
member(whichCastmember).camera(whichCamera).backdrop[backdropIndex].scale
member(whichCastmember).camera(whichCamera).overlay[overlayIndex].scale
```

Description

3D property; allows you to get or set the scale value used by a specific overlay or backdrop in the referenced camera’s list of overlays or backdrops to display. The width and height of the backdrop or overlay are multiplied by the scale value.The default value for this property is 1.0.

Example

This statement doubles the size of a backdrop:

```
sprite(25).camera.backdrop[1].scale = 2.0
```

See also

bevelDepth, overlay

scale (Member)

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.scale

// JavaScript syntax
memberOrSpriteObjRef.scale;
```

Description

Cast member property and sprite property; controls the scaling of a QuickTime, vector shape, or Flash movie sprite.

For QuickTime, this property does not scale the sprite's bounding rectangle or the sprite's controller. Instead, it scales the image around the image's center point within the bounding rectangle. The scaling is specified as a Director list containing two percentages stored as float-point values:

```
[xPercent, yPercent]
```

The *xPercent* parameter specifies the amount of horizontal scaling; the *yPercent* parameter specifies vertical scaling.

When the sprite's `crop` property is set to `TRUE`, the `scale` property can be used to simulate zooming within the sprite's bounding rectangle. When the sprite's `crop` property is set to `FALSE`, the `scale` property is ignored.

This property can be tested and set. The default value is [1.0000,1.0000].

For Flash movie or vector shape cast members, the scale is a floating-point value. The movie is scaled from its origin point, as specified by its `originMode` property.

Note: *This property must be set to the default value if the scaleMode property is set to #autoSize; otherwise the sprite does not display correctly.*

Example

This handler accepts a reference to a Flash movie sprite as a parameter, reduces the movie's scale to 0% (so it disappears), and then scales it up again in 5% increments until it is full size (100%) again:

```
-- Lingo syntax
on scaleMovie whichSprite
    sprite(whichSprite).scale = 0
    _movie.updatestage()
    repeat with i = 1 to 20
        sprite(whichSprite).scale = i * 5
        _movie.updatestage()
    end repeat
end

// JavaScript syntax
function scaleMovie(whichSprite) {
```

```
    sprite(whichSprite).scale = 0;
    _movie.updateStage();
    var i = 1;
    while (i < 21) {
        sprite(whichSprite).scale = i * 5;
        _movie.updateStage();
        i++;
    }
}
```

See also

scaleMode, originMode

scale (transform)

Usage

```
member(whichCastmember).node(whichNode).transform.scale
member(whichCastmember).node(whichNode).getWorldTransform().scale
transform.scale
```

Description

3D property; allows you to get or set the scaling component of a transform. A transform defines a scale, position and rotation within a given frame of reference. The `scale` property allows you to get and set the degree of scaling of the transform along each of the three axes. The default value of this property is `vector(1.0,1.0,1.0)`.

A node can be a camera, group, light or model object. This command does not have any visual effect on lights or cameras as they do not contain geometry. Setting the `scale` property of a node's transform defines that object's scaling along the X, Y and Z axes within the transform's frame of reference. Getting the `scale` property of an object's world relative transform using `getWorldTransform().scale` returns the object's scaling relative to the world origin. Setting the `scale` property of an object's parent relative transform using `transform.scale` defines the object's scaling relative to its parent node.

Example

This statement sets the `scale` property of the transform of the model named Moon to vector(2,5,3):

```
member("Scene").model("Moon").transform.scale = vector(2,5,3)
```

See also

transform (property), getWorldTransform(), position (transform), rotation (transform), scale (command)

scaleMode

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.scaleMode

// JavaScript syntax
memberOrSpriteObjRef.scaleMode;
```

Description

Cast member property and sprite property; controls the way a Flash movie or vector shape is scaled within a sprite's bounding rectangle. When you scale a Flash movie sprite by setting its `scale` and `viewScale` properties, the sprite itself is not scaled; only the view of the movie within the sprite is scaled. The `scaleMode` property can have these values:

- `#showAll` (default for Director movies prior to version 7)—Maintains the aspect ratio of the original Flash movie cast member. If necessary, fill in any gap in the horizontal or vertical dimension using the background color.
- `#noBorder`—Maintains the aspect ratio of the original Flash movie cast member. If necessary, crop the horizontal or vertical dimension.
- `#exactFit`—Does not maintain the aspect ratio of the original Flash movie cast member. Stretch the Flash movie to fit the exact dimensions of the sprite.
- `#noScale`—preserves the original size of the Flash media, regardless of how the sprite is sized on the Stage. If the sprite is made smaller than the original Flash movie, the movie displayed in the sprite is cropped to fit the bounds of the sprite.
- `#autoSize` (default)—This specifies that the sprite rectangle is automatically sized and positioned to account for `rotation`, `skew`, `flipH`, and `flipV`. This means that when a Flash sprite is rotated, it will not crop as in earlier versions of Director. The `#autoSize` setting only functions properly when `scale`, `viewScale`, `originPoint`, and `viewPoint` are at their default values.

This property can be tested and set.

Example

The following sprite script checks the Stage color of the Director movie and, if the Stage color is indexed to position 0 in the current palette, the script sets the `scaleMode` property of a Flash movie sprite to `#showAll`. Otherwise, it sets the `scaleMode property` to `#noBorder`.

```
-- Lingo syntax
property spriteNum

on beginsprite me
    if _movie.stage.bgColor = 0 then
        sprite(spriteNum).scaleMode = #showAll
    else
        sprite(spriteNum).scaleMode = #noBorder
    end if
end

// JavaScript syntax
function beginsprite() {
    var stgClr = _movie.stage.bgColor;
    if (stgClr == 0) {
        sprite(this.spriteNum).scaleMode = symbol("showAll");
    } else {
        sprite(this.spriteNum).scaleMode = symbol("noBorder");
    }
}
```

See also

`scale (Member)`

score

Usage

```
-- Lingo syntax
_movie.score

// JavaScript syntax
_movie.score;
```

Description

Movie property; determines which Score is associated with the current movie. Read/write.

This property can be useful for storing the current contents of the Score before wiping out and generating a new one or for assigning the current Score contents to a film loop.

Example

This statement assigns the film loop cast member Waterfall to the Score of the current movie:

```
-- Lingo syntax
_movie.score = member("Waterfall").media

// JavaScript syntax
_movie.score = member("Waterfall").media;
```

See also

Movie

scoreColor

Usage

```
sprite(whichSprite).scoreColor
the scoreColor of sprite whichSprite
```

Description

Sprite property; indicates the Score color assigned to the sprite specified by *whichSprite*. The possible values correspond to color chips 0 to 5 in the current palette.

This property can be tested and set. Setting this property is useful only during authoring and Score recording.

Example

This statement displays in the Message window the value for the Score color assigned to sprite 7:

```
put sprite(7).scorecolor
```

scoreSelection

Usage

```
-- Lingo syntax
_movie.scoreSelection

// JavaScript syntax
```

```
_movie.scoreSelection;
```

Description

Movie property; determines which channels are selected in the Score window. Read/write.

The information is formatted as a linear list of linear lists. Each contiguous selection is in a list format consisting of the starting channel number, ending channel number, starting frame number, and ending frame number. Specify sprite channels by their channel numbers; use the following numbers to specify the other channels.

To specify:	Use:
Frame script channel	0
Sound channel 1	-1
Sound channel 2	-2
Transition channel	-3
Palette channel	-4
Tempo channel	-5

You can select discontinuous channels or frames.

Example

This statement selects sprite channels 15 through 25 in frames 100 through 200:

```
-- Lingo syntax
_movie.scoreSelection = [[15, 25, 100, 200]]

// JavaScript syntax
_movie.scoreSelection = list(list(15, 25, 100, 200));
```

This statement selects sprite channels 15 through 25 and 40 through 50 in frames 100 through 200:

```
-- Lingo syntax
_movie.scoreSelection = [[15, 25, 100, 200], [40, 50, 100, 200]]

// JavaScript syntax
_movie.scoreSelection = list(list(15, 25, 100, 200), list(40, 50, 100, 200));
```

This statement selects the frame script in frames 100 through 200:

```
-- Lingo syntax
_movie.scoreSelection = [[0, 0, 100, 200]]

// JavaScript syntax
_movie.scoreSelection = list(list(0, 0, 100, 200));
```

See also

Movie

script

Usage

```
-- Lingo syntax
_movie.script[scriptNameOrNum]
```

```
// JavaScript syntax
_movie.script[scriptNameOrNum];
```

Description

Movie property; provides indexed or named access to the script cast members of a movie. Read-only.

The *scriptNameOrNum* argument can be either a string that specifies the name of the script cast member or an integer that specifies the number of the script cast member.

- If *scriptNameOrNum* is a string, the `script` property provides access to the script cast member, regardless of which cast library contains that member.
- If *scriptNameOrNum* is an integer, the `script` property provides access only to the script cast member found within the first cast library of the referenced movie; you cannot use indexed access to specify a cast library other than the first one.

Example

The following statement accesses a named script.

```
-- Lingo syntax
bugScript = _movie.script["Warrior Ant"]

// JavaScript syntax
var bugScript = _movie.script["Warrior Ant"];
```

See also

Movie

scripted

Usage

```
-- Lingo syntax
spriteChannelObjRef.scripted

// JavaScript syntax
spriteChannelObjRef.scripted;
```

Description

Sprite Channel property; specifies whether a sprite channel is under script control (TRUE) or under Score control (FALSE). Read-only.

Example

These statements create a scripted sprite from cast member kite in sprite channel 5 if that channel is not already under script control.

```
-- Lingo syntax
if (channel(5).scripted = FALSE) then
    channel(5).makeScriptedSprite(member("kite"))
end if

// JavaScript syntax
if (channel(5).scripted == false) {
    channel(5).makeScriptedSprite(member("kite"));
}
```

See also

Sprite Channel

scriptingXtraList

Usage

```
-- Lingo syntax
_player.scriptingXtraList

// JavaScript syntax
_player.scriptingXtraList;
```

Description

Player property; returns a linear list of all scripting Xtra extensions available to the Director player. Read-only.

The Xtra extensions are those that are present in the Configuration\Xtras folder.

Example

This statement displays in the Message window all available scripting Xtra extensions:

```
-- Lingo syntax
trace(_player.scriptingXtraList)

// JavaScript syntax
trace(_player.scriptingXtraList);
```

See also

mediaXtraList, Player, Scripting Objects, toolXtraList, transitionXtraList, xtraList (Player)

scriptInstanceList

Usage

```
sprite(whichSprite).scriptInstanceList
the scriptInstanceList of sprite whichSprite
```

Description

Sprite property; creates a list of script references attached to a sprite. This property is available only during run time. The list is empty when the movie is not running. Modifications to the list are not saved in the Score. This property is useful for the following tasks:

- Attaching a behavior to a sprite for use during run time
- Determining if behaviors are attached to a sprite; determining what the behaviors are
- Finding a behavior script reference to use with the `sendSprite` command

This property can be tested and set. (It can be set only if the sprite already exists and has at least one instance of a behavior already attached to it.)

Example

This handler displays the list of script references attached to a sprite:

```
-- Lingo
on showScriptRefs spriteNum
    put sprite(spriteNum).scriptInstanceList
end

// Javascript
function showScriptRefs(spriteNum)
{
    trace(sprite(spriteNum).scriptInstanceList);
}
```

These statements attach the script Big Noise to sprite 5:

```
x = script("Big Noise").new()
sprite(5).scriptInstanceList.add(x)
```

See also

scriptNum, sendSprite()

scriptList

Usage

```
sprite(whichSprite).scriptList
the scriptList of sprite whichSprite
```

Description

Sprite property; returns the list of behaviors attached to the given sprite and their properties. This property may only be set by using `setScriptList()`. It may not be set during a score recording session.

Example

This statement displays the list of scripts attached to sprite 1 in the Message window:

```
put sprite(1).scriptList
-- [[(member 2 of castLib 1), "[#myRotateAngle: 10.0000, #myClockwise: 1, #myInitialAngle:
0.0000]"], [(member 3 of castLib 1), "[#myAnglePerFrame: 10.0000, #myTurnFrames: 10,
#myHShiftPerFrame: 10, #myShiftFrames: 10, #myTotalFrames: 60, #mySurfaceHeight: 0]"]]
```

See also

setScriptList(), value()

scriptNum

Usage

```
sprite(whichSprite). scriptNum
scriptNum of sprite whichSprite
```

Description

Sprite property; indicates the number of the script attached to the sprite specified by *whichSprite*. If the sprite has multiple scripts attached, `scriptNum` sprite property returns the number of the first script. (To see a complete list of the scripts attached to a sprite, see the behaviors listed for that sprite in the Behavior Inspector.)

This property can be tested and set during Score recording.

Example

This statement displays the number of the script attached to sprite 4:

```
-- Lingo
put sprite(4).scriptNum

// Javascript
trace(sprite(4).scriptNum);
```

See also

scriptInstanceList

scriptsEnabled

Usage

```
-- Lingo syntax
memberObjRef.scriptsEnabled

// JavaScript syntax
memberObjRef.scriptsEnabled;
```

Description

Director movie cast member property; determines whether scripts in a linked movie are enabled (TRUE or 1) or disabled (FALSE or 0).

This property is available for linked Director movie cast members only.

This property can be tested and set.

Example

This statement turns off scripts in the linked movie Jazz Chronicle:

```
-- Lingo syntax
member("Jazz Chronicle").scriptsEnabled = FALSE

// JavaScript syntax
member("Jazz Chronicle").scriptsEnabled = 0;
```

scriptText

Usage

```
-- Lingo syntax
memberObjRef.scriptText

// JavaScript syntax
memberObjRef.scriptText;
```

Description

Member property; indicates the content of the script, if any, assigned to a cast member.
Read/write.

The text of a script is removed when a movie is converted to a projector, protected, or compressed for Shockwave Player. Such movies then lose their values for the `scriptText` property. Therefore, the movie's `scriptText` property values cannot be retrieved when the movie is played back by a projector. However, Director can set new values for the `scriptTex` property inside the projector. These newly assigned scripts are automatically compiled so that they execute quickly.

Example

This statement makes the contents of field cast member 20 the script of cast member 30:

```
-- Lingo syntax
member(20).text = member(30).scriptText

// JavaScript syntax
member(20).text = member(30).scriptText;
```

See also

Member

scriptType

Usage

```
member whichScript.scriptType
the scriptType of member whichScript
```

Description

Cast member property; indicates the specified script's type. Possible values are `#movie`, `#score`, and `#parent`.

Example

This statement makes the script member Main Script a movie script:

```
-- Lingo
member("Main Script").scriptType = #movie

// Javascript
member("Main Script").scriptType = Symbol("movie");
```

scrollTop

Usage

```
-- Lingo syntax
memberObjRef.scrollTop

// JavaScript syntax
memberObjRef.scrollTop;
```

Description

Cast member property; determines the distance, in pixels, from the top of a field cast member to the top of the field that is currently visible in the scrolling box. By changing the value for `scrollTop` member property while the movie plays, you can change the section of the field that appears in the scrolling field.

This is a way to make custom scrolling behaviors for text and field members.

For example, the following Lingo moves the field cast member Credits up or down within a field's box, depending on the value in the variable `sliderVal`:

```
global sliderVal

on prepareFrame
    newVal = sliderVal * 100
    member("Credits").scrolltop = newVal
end
```

The global variable `sliderVal` could measure how far the user drags a slider. The statement `set newVal = sliderVal * 100` multiplies `sliderVal` to give a value that is greater than the distance the user drags the slider. If `sliderVal` is positive, the text moves up; if `sliderVal` is negative, the text moves down.

Example

This repeat loop makes the field Credits scroll by continuously increasing the value of `scrollTop`:

```
--Lingo syntax
on wa
    member("Credits").scrollTop = 1
    repeat with count = 1 to 150
        member("Credits").scrollTop = member("Credits").scrollTop + 1
        _movie.updateStage()
    end repeat
end

// JavaScript syntax
function wa() {
    member("Credits").scrollTop = 1;
    for (var count = 1; count <= 150; count++) {
        member("Credits").scrollTop = member("Credits").scrollTop + 1;
        _movie.updateStage();
    }
}
```

sds (modifier)

Usage

```
member(whichCastmember).model(whichModel).sds.whichProperty
```

Description

3D modifier; adds geometric detail to models and synthesizes additional details to smooth out curves as the model moves closer to the camera. After you have added the `sds` modifier to a model using `addModifier()`, you can set the properties of the `sds` modifier.

The `sds` modifier directly affects the model resource. Be careful when using the `sds` and `lod` modifiers together, because they perform opposite functions (the `sds` modifier adds geometric detail and the `lod` modifier removes geometric detail). Before adding the `sds` modifier, it is recommended that you set the `lod.auto` modifier property to `FALSE` and set the `lod.level` modifier property to the desired resolution, as follows:

```
member("myMember").model("myModel").lod.auto = 0
member("myMember").model("myModel").lod.level = 100
member("myMember").model("myModel").addmodifier(#sds)
```

The sds modifier cannot be used with models that already use either the inker or toon modifiers.

After you have added the sds modifier to a model resource you can get or set the following properties:

enabled indicates whether subdivision surfaces functionality is enabled (TRUE) or disabled (FALSE). The default setting for this property is TRUE.

depth specifies the maximum number of levels of resolution that the model can display when using the sds modifier.

error indicates the level of error tolerance for the subdivision surfaces functionality. This property applies only when the sds.subdivision property is set to #adaptive.

subdivision indicates the mode of operation of the subdivision surfaces modifier. Possible values are as follows:

- #uniform specifies that the mesh is uniformly scaled up in detail, with each face subdivided the same number of times.
- #adaptive specifies that additional detail is added only when there are major face orientation changes and only to those areas of the mesh that are currently visible.

Note: *For more detailed information about these properties, see the individual property entries.*

Example

The statement displays the sds.depth property value for the model named Terrain:

```
-- Lingo
put member("3D").model("Terrain").sds.depth
-- 2

// Javascript
trace(member("3D").getPropRef("model",1).sds.depth
// 2
```

See also

lod (modifier), toon (modifier), inker (modifier), depth (3D), enabled (sds), error, subdivision, addModifier

searchCurrentFolder

Usage

```
-- Lingo syntax
_player.searchCurrentFolder

// JavaScript syntax
_player.searchCurrentFolder;
```

Description

Player property; determines whether Director searches the current folder when searching filenames. Read/write.

- When the searchCurrentFolder property is TRUE (1), Director searches the current folder when resolving filenames.
- When the searchCurrentFolder property is FALSE (0), Director does not search the current folder when resolving filenames.

This property is TRUE by default.

Note: *_player.searchCurrentFolder is available only in Author and Projector runModes and is not available in the Plugin runMode.*

Example

This statement displays the status of the `searchCurrentFolder` property in the Message window. The result is 1, which is the numeric equivalent of `TRUE`:

```
-- Lingo syntax
put(_player.searchCurrentFolder)

// JavaScript syntax
put(_player.searchCurrentFolder);
```

See also

`Player`

searchPathList

Usage

```
-- Lingo syntax
_player.searchPathList

// JavaScript syntax
_player.searchPathList;
```

Description

Player property; a list of paths that Director searches when trying to find linked media such as digital video, GIFs, bitmaps, or sound files. Read/write.

Each item in the list of paths is a fully qualified pathname as it appears on the current platform at runtime.

The value of `searchPathList` is a linear list that you can manipulate the same as any other list by using commands such as `add()`, `addAt()`, `append()`, `deleteAt()`, and `setAt()`. The default value is an empty list.

URLs should not be used as file references in the search paths.

Adding a large number of paths to `searchPaths` slows searching. Try to minimize the number of paths in the list.

Note: *This property will function on all subsequent movies after being set. Because the current movie's assets have already been loaded, changing the setting will not affect any of these assets.*

Example

This statement displays the paths that Director searches when resolving filenames:

```
-- Lingo syntax
trace(_player.searchPathList)

// JavaScript syntax
trace(_player.searchPathList);
```

The following statement assigns two folders to `searchPaths` in Windows:

```
-- Lingo syntax
_player.searchPathList = ["C:\Director\Projects\", "D:\CDROM\Sources\"]

// JavaScript syntax
```

```
_player.searchPathList = list("C:\\Director\\Projects\\","D:\\CDROM\\Sources\\");
```

The following statement assigns two folders to searchPaths on a Mac:

```
-- Lingo syntax
_player.searchPathList = ["Hard Drive:Director:Projects:", "CDROM:Sources:"]

// JavaScript syntax
_player.searchPathList = list("Hard Drive:Director:Projects:","CDROM:Sources:");
```

See also

Player, searchCurrentFolder

selectedButton

Usage

```
-- Lingo syntax
dvdObjRef.selectedButton

// JavaScript syntax
dvdObjRef.selectedButton;
```

Description

DVD property; returns the index of the button that currently has focus. Read-only.

Example

The statement displays the button which is selected in the DVD object.

```
-- Lingo
put sprite(1).selectedButton
--1

// Javascript
trace(sprite(1).selectedButton)
//1
```

See also

DVD

selectedText

Usage

```
-- Lingo syntax
memberObjRef.selectedText

// JavaScript syntax
memberObjRef.selectedText;
```

Description

Text cast member property; returns the currently selected chunk of text as a single object reference. This allows access to font characteristics as well as to the string information of the actual characters.

Example

The following handler displays the currently selected text being placed in a local variable `object`. Then that object is used to reference various characteristics of the text, which are detailed in the Message window.

```
--Lingo syntax
property spriteNum

on mouseUp(me)
    mySelectionObject = sprite(spriteNum).member.selectedText
    put(mySelectionObject.text)
    put(mySelectionObject.font)
    put(mySelectionObject.fontSize)
    put(mySelectionObject.fontStyle)
end

// JavaScript syntax
function mouseUp() {
    var mySelectionObject = sprite(this.spriteNum).member.selectedText;
    trace(mySelectionObject.text);
    trace(mySelectionObject.font);
    trace(mySelectionObject.fontSize);
    trace(mySelectionObject.fontStyle);
}
```

selection

Usage

```
-- Lingo syntax
castObjRef.selection

// JavaScript syntax
castObjRef.selection;
```

Description

Cast library property; returns the cast members that are selected in a given Cast window. Read/write.

Example

This statement selects cast members 1 through 10 in castLib 1:

```
-- Lingo syntax
castLib(1).selection = [[1, 10]]

// JavaScript syntax
castLib(1).selection = list( list(1, 10) );
```

This statement selects cast members 1 through 10, and 30 through 40, in castLib 1:

```
-- Lingo syntax
castLib(1).selection = [[1, 10], [30, 40]]

// JavaScript syntax
castLib(1).selection = list( list(1, 10), list(30, 40) );
```

See also

Cast Library

selection (text cast member property)

Usage

```
member(whichTextMember).selection
```

Description

Text cast member property; returns a list of the first and last character selected in the text cast member.

This property can be tested and set.

Example

The following statement sets the selection displayed by the sprite of text member myAnswer so that characters 6 through 10 are highlighted:

```
member("myAnswer").selection = [6, 10]
```

See also

color(), selStart, selEnd

selEnd

Usage

```
-- Lingo syntax
_movie.selEnd

// JavaScript syntax
_movie.selEnd;
```

Description

This is a movie property that specifies the last character of a selection. It is used with selStart to identify a selection in the current editable field, beginning with the first character. The maxiumum value for selEnd is the number of characters in the currently editable field. This property applies to field member sprites, but not to text member sprites. If you assign a higher selStart value than selEnd, the value of selEnd is reset to that for selStart.

This property can be tested and set. The default value is 0.

Example

These statements select "cde" from the field "abcdefg":

```
-- Lingo syntax
_movie.selStart = 2
_movie.selEnd = 5

// JavaScript syntax
_movie.selStart = 2;
_movie.selEnd = 5;
```

This statement makes a selection 20 characters long:

```
-- Lingo syntax
_movie.selEnd = _movie.selStart + 20

// JavaScript syntax
```

```
_movie.selEnd = _movie.selStart + 20;
```

See also

editable, hilite (command), selection() (function), selStart, text

selStart

Usage

```
-- Lingo syntax
_movie.selStart

// JavaScript syntax
_movie.selStart;
```

Description

This is a movie property specifying the preceding position of a starting character in a selection. It is used with selEnd to identify a selection in the current editable field. A value of 0 indicates a position before the first character. The maximum value for selStart is the number of characters in the currently editable field. This property applies to field member sprites, but not to text member sprites. If you assign a higher selStart value than selEnd, the value of selEnd is reset to that for selStart.

This property can be tested and set. The default value is 0.

Example

These statements select "cde" from the field "abcdefg":

```
-- Lingo syntax
_movie.selStart = 2
_movie.selEnd = 5

// JavaScript syntax
_movie.selStart = 2;
_movie.selEnd = 5;
```

This statement makes a selection 20 characters long:

```
-- Lingo syntax
_movie.selEnd = _movie.selStart + 20

// JavaScript syntax
_movie.selEnd = _movie.selStart + 20;
```

See also

selection() (function), selEnd, text

serialNumber

Usage

```
-- Lingo syntax
_player.serialNumber

// JavaScript syntax
_player.serialNumber;
```

Description

Movie property; a string containing the serial number entered when Director was installed. Read-only.

This property is available in the authoring environment only. It could be used in a movie in a window (MIAW) tool that is personalized to show the user's information.

Example

This handler would be located in a movie script of a MIAW. It places the user's name and the serial number into a display field when the window is opened:

```
-- Lingo syntax
on prepareMovie
displayString = _player.userName & RETURN & _player.organizationName & RETURN &
_player.serialNumber
member("User Info").text = displayString
end

// JavaScript syntax
function prepareMovie() {
var displayString = _player.userName + "\n" + _player.organizationName + "\n" +
_player.serialNumber;
member("User Info").text = displayString;
}
```

See also

Player

shader

Usage

```
member(whichCastmember).shader(whichShader)
member(whichCastmember).shader[index]
member(whichCastmember).model(whichModel).shader
member(whichCastmember).modelResource(whichModelResource).face[index].shader
```

Description

3D element, model property, and face property; the object used to define the appearance of the surface of the model. The shader is the "skin" which is wrapped around the model resource used by the model.

The shader itself is not an image. The visible component of a shader is created with up to eight layers of `texture`. These eight texture layers are either created from bitmap cast members or image objects within Director or imported with models from 3D modeling programs. For more information, see `texture`.

Every model has a linear list of shaders called the `shaderlist`. The number of entries in this list equals the number of meshes in the model resource used by the model. Each mesh can be shaded by only one shader.

The 3D cast member has a default shader named `DefaultShader`, which cannot be deleted. This shader is used when no shader has been assigned to a model and when a shader being used by a model is deleted.

The syntax `member(`*`whichCastmember`*`).model(`*`whichModel`*`).shader` gives access to the first shader in the model's `shaderlist` and is equivalent to `member(`*`whichCastmember`*`).model(`*`whichModel`*`).shaderList[1]`.

Create and delete shaders with the `newShader()` and `deleteShader()` commands.

Shaders are stored in the shader palette of the 3D cast member. They can be referenced by name (*whichShader*) or palette index (*shaderIndex*). A shader can be used by any number of models. Changes to a shader will appear in all models which use that shader.

There are four types of shaders:

`#standard` shaders present their textures realistically.

`#painter`, `#engraver`, and `#newsprint` shaders stylize their textures for painting, engraving, and newsprint effects. They have special properties in addition to the `#standard` shader properties.

For more information about shader properties, see the Using Director topics in the Director Help Panel.

The shaders used by individual faces of `#mesh` primitives can be set with the syntax `member(whichCastmember).modelResource(whichModelResource).face[index].shader`. Changes to this property require a call to the `build()` command.

Example

This statement sets the `shader` property of the model named Wall to the shader named WallSurface:

```
member("Room").model("Wall").shader = member("Room").shader("WallSurface")
```

See also

shaderList, newShader, deleteShader, face, texture

shaderList

Usage

```
member(whichCastmember).model(whichModel).shaderList
member(whichCastmember).model(whichModel).shaderList[index]
```

Description

3D model property; a linear list of `shadowPercentage` applied to the model. The number of entries in this list equals the number of meshes in the model resource used by the model. Each mesh can be shaded by only one shader.

Set the shader at the specified *index* position in the `shaderlist` with this syntax:
`member(whichCastmember).model(whichModel).shaderList[index] = shaderReference`

With 3D text, each character is a separate mesh. Set the value of `index` to the number of the character whose shader you want to set.

Set all *index* positions in the `shaderList` to the same shader with this syntax (note the absence of an index for the `shaderList`): `member(whichCastmember).model(whichModel).shaderList = shaderReference`

Set a property of a shader in the `shaderlist` with this syntax:

```
member(whichCastmember).model(whichModel).shaderList[index].\
whichProperty = propValue
```

Set a property of all of the shaders of a model to the same value with this syntax (note the absence of an index for the shaderList):

```
member(whichCastmember).model(whichModel).shaderList.whichProperty = propValue
```

Example

This statement sets the second shader in the `shaderList` of the model named Bumper to the shader named Chrome:

```
member("Car").model("Bumper").shaderList[2] = member("Car").shader("Chrome")
```

This statement sets the all the shaders in the `shaderList` of the model named Bumper to the shader named Chrome:

```
member("Car").model("Bumper").shaderList = member("Car").shader("Chrome")
```

See also

shadowPercentage

shadowPercentage

Usage

```
member(whichCastmember).model(whichModel).toon.shadowPercentage
member(whichCastmember).model(whichModel).shader.shadowPercentage
member(whichCastmember).shader(whichShader).shadowPercentage
```

Description

3D `toon` modifier and `painter` shader property; indicates the percentage of available colors that are used in the area of the model's surface where light does not create highlights.

The range of this property is 0 to 100, and the default value is 50.

The number of colors used by the `toon` modifier and `painter` shader for a model is determined by the colorSteps property of the model's `toon` modifier or `painter` shader.

Example

The following statement sets the `shadowPercentage` property of the `toon` modifier for the model named Teapot to 50. Half of the colors available to the `toon` modifier for this model will be used for the shadow area of the model's surface.

```
-- Lingo
member("shapes").model("Teapot").toon.shadowPercentage = 50

// Javascript
member("shapes").getPropRef("model",1).getProp("toon").shadowPercentage=50;
```

See also

colorSteps, shadowStrength

shadowStrength

Usage

```
member(whichCastmember).model(whichModel).toon.shadowStrength
member(whichCastmember).model(whichModel).shader.shadowStrength
member(whichCastmember).shader(whichShader).shadowStrength
```

Description

3D `toon` modifier and `#painter` shader property; indicates the brightness of the area of the model's surface where light does not create highlights.

The default value of this property is 1.0.

Example

The following statement sets the `shadowStrength` property of the toon modifier for the model named Sphere to 0.1. The area of the model's surface that is not highlighted will be very dark.

```
member("Shapes").model("Sphere").toon.shadowStrength = 0.1
```

shapeType

Usage

```
member(whichCastMember).shapeType
the shapeType of member whichCastMember
```

Description

Shape cast member property; indicates the specified shape's type. Possible types are `#rect`, `#roundRect`, `#oval`, and `#line`. You can use this property to specify a shape cast member's type after creating the shape cast member using Lingo.

Example

These statements create a new shape cast member numbered 100 and then define it as an oval:

```
-- Lingo
new(#shape, member 100)
member(100).shapeType = #oval

// Javascript
var t = _movie.newMember(symbol("shape"));
t.shapeType=symbol("oval");
```

shiftDown

Syntax

```
-- Lingo syntax
_key.shiftDown

// JavaScript syntax
_key.shiftDown;
```

Description

Key property; indicates whether the user is pressing the Shift key. Read-only.

This property returns `TRUE` if the user is pressing the Shift key; otherwise, it returns `FALSE`.

This property must be tested in conjunction with another key.

Example

This statement checks whether the Shift key is being pressed and calls the handler `doCapitalA` if it is:

```
-- Lingo syntax
if (_key.shiftDown) then
    doCapitalA(_key.key)
end if
```

```
// JavaScript syntax
if (_key.shiftDown) {
    doCapitalA(_key.key);
}
```

See also

controlDown, Key, key, keyCode, optionDown

shininess

Usage

```
member(whichCastmember).shader(whichShader).shininess
member(whichCastmember).model(whichModel).shader.shininess
member(whichCastmember).model(whichModel).shaderList[shaderListIndex].shininess
```

Description

3D standard shader property; allows you to get or set the shininess of a surface. Shininess is defined as the percentage of shader surface devoted to highlights. The value is an integer between 0 and 100, with a default of 30.

All shaders have access to the `#standard` shader properties; in addition to these standard shader properties shaders of the types `#engraver`, `#newsprint`, and `#painter` have properties unique to their type. For more information, see newShader.

Example

The following statement sets the `shininess` property of the first shader in the shader list of the model gbCyl3 to 60. Sixty percent of the surface of the shader will be dedicated to highlights.

```
-- Lingo
member("Scene").model("gbCyl3").shader.shininess = 60

// Javascript
member("Scene").getPropRef("model",1).getProp("shader").shininess=60;
```

silhouettes

Usage

```
member(whichCastmember).model(whichModel).inker.silhouettes
member(whichCastmember).model(whichModel).toon.silhouettes
```

Description

3D `toon` and `inker` modifier property; indicates the presence `(TRUE)` or absence `(FALSE)` of lines drawn by the modifier at the visible edges of the model.

Silhouette lines are drawn around the model's 2D image on the camera's projection plane. Their relationship to the model's mesh is not fixed, unlike crease or boundary lines, which are drawn on features of the mesh.

Silhouette lines are similar to the lines that outline images in a child's coloring book.

The default value for this property is `TRUE`.

Example

The following statement sets the `silhouettes` property of the `inker` modifier for the model named Sphere to `FALSE`. Lines will not be drawn around the profile of the model.

```
-- Lingo
member("Shapes").model("Sphere").inker.silhouettes = FALSE

// Javascript
member("Shapes").getPropRef("model",1).inker.silhouetess = false;
```

size

Usage

```
-- Lingo syntax
memberObjRef.size

// JavaScript syntax
memberObjRef.size;
```

Description

Member property; returns the size in memory, in bytes, of a specific cast member. Read-only.

Divide bytes by 1024 to convert to kilobytes.

Example

This line outputs the size of the cast member Shrine in a field named How Big:

```
-- Lingo syntax
member("How Big").text = string(member("shrine").size)

// JavaScript syntax
member("How Big").text = member("shrine").size.toString();
```

See also

`Member`

sizeRange

Usage

```
member(whichCastmember).modelResource
(whichModelResource).sizeRange.start
modelResourceObjectReference.sizeRange.start
member(whichCastmember).modelResource
(whichModelResource).sizeRange.end
modelResourceObjectReference.sizeRange.end
```

Description

3D property; when used with a model resource whose type is `#particle`, this property allows you to get or set the `start` and `end` property of the model resource's `sizeRange`. Particles are measured in world units.

The size of particles in the system is interpolated linearly between `sizeRange.start` and `sizeRange.end` over the lifetime of each particle.

This property must be an integer greater than 0, and has a default value of 1.

Example

In this example, mrFount is a model resource of the type `#particle`. This statement sets the `sizeRange` properties of mrFount. The first line sets the start value to 4, and the second line sets the end value to 1. The effect of this statement is that the particles of mrFount are size 4 when they first appear, and gradually shrink to a size of 1 during their lifetime.

```
-- Lingo
member("fountain").modelResource("mrFount").sizeRange.start = 4
member("fountain").modelResource("mrFount").sizeRange.end = 1

// Javascript
member("fountain").getPropRef("modelResource",1).getProp("sizeRange").start=4;
member("fountain").getPropRef("modelResource",1).getProp("sizeRange").end=4;
```

sizeState

Usage

```
-- Lingo syntax
windowObjRef.sizeState

// JavaScript syntax
windowObjRef.sizeState;
```

Description

Window property; returns the size state of a window. Read-only.

The returned size state will be one of the following values:

Size state	Description
#minimized	Specifies that the window is currently minimized.
#maximized	Specifies that the window is currently maximized.
#normal	Specifies that the window is currently neither minimized nor maximized.

Example

These statements maximize the window named Artists if it is not already maximized.

```
-- Lingo syntax
if (window("Artists").sizeState <> #maximized) then
    window("Artists").maximize()
end if

// JavaScript syntax
if (window("Artists").sizeState != symbol("maximized")) {
    window("Artists").maximize();
}
```

See also

Window

skew

Usage

```
-- Lingo syntax
spriteObjRef.skew

// JavaScript syntax
spriteObjRef.skew;
```

Description

Sprite property; returns, as a float value in hundredths of a degree, the angle to which the vertical edges of the sprite are tilted (skewed) from the vertical. Read/write.

Negative values indicate a skew to the left; positive values indicate a skew to the right. Values greater than 90 flip an image vertically.

The Score can retain information for skewing an image from +21,474,836.47 to -21,474,836.48 , allowing 59,652 full rotations in either direction.

When the skew limit is reached (slightly past the 59,652th rotation), the skew resets to +116.47 or -116.48 —not 0.00 . This is because +21,474,836.47 is equal to +116.47 , and -21,474,836.48 is equal to -116.48 (or +243.12). To avoid this reset condition, constrain angles to ±360 in either direction when using script to perform continuous skewing.

Example

The following behavior causes a sprite to skew continuously by 2 degrees every time the playhead advances, while limiting the angle to 360 degrees.

```
-- Lingo syntax
property spriteNum

on prepareFrame me
    sprite(spriteNum).skew = integer(sprite(spriteNum).skew + 2) mod 360
end

// JavaScript syntax
function prepareFrame() {
    sprite(this.spriteNum).skew = parseInt(sprite(this.spriteNum).skew + 2) % 360;
}
```

See also

flipH, flipV, rotation, Sprite

smoothness

Usage

```
member(whichTextmember).smoothness
member(whichCastMember).modelResource(whichExtruderModelResource).smoothness
```

Description

3D extruder model resource and text property; allows you to get or set an integer controlling the number of segments used to create a 3D text cast member. The higher the number, the smoother the text appears. The range of this property is 1 to 10, and the default value is 5.

For more information about working with extruder model resources and text cast members, see `extrude3D`.

Example

In this example, the cast member Logo is a text cast member. This statement sets the smoothness of Logo to 8. When Logo is displayed in 3D mode, the edges of its letters will be very smooth.

```
-- Lingo
member("Logo").smoothness = 8

// Javascript
member("Logo").smoothness = 8;
```

In this example, the model resource of the model Slogan is extruded text. This statement sets the smoothness of Slogan's model resource to 1, causing the Slogan's letters to appear very angular.

```
-- Lingo
member("Scene").model("Slogan").resource.smoothness = 1

// Javascript
member("Scene").getPropRef("model",1).getProp("resource").smoothness=1;
```

See also

`extrude3D`

sound (Member)

Usage

```
-- Lingo syntax
memberObjRef.sound

// JavaScript syntax
memberObjRef.sound;
```

Description

Cast member property; controls whether a movie, digital video, or Flash movie's sound is enabled (TRUE, default) or disabled (FALSE). Read/write.

In Flash members, the new setting takes effect after the currently playing sound ends.

To see an example of `sound` used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This handler accepts a member reference and toggles the member's `sound` property on or off:

```
-- Lingo syntax
on ToggleSound(whichMember)
    member(whichMember).sound = not(member(whichMember).sound)
end

// JavaScript syntax
function ToggleSound(whichMember) {
    member(whichMember).sound = !(member(whichMember).sound);
}
```

See also

Flash Movie

sound (Player)

Usage

```
-- Lingo syntax
_player.sound[intSoundChannelNum]

// JavaScript syntax
_player.sound[intSoundChannelNum];
```

Description

Player property; provides indexed access to a Sound Channel object by using a Player property. Read-only.

The *intSoundChannelNum* argument is an integer that specifies the number of the sound channel to access.

The functionality of this property is identical to the top level `sound()` method.

Example

This statement sets the variable `mySound` to the sound in sound channel 3:

```
-- Lingo syntax
mySound = _player.sound[3]

// JavaScript syntax
var mySound = _player.sound[3];
```

See also

Player, sound(), Sound Channel

soundChannel (SWA)

Usage

```
-- Lingo syntax
memberObjRef.soundChannel

// JavaScript syntax
memberObjRef.soundChannel;
```

Description

Shockwave Audio (SWA) cast member property; specifies the sound channel in which the SWA sound plays.

If no channel number or channel 0 is specified, the SWA streaming cast member assigns the sound to the highest numbered sound channel that is unused.

Shockwave Audio streaming sounds can appear as sprites in sprite channels, but they play sound in a sound channel. Refer to SWA sound sprites by their sprite channel number, not their sound channel number.

This property can be tested and set.

Example

This statement tells the SWA streaming cast member Frank Zappa to play in sound channel 3:

```
-- Lingo syntax
member("Frank Zappa").soundChannel = 3

// JavaScript syntax
member("Frank Zappa").soundChannel = 3;
```

soundChannel (RealMedia)

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.soundChannel

// JavaScript syntax
memberOrSpriteObjRef.soundChannel;
```

Description

RealMedia sprite or cast member property; allows you to get or set the sound channel used to play the audio in the RealMedia stream. Setting this property allows you to control the audio of a RealMedia stream using the Lingo sound methods and properties. Setting this property to a value outside the range 0–8 causes a Lingo error. This property has no effect if `realPlayerNativeAudio()` is set to `TRUE`.

The default setting for this property is 0, which means that the RealMedia audio will play in the highest sound channel available, and the property's value changes during playback depending on which channel is being used. When the RealMedia cast member is playing, this property reflects the sound channel currently in use. When the RealMedia cast member is stopped, this property reverts to 0.

If you specify a channel (1–8) for this property and there are sounds currently playing in that channel (from other parts of the movie), they will be stopped and the RealMedia audio will play in the channel instead.

Concurrently playing RealMedia cast members are not supported; if your movie contains RealMedia cast members that play concurrently, their sounds are played simultaneously in the same sound channel.

Example

The following examples show that the sound in the RealMedia stream in sprite 2 and the cast member Real will be played in sound channel 2.

```
-- Lingo syntax
put(sprite(2).soundChannel) -- 2
put(member("Real").soundChannel) -- 2

// JavaScript syntax
put(sprite(2).soundChannel); // 2
put(member("Real").soundChannel); // 2
```

The following examples assign sound channel 1 to the RealMedia stream in sprite 2 and the cast member Real.

```
-- Lingo syntax
sprite(2).soundChannel = 1
member("Real").soundChannel = 1

// JavaScript syntax
sprite(2).soundChannel = 1;
member("Real").soundChannel = 1;
```

See also

realPlayerNativeAudio()

soundDevice

Usage

```
-- Lingo syntax
_sound.soundDevice

// JavaScript syntax
_sound.soundDevice;
```

Description

Sound property; allows the sound mixing device to be set while the movie plays. Read/write.

The possible settings for `soundDevice` are the devices listed in `soundDeviceList`.

Several sound devices can be referenced. The various sound devices for Windows have different advantages.

- MacroMix (Windows®)—The lowest common denominator for Windows playback. This device functions on any Windows computer, but its latency is not as good as that of other devices.
- QT3Mix (Windows)—Mixes sound with QuickTime audio and possibly with other applications if they use DirectSound. This device requires that QuickTime be installed and has better latency than MacroMix.
- DirectSound® (Windows)—Similar to QT3Mix, but provides higher latency.
- MacSoundManager (Mac)—The only sound device available on the Mac.

Example

The following statement sets the sound device to the MacroMix for a Windows computer. If the newly assigned device fails, the `soundDevice` property is not changed.

```
-- Lingo syntax
_sound.soundDevice = "MacroMix"

// JavaScript syntax
_sound.soundDevice = "MacroMix";
```

See also

Sound, soundDeviceList

soundDeviceList

Usage

```
-- Lingo syntax
_sound.soundDeviceList

// JavaScript syntax
_sound.soundDeviceList;
```

Description

Sound property; creates a linear list of sound devices available on the current computer. Read-only.

For the Mac, this property lists one device, MacSoundManager.

Example

This statement displays a typical sound device list on a Windows computer:

```
-- Lingo syntax
trace(_sound.soundDeviceList)

// JavaScript syntax
trace(_sound.soundDeviceList);
```

See also

Sound, soundDevice

soundEnabled

Usage

```
-- Lingo syntax
_sound.soundEnabled

// JavaScript syntax
_sound.soundEnabled;
```

Description

Sound property; determines whether the sound is on (TRUE, default) or off (FALSE). Read/write.

When you set this property to FALSE, the sound is turned off, but the volume setting is not changed.

Example

This statement sets soundEnabled to the opposite of its current setting; it turns the sound on if it is off and turns it off if it is on:

```
-- Lingo syntax
_sound.soundEnabled = not(_sound.soundEnabled)

// JavaScript syntax
_sound.soundEnabled = !(_sound.soundEnabled);
```

See also

Sound

soundKeepDevice

Usage

```
-- Lingo syntax
_sound.soundKeepDevice

// JavaScript syntax
_sound.soundKeepDevice;
```

Description

Sound property; for Windows only, determines whether the sound driver unloads and reloads each time a sound needs to play. Read/write.

The default value of this property is `TRUE`, which prevents the sound driver from unloading and reloading each time a sound needs to play.

You may need to set this property to `FALSE` before playing a sound to ensure that the sound device is unloaded and made available to other applications or processes on the computer after the sound has finished.

Setting this property to `FALSE` may adversely affect performance if sound playback is used frequently throughout the Director application.

Example

This statement sets the `soundKeepDevice` property to `FALSE`:

```
-- Lingo syntax
_sound.soundKeepDevice = FALSE

// JavaScript syntax
_sound.soundKeepDevice = false;
```

See also

`Sound`

soundLevel

Usage

```
-- Lingo syntax
_sound.soundLevel

// JavaScript syntax
_sound.soundLevel;
```

Description

Sound property; returns or sets the volume level of the sound played through the computer's speaker. Read/write.

Possible values range from 0 (no sound) to 7 (the maximum, default).

In Windows, the system sound setting combines with the volume control of the external speakers. Thus, the actual volume that results from setting the sound level can vary. Avoid setting the `soundLevel` property unless you are sure that the result is acceptable to the user. It is better to set the individual volumes of the channels and sprites with the Sound Channel object's `volume` property.

These values correspond to the settings in the Mac Sound control panel. Using this property, script can change the sound volume directly or perform some other action when the sound is at a specified level.

To see an example of `soundLevel` used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement sets the variable `oldSound` equal to the current sound level:

```
-- Lingo syntax
```

```
oldSound = _sound.soundLevel

// JavaScript syntax
var oldSound = _sound.soundLevel;
```

This statement sets the sound level to 5:

```
-- Lingo syntax
_sound.soundLevel = 5

// JavaScript syntax
_sound.soundLevel = 5;
```

See also

Sound, volume (Windows Media)

soundMixMedia

Usage

```
-- Lingo syntax
_sound.soundMixMedia

// JavaScript syntax
_sound.soundMixMedia;
```

Description

Sound property; determines whether Flash cast members will mix their sound with sounds in the Score sound channels. Read/write.

This property defaults to TRUE for movies made with Director 7 and later and FALSE for earlier ones. It is also valid only on Windows.

When this property is TRUE, Flash cast members will mix their sound with sounds in the Score sound channels. Director takes over the mixing and playback of sounds from Flash cast members.

It is possible that slight differences may occur in the way Flash sounds play back. To hear the Flash sounds exactly they would be rendered in Flash, set this property to FALSE.

When this property is set to FALSE, Flash sounds will not be mixed and must be played at separate times.

Example

The following statements would show the default value of the soundMixMedia property in the authoring environment.

```
-- Lingo
put _sound.soundMixMedia
-- 1

// Javascript
trace( _sound.soundMixMedia);
// 1
```

See also

Sound

source

Usage

```
sprite(whichSprite).camera.backdrop[backdropIndex].source
member(whichCastmember).camera(whichCamera).backdrop
[backdropIndex].source
sprite(whichSprite).camera.overlay[overlayIndex].source
member(whichCastmember).camera(whichCamera).overlay
[overlayIndex].source
```

Description

3D backdrop and overlay property; allows you to get or set the texture to use as the source image for the overlay or backdrop.

Example

This statement sets the source of backdrop 1 to the texture Cedar:

```
sprite(3).camera.backdrop[1].source =
sprite(3).member.texture("Cedar")
```

See also

bevelDepth, overlay

sourceFileName

Usage

```
flashCastMember.sourceFileName
```

Description

Flash cast member property; specifies the pathname of the FLA source file to be used during launch-and-edit operations. This property can be tested and set. The default is an empty string.

Example

This statement sets the sourceFileName of the Flash cast member "SWF" to `C:\FlashFiles\myFile.fla`:

```
-- Lingo
member("SWF").sourceFileName = "C:\FlashFiles\myFile.fla"

// Javascript
member("SWF").sourceFileName = "C:\FlashFiles\myFile.fla";
```

sourceRect

Usage

```
-- Lingo syntax
windowObjRect.sourceRect

// JavaScript syntax
```

```
windowObjRect.sourceRect;
```

Description

Window property; specifies the original Stage coordinates of the movie playing in a window. Read-only.

This property is useful for returning a window to its original size and position after it has been dragged or its rectangle has been set.

Example

This statement displays the original coordinates of the Stage named Control_panel in the Message window:

```
-- Lingo syntax
put(window("Control_panel").sourceRect)

// JavaScript syntax
put(window("Control_panel").sourceRect);
```

See also

Window

specular (light)

Usage

```
member(whichCastmember).light(whichLight).specular
```

Description

3D light property; allows you to get or set whether specularity is on (TRUE) or off (FALSE). Specularity refers to the ability to have a highlight appear on a model where the light hitting the model is reflected toward the camera. The shininess of the model's shader determines how large the specular portion of the highlight is. The value for this property is ignored for ambient lights. The default value for this property is TRUE.

Note: *Turning off this property may increase performance.*

Example

The following statement sets the specular property of the light omni2 to FALSE. This light does not cause highlights. If this is the only light currently shining in the scene, there will be no specular highlights on any of the shaders in the scene.

```
-- Lingo
member("3d world").light("omni2").specular = FALSE

// Javascript
member("3d world").getPropRef("light",1).specular = false;
```

See also

silhouettes, specularLightMap

specular (shader)

Usage

```
member(whichCastmember).shader(whichShader).specular
```

Description

3D standard shader property; allows you get or set the specular color of a given shader. The specular color is the color of the highlight generated when specularity is turned on. There must be lights in the scene with the specular property set to TRUE, for this property to have a visible effect. The specular color is influenced by the color of the lights illuminating the object. If the specular color is white but the color of a light is red, there will be a red specular highlight appearing on the object. The default value for this property is rgb(255, 255, 255) which is white.

All shaders have access to the #standard shader properties; in addition to these standard shader properties shaders of the types #engraver, #newsprint, and #painter have properties unique to their type. For more information, see newShader.

Example

```
-- Lingo
put member("scene").shader("plutomat").specular
--rgb(11, 11, 11)

// Javascript
trace(member("scene").getPropRef("shader",1).specular
// <rgb(11,11,11)>
```

See also

silhouettes, specular (light), specularColor, emissive

specularColor

Usage

```
member(whichCastmember).specularColor
```

Description

3D cast member property; allows you to get or set the RGB value of the specular color of the first shader in the cast member. The first shader in the cast member's shader palette is always the default shader. This and all other 3D cast member properties are saved with the cast member and are restored the next time you open the movie. The default value for this property is rgb(255, 255, 255) which is white.

Example

The following statement sets the specular color of the first shader in the cast member Scene to rgb(255, 0, 0). It is equivalent to member("Scene").shader[1].specular = rgb(255, 0, 0). However, that syntax won't save the new value with the cast member when the movie is saved. Only member.specularColor will save the new color value.

```
-- Lingo
member("Scene").specularColor = rgb(255, 0, 0)

// Javascript
member("Scene").specularColor = rgb(255, 0, 0);
```

See also

silhouettes, specular (light), specular (shader)

specularLightMap

Usage

```
member(whichCastmember).shader(whichShader).specularLightMap
member(whichCastmember).model(whichModel).shader.specularLightMap
member(whichCastmember).model(whichModel).shaderList[shaderListIndex].specularLightMap
```

Description

3D standard shader property; allows you to get or set the fifth texture layer of a given standard shader. This property is ignored if the `toon` modifier is applied to the model resource.

The values that can be set are as follows:

- `textureModeList[5] = #specular`
- `blendFunctionList[5] = #add`
- `blendFunctionList[1] = #replace`
- `default = void`

This helper property provides a simple interface for setting up a common use of specular light mapping.

All shaders have access to the `#standard` shader properties; in addition to these standard shader properties shaders of the types `#engraver`, `#newsprint`, and `#painter` have properties unique to their type. For more information, see the `newShader`.

Example

This statement sets the texture Oval as the specularLightMap of the shader used by the model GlassBox:

```
-- Lingo
member("3DPlanet").model("GlassBox").shader.specularLightMap = \
member("3DPlanet").texture("Oval")

// Javascript
member("3DPlanet").getPropRef("model",1).getProp("shader").specularLightMap
=member("3DPlanet").getPropRef("texture",1);
```

See also

`diffuseLightMap`

spotAngle

Usage

```
member(whichCastmember).light(whichLight).spotAngle
```

Description

3D property; allows you to get or set the angle of the light projection cone. Light that is falls outside of the angle specified for this property, contributes no intensity. This property takes float value between 0.0 and 180.00, and has a default value of 90.0. The float value you specify corresponds to half the angle; for instance if you wish to specify a 90° angle you would pass a value of 45.0.

Example

This statement sets the `spotAngle` property of the light unidirectional to 8. The angle of the light projection cone will be 16°:

```
-- Lingo
member("3d world").light("unidirectional").spotAngle = 8

// Javascript
member("3d world").getPropRef("light",1).spotAngle = 8;
```

spotDecay

Usage

```
member(whichCastmember).light(whichLight).spotDecay
```

Description

3D light property; allows you get or set whether a spotlight's intensity falls off with the distance from the camera. The default value for this property is `FALSE`.

Example

The following statement sets the `spotDecay` property of light 1 to `TRUE`. Models that are farther away from light 1 will be less brightly lit than models that are closer to it.

```
-- Lingo
member("Scene").light[1].spotDecay = TRUE

// Javascript
member("Scene").getPropRef("light",1).spotDecay = true;
```

sprite (Movie)

Usage

```
-- Lingo syntax
_movie.sprite[spriteNameOrNum]

// JavaScript syntax
_movie.sprite[spriteNameOrNum];
```

Description

Movie property; provides indexed or named access to a movie sprite. Read-only.

The *`spriteNameOrNum`* argument can be either a string that specifies the name of the sprite or an integer that specifies the number of the sprite.

Example

The following statement sets the variable `sportSprite` to the movie sprite 5:

```
-- Lingo syntax
sportSprite = _movie.sprite[5]

// JavaScript syntax
var sportSprite = _movie.sprite[5];
```

See also

Movie

sprite (Sprite Channel)

Usage

```
-- Lingo syntax
spriteChannelObjRef.sprite

// JavaScript syntax
spriteChannelObjRef.sprite;
```

Description

Sprite Channel property; returns a reference to the sprite in the current frame of a sprite channel. Read-only.

Example

This statement sets the variable `mySprite` to the sprite in the sprite channel named Ribbon.

```
-- Lingo syntax
mySprite = channel("Ribbon").sprite

// JavaScript syntax
var mySprite = channel("Ribbon").sprite;
```

See also

Sprite Channel

spriteNum

Usage

```
-- Lingo syntax
spriteObjRef.spriteNum

// JavaScript syntax
spriteObjRef.spriteNum;
```

Description

Sprite property; determines the channel number the behavior's sprite is in and makes it available to any behaviors. Read-only.

Simply declare the property at the top of the behavior, along with any other properties the behavior may use.

If you use a `new()` handler to create an instance of the behavior, the script's `new()` handler must explicitly set the `spriteNum` property to the sprite's number. This provides a way to identify the sprite the script is attached to. The sprite's number must be passed to the `new()` handler as an argument when the `new()` handler is called.

Example

In this handler, the `spriteNum` property is automatically set for script instances that are created by the system:

```
-- Lingo syntax
property spriteNum, pMySpriteRef
```

```
on mouseDown me
    sprite(spriteNum).member = member("DownPict")
end

// JavaScript syntax
function mouseDown() {
    sprite(this.spriteNum).member = member("DownPict");
}
```

This handler uses the automatic value inserted into the `spriteNum` property to assign the sprite reference to a new property variable `pMySpriteRef`, as a convenience:

```
-- Lingo syntax
property spriteNum, pMySpriteRef

on beginSprite me
    pMySpriteRef = sprite(me.spriteNum)
end

// JavaScript syntax
function beginSprite() {
    this.pMySpriteRef = sprite(this.spriteNum);
}
```

This approach allows the use of the reference `pMySpriteRef` later in the script, with the handler using the syntax:

```
-- Lingo syntax
currMember = pMySpriteRef.member

// JavaScript syntax
var currMember = pMySpriteRef.member
```

instead of the following syntax which is somewhat longer:

```
Lingo syntax
currMember = sprite(spriteNum).member

// JavaScript syntax
var currMember = sprite(this.spriteNum).member
```

This alternative approach is merely for convenience, and provides no different functionality.

See also

`new()`, `Sprite`

stage

Usage

```
-- Lingo syntax
_movie.stage

// JavaScript syntax
_movie.stage;
```

Description

Movie property; refers to the main movie. Read-only.

This property is useful when sending a message to the main movie from a child movie.

Example

This statement displays the current setting for the Stage:

```
-- Lingo syntax
put(_movie.stage.rect)

// JavaScript syntax
put(_movie.stage.rect);
```

See also

Movie

startAngle

Usage

```
member(whichCastmember).modelResource(whichModelResource).
startAngle
modelResourceObjectReference.startAngle
```

Description

3D property; when used with a model resource whose type is `#cylinder` or `#sphere`, this command allows you to both get and set the `startAngle` property of the referenced model resource, as a floating-point value from 0.0 to 360.0. The default value for this property is 0.0.

The `startAngle` property determines the starting sweep angle of the model resource, and works in conjunction with the `endAngle` property to draw spheres and cylinders. For example, if you want to make a half sphere, set `startAngle` to 0.0 and `endAngle` to 180.0.

Example

The following statement sets the startAngle of the model resource Sphere01 to 0.0. If its `endAngle` is set to 90, then only one quarter of any model that uses this model resource will appear.

```
-- Lingo
put member("3D World").modelResource(1).startAngle
-- 0.0

// Javascript
trace(member("3D World").getPropRef("modelResource",1).startAngle);
// 0
```

See also

endAngle

startFrame

Usage

```
-- Lingo syntax
spriteObjRef.startFrame
```

```
// JavaScript syntax
spriteObjRef.startFrame;
```

Description

Sprite property; returns the frame number of the starting frame of a sprite span. Read-only.

This property is useful in determining the span in the Score that a particular sprite covers. It is available only in a frame that contains the sprite, and cannot be applied to sprites in different frames of the movie.

Example

This statement displays the starting frame of the sprite in channel 5 in the Message window:

```
-- Lingo syntax
put(sprite(5).startFrame)

// JavaScript syntax
put(sprite(5).startFrame);
```

See also

endFrame, Sprite

startTime

Usage

```
-- Lingo syntax
soundChannelObjRef.startTime

// JavaScript syntax
soundChannelObjRef.startTime;
```

Description

Sound Channel property; indicates the start time of the currently playing or paused sound as set when the sound was queued. Read-only.

This property cannot be set after the sound has been queued. If no value was supplied when the sound was queued, this property returns 0.

Example

This statement starts the digital video sprite in channel 5 at 100 ticks into the digital video:

```
-- Lingo syntax
sprite(5).startTime = 100

// JavaScript syntax
sprite(5).startTime = 100;
```

See also

Sound Channel

startTimeList

Usage

```
-- Lingo syntax
dvdObjRef.startTimeList

// JavaScript syntax
dvdObjRef.startTimeList;
```

Description

DVD property; a property list that specifies the time or chapter at which playback starts. Read/write.

A `startTimeList` is a property list that can be either chapter based or time based.

A chapter based `startTimeList` contains the following properties:

- `title`. Specifies the title that contains the chapter to play.
- `chapter`. Specifies the chapter to play.

This `startTimeList` starts playing at chapter 2 of title 1:

```
[#title:1, #chapter:2]
```

A time based `startTimeList` contains the following properties:

- `title`. Specifies the title.
- `hours`. Specifies the hour at which playback starts.
- `min`. Specifies the minute at which playback starts.
- `sec`. Specifies the second at which playback starts.
- `frames`. Specifies the frames at which playback starts.

This `startTimeList` starts playing at a specific time in title 1:

```
[#title:1, #hours:0, #minutes:45, #seconds:15, #frames:15]
```

This `startTimeList` only lists one time parameter:

```
[#title:1, #seconds:15]
```

The startTimeList can be cleared by setting it to 0.

Example

The following statement sets the startTimeList of the dvd object to 0.

```
-- Lingo
sprite(1).startTimeList = 0

// Javascript
sprite(1).startTimeList = 0;
```

See also

DVD, play() (DVD), stopTimeList

state (3D)

Usage

```
member(whichCastmember).state
```

Description

3D property; returns the current state of the referenced member in the streaming and loading process. This property refers to the initial file import or the last file load requested.

The `state` property of the member determines what, if any, 3D Lingo can be performed on the cast member.

This property can have any of the following values:

- `0`—indicates that the member is currently not loaded and therefore no 3D media are available. No 3D Lingo should be performed on the member.
- `1`—indicates that the media loading has begun.
- `2`—indicates that the member's initial load segment is loaded. All objects with a stream priority of zero, as set upon creation of the model file, will be loaded at this time, because they are part of the initial load segment. You can perform most 3D Lingo associated with objects that have a load priority of zero. Do not use the `loadFile` and `resetWorld` commands during this state.
- `3`—indicates that all the additional media of the member are being loaded and decompressed. Most 3D Lingo can be performed at this point. Do not use the `loadFile` and `resetWorld` commands during this state.
- `4`—indicates that all of the member's media have been loaded and all decompression is complete. All 3D Lingo can now be performed on the cast member.
- `-1`—indicates that an undefined error occurred during the media streaming process. Because the error may have occurred at any point during the loading process, the state of the cast member is not reliable.

In general, avoid using Lingo on 3D cast members with a current state lower than 3.

Example

This statement shows that the cast member named PartyScene has finished loading and preparing for playback, and no errors occurred during the load:

```
-- Lingo
put member("PartyScene").state
-- 4

// Javascript
trace(member("PartyScene").state);
// 4
```

state (Flash, SWA)

Usage

```
-- Lingo syntax
memberObjRef.state

// JavaScript syntax
memberObjRef.state;
```

Description

Cast member property; for Shockwave Audio (SWA) streaming cast members and Flash movie cast members, determines the current state of the streaming file. The properties `streamName`, `URL`, and `preLoadTime` can be changed only when the SWA sound is stopped.

The following properties for the SWA file return meaningful information only after the file begins streaming: `cuePointNames`, `cuePointTimes`, `currentTime`, `duration`, `percentPlayed`, `percentStreamed`, `bitRate`, `sampleRate`, and `numChannels`.

For SWA streaming cast members, the following values are possible:

- 0—Cast streaming has stopped.
- 1—The cast member is reloading.
- 2—Preloading ended successfully.
- 3—The cast member is playing.
- 4—The cast member is paused.
- 5—The cast member has finished streaming.
- 9—An error occurred.
- 10—There is insufficient CPU space.

For Flash movie cast members, this property returns a valid value only when the Director movie is running. The following values are possible:

- 0—The cast member is not in memory.
- 1—The header is currently loading.
- 2—The header has finished loading.
- 3—The cast member's media is currently loading.
- 4—The cast member's media has finished loading.
- -1—An error occurred.

This property can be tested but not set.

Example

This statement issues an alert if an error is detected for the SWA streaming cast member:

```
-- Lingo syntax
on mouseDown
    if member("Ella Fitzgerald").state = 9 then
        _player.alert("Sorry, can't find an audio file to stream.")
    end if
end

// JavaScript syntax
function mouseDown() {
    var ellaSt = member("Ella Fitzgerald").state;
    if (ellaSt == 9) {
        _player.alert("Sorry, can't find an audio file to stream.");
    }
}
```

See also

clearError(), getError() (Flash, SWA)

state (RealMedia)

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.state

// JavaScript syntax
memberOrSpriteObjRef.state;
```

Description

RealMedia sprite or cast member property; returns the current state of the RealMedia stream, expressed as an integer in the range 1 to 4. Each state value corresponds to a specific point in the streaming process. This property is dynamic during playback and can be tested but not set.

The streaming process is initiated when the playhead enters the span of the RealMedia sprite in the Score, the `play` method is invoked on a RealMedia sprite or cast member, or a user clicks the Play button in the RealMedia viewer. Calling this property returns a numeric value indicating the state of the streaming process for the RealMedia cast member. For each state, there is one or more corresponding `mediaStatus (RealMedia, Windows Media)` property value; each `mediaStatus` value is observed only in one state. For example, the `mediaStatus` property values `#seeking` and `#buffering` are present only when the value of `state` is 3.

The value of the `state` property provides important information in terms of performing Lingo on a cast member. If `member.state` is less than 2, some of the Lingo properties may be incorrect, and as a result, any Lingo relying on property data would be incorrect. When `member.state` is greater than or equal to 2 and less than 4, the RealMedia cast member is not displayed, but all the Lingo properties and methods have well-defined values and can be used to perform Lingo operations on the cast member.

When the streaming process is initiated, the `state` property cycles through the following states, unless an error (-1) occurs, which prevents the streaming process from starting:

-1 (error) indicates that there is something wrong, possibly a leftover error from the previous RealMedia stream. You may get more information by checking the `lastError` property. This state is the equivalent of `#error` for the `mediaStatus` property.

0 (closed) indicates that streaming has not begun, or that cast member properties are in initial states or are copies from an earlier playing of the cast member. This state is the equivalent of `#closed` for the `mediaStatus` property.

1 (connecting) indicates that streaming has begun but is in the very early stages of connecting to the server, and there is not enough information available locally to do anything with the cast member. This state is the equivalent of `#connecting` for the `mediaStatus` property.

2 (open) indicates that the Lingo properties have been refreshed from the actual stream. When `state` is greater than or equal to 2, the `height`, `width`, and `duration` properties of the RealMedia stream are known. This state is transitory and quickly changes to state 3. This state is the equivalent of `#opened` for the `mediaStatus` property.

3 (seeking or buffering) indicates that all of the RealMedia cast member's Lingo properties are current, but the cast member is not quite ready to play. The Stage or RealMedia viewer displays a black rectangle or the RealNetworks logo. If this state is the result of rebuffering due to network congestion, the `state` value quickly changes back to `4` (playing). This state is the equivalent of `#buffering` or `#seeking` for the `mediaStatus` property.

4 (playing) indicates that the RealMedia stream is playing (or paused) without problems or errors. This is the state during normal playback. This state is the equivalent of `#playing` or `#paused` for the `mediaStatus` property.

Example

The following examples show that the state of streams in sprite 2 and the cast member Real is 0, which is closed:

```
-- Lingo syntax
put(sprite(2).state) -- 0
put(member("Real").state) -- 0

// JavaScript syntax
put(sprite(2).state); // 0
put(member("Real").state); // 0
```

See also

`mediaStatus (RealMedia, Windows Media)`, `percentBuffered`, `lastError`

static

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.static

// JavaScript syntax
memberOrSpriteObjRef.static;
```

Description

Cast member property and sprite property; controls playback performance of a Flash movie sprite depending on whether the movie contains animation. If the movie contains animation (`FALSE`, default), the property redraws the sprite for each frame; if the movie doesn't contain animation (`TRUE`), the property redraws the sprite only when it moves or changes size.

This property can be tested and set.

Note: *Set the `static` property to `TRUE` only when the Flash movie sprite does not intersect other moving Director sprites. If the Flash movie intersects moving Director sprites, it may not redraw correctly.*

Example

This sprite script displays in the Message window the channel number of a Flash movie sprite and indicates whether the Flash movie contains animation:

```
-- Lingo syntax
property spriteNum

on beginSprite me
    if sprite(spriteNum).static then
        animationType = "does not have animation."
    else
        animationType = "has animation."
    end if
    put("The Flash movie in channel" && spriteNum && animationType)
end

// JavaScript syntax
function beginSprite() {
```

```
    var st = sprite(this.spriteNum).static;
    if (st == 1) {
        animationType = "does not have animation.";
    } else {
        animationType = "has animation.";
    }
    trace("The Flash movie in channel " + this.spriteNum + animationType);
}
```

staticQuality

Usage

```
-- Lingo syntax
spriteObjRef.staticQuality

// JavaScript syntax
spriteObjRef.staticQuality;
```

Description

QuickTime VR sprite property; specifies the codec quality used when the panorama image is static. Possible values are `#minQuality`, `#maxQuality`, and `#normalQuality`.

This property can be tested and set.

Example

This statement shows the staticQuality of the QuickTime sprite in the message window.

```
-- Lingo
put sprite(1).staticQuality
-- #normalQuality

// Javascript
trace(sprite(1).staticQuality)
// symbol(normalQuality)
```

status

Usage

```
-- Lingo syntax
soundChannelObjRef.status

// JavaScript syntax
soundChannelObjRef.status;
```

Description

Sound Channel property; indicates the status of a sound channel. Read-only.

Possible values include:

Status	Name	Meaning
0	Idle	No sounds are queued or playing.
1	Loading	A queued sound is being preloaded but is not yet playing.
2	Queued	The sound channel has finished preloading a queued sound but is not yet playing the sound.
3	Playing	A sound is playing.
4	Paused	A sound is paused.

Example

This statement displays the current status of sound channel 2 in the Message window:

```
-- Lingo syntax
put(sound(2).status)

// JavaScript syntax
put(sound(2).status);
```

See also

Sound Channel

stillDown

Usage

```
-- Lingo syntax
_mouse.stillDown

// JavaScript syntax
_mouse.stillDown;
```

Description

Mouse property; indicates whether the user is pressing the mouse button (TRUE) or not (FALSE). Read-only.

This function is useful within a mouseDown script to trigger certain events only after the mouseUp function.

Script cannot test stillDown when it is used inside a loop. Use the mouseDown function inside loops instead.

Example

This statement checks whether the mouse button is being pressed and calls the handler dragProcedure if it is:

```
-- Lingo syntax
if (_mouse.stillDown) then
    dragProcedure
end if

// JavaScript syntax
if (_mouse.stillDown) {
    dragProcedure();
}
```

See also

Mouse, mouseDown, mouseUp

stopTime

Usage

```
sprite(whichSprite).stopTime
the stopTime of sprite whichSprite
```

Description

Sprite property; determines when the specified digital video sprite stops. The value of `stopTime` is measured in ticks.

This property can be tested and set.

Example

This statement stops the digital video sprite in channel 5 at 100 ticks into the digital video:

```
-- Lingo
sprite(5).stopTime = 100

// Javascript
sprite(5).stopTime = 100;
```

stopTimeList

Usage

```
-- Lingo syntax
dvdObjRef.stopTimeList

// JavaScript syntax
dvdObjRef.stopTimeList;
```

Description

DVD property; a property list that specifies the time or chapter at which playback stops.
Read/write.

A `stopTimeList` is a property list that can be either chapter based or time based.

A chapter based `stopTimeList` contains the following properties:

- `title`. Specifies the title.
- `chapter`. Specifies the chapter. Playback stops after this chapter is played.

This stopTimeList stops playing at chapter 4 of title 1:

```
[#title:1, #chapter:4]
```

A time based `stopTimeList` contains the following properties:

- `title`. Specifies the title.
- `hours`. Specifies the hour at which playback stops.
- `min`. Specifies the minute at which playback stops.
- `sec`. Specifies the second at which playback stops.
- `frames`. Specifies the frames at which playback stops.

This `stopTimeList` stops playing at a specific time in title 1:

```
[#title:1, #hours:0, #minutes:55, #seconds:45, #frames:15]
```

This `stopTimeList` only lists one time parameter:

```
[#title:1, #seconds:45]
```

The stopTimeList can be cleared by setting it to 0.

See also

`DVD`, `play() (DVD)`, `startTimeList`

streamMode

Usage

```
-- Lingo syntax
memberObjRef.streamMode

// JavaScript syntax
memberObjRef.streamMode;
```

Description

Flash cast member property; controls the way a linked Flash movie cast member is streamed into memory, as follows:

- `#frame` (default)—Streams part of the cast member each time the Director frame advances while the sprite is on the Stage.
- `#idle`—Streams part of the cast member each time an idle event is generated or at least once per Director frame while the sprite is on the Stage.
- `#manual`—Streams part of the cast member into memory only when the `stream` command is issued for that cast member.

This property can be tested and set.

Example

This `startMovie` script searches the internal cast for Flash movie cast members and sets their `streamMode` properties to `#manual`:

```
-- Lingo syntax
on startMovie
    repeat with i = 1 to castLib(1).member.count
    if member(i, 1).type = #flash then
        member(i, 1).streamMode = #manual
    end if
    end repeat
end

// JavaScript syntax
function startMovie() {
    i = 1;
    while( i < (castLib(whichCast).member.count) + 1)
        var tp = member(i, whichCast).type;
        if (tp == "flash") {
            member(i, 1).streamMode = symbol("manual");
            i++;
```

```
        }
    }
}
```

streamName

Usage

```
-- Lingo syntax
memberObjRef.streamName

// JavaScript syntax
memberObjRef.streamName;
```

Description

Shockwave Audio (SWA) cast member property; specifies a URL or filename for a streaming cast member. This property functions the same as the `URL` member property.

This property can be tested and set.

Example

The following statement links the file BigBand.swa to an SWA streaming cast member. The linked file is on the disk MyDisk in the folder named Sounds.

```
-- Lingo syntax
member("SWAstream").streamName = "MyDisk/sounds/BigBand.swa"

// JavaScript syntax
member("SWAstream").streamName = "MyDisk/sounds/BigBand.swa";
```

streamSize

Usage

```
-- Lingo syntax
memberObjRef.streamSize

// JavaScript syntax
memberObjRef.streamSize;
```

Description

Cast member property; reports an integer value indicating the total number of bytes in the stream for the specified cast member. The `streamSize` property returns a value only when the Director movie is playing.

This property can be tested but not set.

Example

The following frame script checks to see if a Flash movie cast member named Intro Movie has finished streaming into memory. If it hasn't, the script updates a field cast member to indicate the number of bytes that have been streamed (using the `bytesStreamed` member property) and the total number of bytes for the cast member (using the `streamSize` member property). The script keeps the playhead looping in the current frame until the movie finishes loading into memory.

```
-- Lingo syntax
```

```
on exitFrame
    if member("Intro Movie").percentStreamed < 100 then
        member("Message Line").text = string(member("Intro Movie").bytesStreamed) && "of" \
&& string(member("Intro Movie").streamSize) && "bytes have downloaded so far."
        _movie.go(_movie.frame)
    end if
end

// JavaScript syntax
function exitFrame() {
    var pctStm = member("Intro Movie").percentStreamed;
    var strSs = new String(member("Intro Movie").streamSize);
    var strIm = new String(member("Intro Movie").bytesStreamed);
    if (pctStm < 100) {
        member("Message Line").text = strStm + " of " + strSS+ " bytes have downloaded so
far.";
        _ movie.go(_movie.frame);
    }
}
```

streamSize (3D)

Usage

```
member(whichCastmember).streamSize
```

Description

3D property; allows you to get the size of the data stream to be downloaded, from 0 to maxInteger. This command refers to the initial file import or the last `loadFile()` requested.

Example

This statement shows that the last file load associated with the cast member Scene has a total size of 325300 bytes:

```
put member("Scene").streamSize
-- 325300
```

See also

bytesStreamed (3D), percentStreamed (3D), state (3D), preLoad (3D)

strokeColor

Usage

```
-- Lingo syntax
memberObjRef.strokeColor

// JavaScript syntax
memberObjRef.strokeColor;
```

Description

Vector shape cast member property; indicates the color in RGB of the shape's framing stroke.

To see an example of `strokeColor` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This sets the strokeColor of cast member "line" to red:

```
-- Lingo syntax
member("line").strokeColor = color(255, 0, 0)

// JavaScript syntax
member("line").strokeColor = color(255, 0, 0);
```

See also

color(), fillColor, endColor, backgroundColor

strokeWidth

Usage

```
-- Lingo syntax
memberObjRef.strokeWidth

// JavaScript syntax
memberObjRef.strokeWidth;
```

Description

Vector shape cast member property; indicates the width, in pixels, of the shape's framing stroke.

The value is a floating-point number between 0 and 100 and can be tested and set.

To see an example of `strokeWidth` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

The following code sets the `strokeWidth` of cast member "line" to 10 pixels:

```
-- Lingo syntax
member("line").strokeWidth = 10

// JavaScript syntax
member("line").strokeWidth = 10;
```

style

Usage

```
member(whichCastmember).model(whichModel).toon.style
member(whichCastmember).model(whichModel).shader.style
member(whichCastmember).shader(whichShader).style
```

Description

3D `toon` modifier and `painter` shader property; indicates the way the `toon` modifier and `painter` shader apply color to a model. Possible values are as follows:

- `#toon` uses sharp transitions between colors.
- `#gradient` uses smooth transitions between colors. This is the default value.

- `#blackAndWhite` uses two-color black and white.

The number of colors used by the `toon` modifier and `painter` shader is set with the `colorSteps` property of the modifier or shader.

Example

The following statement sets the `style` property of the toon modifier for the model named Teapot to `#blackAndWhite`. The model will be rendered in two-color black and white

```
member("Shapes").model("Teapot").toon.style = #blackAndWhite
```

subdivision

Usage

```
member(whichCastmember).model(whichModel).sds.subdivision
```

Description

3D `sds` modifier property; allows you to get or set the subdivision surfaces mode of operation. Possible values are as follows:

- `#uniform` specifies that the mesh is uniformly scaled up in detail, with each face subdivided the same number of times.
- `#adaptive` specifies that additional detail is added only when there are large surface orientation changes and only to those areas of the mesh that are currently visible.

The `sds` modifier cannot be used with the `inker` or `toon` modifiers, and caution should be used when using the `sds` modifier with the `lod` modifier. See the `sds` modifier entry for more information.

Example

The following statement sets the `subdivision` property of the `sds` modifier of the model named Baby to `#adaptive`. Baby's geometry will not be modified uniformly.

```
-- Lingo
member("Scene").model(1).sds.subdivision = #adaptive

// Javascript
Member("Scene").getPropRef("model",1).sds.subdivision = symbol("adaptive");
```

See also

`sds (modifier)`, `error`, `enabled (sds)`, `depth (3D)`, `tension`

subPicture

Usage

```
-- Lingo syntax
dvdObjRef.subPicture

// JavaScript syntax
dvdObjRef.subPicture;
```

Description

DVD property. Determines the current subpicture, if any. Read/write.

The value of `subPicture` is an integer. A value of 0 disables `subPicture`.

See also

DVD

subPictureCount

Usage

```
-- Lingo syntax
dvdObjRef.subPictureCount

// JavaScript syntax
dvdObjRef.subPictureCount;
```

Description

DVD property. Returns the number of available sub pictures. Read-only.

See also

DVD

suspendUpdates

Usage

```
sprite(which3dSprite).suspendUpdates
```

Description

3D sprite property; when set to `TRUE`, causes the sprite not to be updated as part of normal screen redraw operations. This can improve movie playback performance. Certain kinds of screen updates will still affect the sprite, such as those due to dragging another window over the sprite. When the `suspendUpdates` property is set to `FALSE`, the sprite is redrawn normally.

It is important to keep the `suspendUpdates` property set to `FALSE` while any element within the 3D sprite is being animated.

Example

The following statement suspends the updates on the sprite(3).

```
-- Lingo
sprite(3).suspendUpdates = FALSE

// Javascript
sprite(3).suspendUpdates = false;
```

switchColorDepth

Usage

```
-- Lingo syntax
_player.switchColorDepth

// JavaScript syntax
_player.switchColorDepth;
```

Description

Player property; determines whether Director switches the monitor that the Stage occupies to the color depth of the movie being loaded (`TRUE`) or leaves the color depth of the monitor unchanged when a movie is loaded (`FALSE`, default). Read/write.

When `switchColorDepth` is `TRUE`, nothing happens until a new movie is loaded.

Setting the monitor's color depth to that of the movie is good practice.

- When the monitor's color depth is set below that of the movie, resetting it to the color depth of the movie (assuming that the monitor can provide that color depth) helps maintain the movie's original appearance.
- When the monitor's color depth is higher than that of the movie, reducing the monitor's color depth plays the movie using the minimum amount of memory, loads cast members more efficiently, and causes animation to occur more quickly.

The value of this property can also be set using the Reset Monitor to Movie's Color Depth option in the General Preferences dialog box.

Example

This statement sets the variable named `switcher` to the current setting of `switchColorDepth`:

```
-- Lingo syntax
switcher = _player.switchColorDepth

// JavaScript syntax
var switcher = _player.switchColorDepth;
```

This statement checks whether the current color depth is 8-bit and turns the `switchColorDepth` property on if it is:

```
-- Lingo syntax
if (_system.colorDepth = 8) then
    _player.switchColorDepth = TRUE
end if

// JavaScript syntax
if (_system.colorDepth == 8) {
    _player.switchColorDepth = true;
}
```

See also

colorDepth, Player

systemTrayIcon

Usage

```
-- Lingo syntax
_movie.displayTemplate.systemTrayIcon
windowObjRef.systemTrayIcon

// JavaScript syntax
_movie.displayTemplate.systemTrayIcon;
windowObjRef.systemTrayIcon;
```

Description

Movie and Windows property (Microsoft Windows only). Determines whether a window has an icon in the system tray of a user's desktop. Read/write.

If `systemTrayIcon` is `TRUE`, a window icon is placed in the system tray.

If `systemTrayIcon` is `FALSE`, no icon appears in the system tray.

Example

The following statement shows the system tray icon when the movie is playing.

```
-- Lingo
_movie.displayTemplate.systemTrayIcon = TRUE

// Javascript
_movie.displayTemplate.systemTrayIcon = true;
```

See also

`displayTemplate`, `Movie`, `systemTrayTooltip`, `Window`

systemTrayTooltip

Usage

```
-- Lingo syntax
_movie.displayTemplate.systemTrayTooltip
windowObjRef.systemTrayTooltip

// JavaScript syntax
_movie.displayTemplate.systemTrayTooltip;
windowObjRef.systemTrayTooltip;
```

Description

Movie and Windows property (Microsoft Windows only). Determines the string that appears in the tooltip pop-up of the system tray icon. Read/write.

This property is only applicable if the `systemTrayIcon` property is set to `TRUE`. If `systemTrayIcon` is `TRUE`, the tooltip will appear when a user mouses over the system tray icon.

The default value of `systemTrayTooltip` is the title of the window.

See also

`displayTemplate`, `Movie`, `systemTrayIcon`, `Window`

tabCount

Usage

```
chunkExpression.tabCount
```

Description

Text cast member property; indicates how many unique tab stops are in the specified chunk expression of the text cast member.

The value is an integer equal to or greater than 0, and may be tested but not set.

Example

The following statement shows the number of tabs in a text field.

```
-- Lingo
put member(1).tabCount
-- 3

// Javascript
trace(member(1).tabCount);
// 3
```

tabs

Usage

```
member(whichTextMember).tabs
```

Description

Text cast member property; this property contains a list of all the tab stops set in the text cast member.

Each element of the list contains information regarding that tab for the text cast member. The possible properties in the list are as follows:

#type	Can be `#left`, `#center`, `#right`, or `#decimal`.
#position	Integer value indicating the position of the tab in points.

You can get and set this property. When `tabs` is set, the `type` property is optional. If `type` is not specified, the tab type defaults to `#left`.

Example

This statement retrieves and displays in the Message window all the tabs for the text cast member Intro credits:

```
-- Lingo
put member("Intro credits").tabs
-- [[#type: #left, #position: 36], [#type: #Decimal, #position: 141], [#type: #right,
#position: 216]]

// Javascript
trace(member("Intro credits").tabs)
// < [[#type: #left, #position: 36], [#type: #Decimal, #position: 141], [#type: #right,
#position: 216]]>
```

target

Usage

```
timeoutObject.target
```

Description

Timeout object property; indicates the child object that the given *timeoutObject* will send its timeout events to. Timeout objects whose target property is VOID will send their events to a handler in a movie script.

This property is useful for debugging behaviors and parent scripts that use timeout objects.

Example

This statement displays the name of the child object that will receive timeout events from the timeout object timerOne in the Message window:

```
-- Lingo
put timeout("timerOne").target

// Javascript
trace(timeout("timerOne").target)
```

See also

name (timeout), timeout(), timeoutHandler, timeoutList

targetFrameRate

Usage

```
sprite(which3dSprite).targetFrameRate
```

Description

3D sprite property; determines the preferred number of frames per second to use when rendering a 3D sprite. The default value is 30 frames per second. The targetFrameRate property is only used if the useTargetFrameRate property is set to TRUE. If the useTargetFrameRate property is set to TRUE, Director will reduce the polygon count of the models in the sprite if necessary to maintain the specified frame rate.

Example

These statements set the targetFrameRate property of sprite 3 to 45 and enforce the frame rate by setting the useTargetFrameRate property of the sprite to TRUE:

```
-- Lingo
sprite(3).targetFrameRate = 45
sprite(3).useTargetFrameRate = TRUE

// Javascript
sprite(3).targetFrameRate = 45;
sprite(3).useTargetFrameRate = true;
```

See also

useTargetFrameRate

tension

Usage

```
member(whichCastmember).model(whichModel).sds.tension
```

Description

3D subdivision surface property; allows you to get or set a floating-point percentage between 0.0 and 100.0 that controls how tightly the newly generated surface matches the original surface. The higher this value, the more tightly the subdivided surface matches the original surface. The default is 65.0.

Example

The following statement sets the `tension` property of the `sds` modifier of the model Baby to 35. If the `sds` modifier's `error` setting is low and its `depth` setting is high, this statement will have a very pronounced effect on Baby's geometry.

```
-- Lingo
member("scene").model("Baby").sds.tension = 35

// Javascript
member("scene").getPropRef("model",1).sds.tension = 35;
```

See also

`sds (modifier)`, `error`, `depth (3D)`

text

Usage

```
-- Lingo syntax
memberObjRef.text

// JavaScript syntax
memberObjRef.text;
```

Description

Text cast member property; determines the character string in the field cast member specified by *`whichCastMember`*.

The `text` cast member property is useful for displaying messages and recording what the user types.

This property can be tested and set.

When you use Lingo to change the entire text of a cast member you remove any special formatting you have applied to individual words or lines. Altering the `text` cast member property reapplies global formatting. To change particular portions of the text, refer to lines, words, or items in the text.

When the movie plays back as an applet, this property's value is `""` (an empty string) for a field cast member whose text has not yet streamed in.

To see an example of `text` used in a completed movie, see the Forms and Post, and Text movies in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement places the phrase "Thank you." in the empty cast member Response:

```
--Lingo syntax
if (member("Response").text = EMPTY) then
    member("Response").text = "Thank You."
end if

// JavaScript syntax
if (member("Response").text == " ") {
    member("Response").text = "Thank You.";
}
```

This statement sets the content of cast member Notice to "You have made the right decision!"

```
--Lingo syntax
member("Notice").text = "You have made the right decision!"

// JavaScript syntax
member("Notice").text = "You have made the right decision!";
```

See also

selEnd, selStart

texture

Usage

```
member(whichCastmember).texture(whichTexture)
member(whichCastmember).texture[index]
member(whichCastmember).shader(whichShader).texture
member(whichCastmember).model(whichModel).shader.texture
member(whichCastmember).model(whichModel).shaderList.texture
member(whichCastmember).model(whichModel).shaderList[index].texture
member(whichCastmember).modelResource(whichParticleSystemModelResource).texture
```

Description

3D element and shader property; an image object used by a shader to define the appearance of the surface of a model. The image is wrapped on to the geometry of the model by the shader.

The visible component of a shader is created with up to eight layers of textures. These eight texture layers are either created from bitmap cast members or image objects within Director or imported with models from 3D modeling programs.

Create and delete textures with the `newTexture()` and `deleteTexture()` commands.

Textures are stored in the texture palette of the 3D cast member. They can be referenced by name (*`whichTexture`*) or palette index (*`textureIndex`*). A texture can be used by any number of shaders. Changes to a texture will appear in all shaders which use that texture.

There are three types of textures:

`#fromCastmember`; the texture is created from a bitmap cast member using the `newTexture()` command.

`#fromImageObject`; the texture is created from a lingo image object using the `newTexture()` command.

`#importedFromFile`; the texture is imported with a model from a 3D modeling program.

For more information about texture properties, see the Using Director topics in the Director Help Panel.

The texture of a particle system is a property of the model resource, whose type is `#particle`.

Example

This statement sets the `texture` property of the shader named WallSurface to the texture named BluePaint:

```
-- Lingo
member("Room").shader("WallSurface").texture = member("Room").texture("BluePaint")

// Javascript
member("Room").getPropRef("shader",1).texture = member("Room").getPropRef("texture",1);
```

See also

`newTexture`, `deleteTexture`

textureCoordinateList

Usage

```
member(whichCastmember).modelResource(whichmodelResource).
textureCoordinateList
modelResourceObjectReference.textureCoordinateList
```

Description

3D property; when used with a model resource whose type is `#mesh`, or with a `meshDeform` modifier attached to a model, this property allows you to get or set the `textureCoordinateList` property of the model resource.

The `textureCoordinateList` property is a list of sublists identifying locations in an image that are to be used when texture mapping a triangle. Each sublist consists of two values indicating a location in a texture map. The values must be between 0.0 and 1.0 so that they can be scaled to arbitrarily sized texture maps. The default is an empty list.

Manipulate `modelResource.face[index].textureCoordinates` or `model.meshdeform.mesh[index].face[index]` to change the mapping between `textureCoordinates` and the corners of a mesh face.

Example

```
-- Lingo
put member(5).modelResource("mesh square").textureCoordinateList
--[ [0.1, 0.1], [0.2, 0.1], [0.3, 0.1], [0.1, 0.2], [0.2, 0.2], [0.3, 0.2], [0.1, 0.3], [0.2,
0.3], [0.3, 0.3] ]

// Javascript
trace(member(5).getPropRef("modelResource",1).textureCoordinateList;
// <[ [0.1, 0.1], [0.2, 0.1], [0.3, 0.1], [0.1, 0.2], [0.2, 0.2], [0.3, 0.2], [0.1, 0.3],
[0.2, 0.3], [0.3, 0.3] ]>
```

See also

`face`, `texture`, `meshDeform (modifier)`

textureCoordinates

Usage

```
member(whichCastmember).modelResource(whichModelResource).face[faceIndex].textureCoordinat
es
modelResourceObject.face[faceIndex].textureCoordinates
```

Description

3D property; identifies which elements in the `textureCoordinateList` to use for the *`faceIndex'd`* face. This property must be a list of three integers specifying indices in the `textureCoordinateList`, corresponding to the `textureCoordinates` to use for each corner of the mesh's face.

See also

`face`, `textureCoordinateList`

textureLayer

Usage

```
member(whichCastmember).model(whichModel).meshDeform.mesh[index].textureLayer.count
member(whichCastmember).model(whichModel).meshdeform.mesh[index].texturelayer.add()
member(whichCastmember).model(whichModel).meshdeform.mesh[index].texturelayer[index].textu
reCoordinateList.
```

Description

3D `meshdeform` modifier properties; using these properties you can get and set information about the texture layers of a specified mesh.

You can have up to eight texture layers for a shader, each layer can contain only one texture, but the same texture can be specified for more than one layer. Texture layers are layers of textures used by shaders.

Use the following properties to access and manipulate texture layers:

`meshdeform.mesh[index].texturelayer.count` returns the number of texture layers for the specified mesh.

`model.meshdeform.mesh[index].texturelayer.add()` adds an empty texture layer to the specified mesh.

`model.meshdeform.mesh[index].texturelayer[index].texturecoordinatelist` allows you to set or get a list of textureCoordinates for a particular layer of the specified mesh. You can also use this property to copy texture coordinates between texture layers as follows:

```
model.meshdeform.texturelayer[a].texturecoordinatelist =
model.meshdeform.texturelayer[b].texturecoordinatelist
```

```
See also
meshDeform (modifier), mesh (property), textureCoordinateList, add (3D texture), count,
texture, textureModeList
```

textureList

Usage

```
member(whichMember).model(whichModel).shader(whichShader).textureList
member(whichMember).model(whichModel).shader(whichShader).textureList[index]
```

Description

3D shader property; determines the list of textures applied to the shader. A shader can have up to 8 layers of textures. When tested, this property returns a linear list of texture objects. When set without specifying an index, this property specifies the texture object to be applied to all layers. Setting the `textureList` property to `VOID` disables texturing for all layers. The default value is `VOID`.

To test or set the texture object for a specific texture layer, include an index value.

Example

This statement sets the 3rd texture layer of the shader named "WallSurface" to the texture named "BluePaint" in the cast member named "Room":

```
-- Lingo
member(3).model("Car").shader("WallSurface").textureList[3] =
member("Room").texture("BluePaint")

// Javascript
member(3).getPropRef("model",1).getPropRef("shader",1).textureList[3] =
member("Room").getPropRef("texture",1);
```

See also

textureModeList

textureMember

Usage

```
member(whichCastmember).textureMember
```

Description

3D cast member property; indicates the name of the bitmap cast member used as the source of the default texture for the 3D cast member.

The 3D cast member's `textureType` property must be set to `#member` for the `textureMember` property to be effective.

Example

The following statement sets the `textureMember` property of the cast member named YardScene to "Fence". If the `textureType` property of YardScene is set to `#member`, the cast member named Fence will be the source bitmap for the default texture in YardScene.

```
-- Lingo
member("YardScene").textureMember = "Fence"

// Javascript
member("YardScene").textureMember = "Fence";
```

See also

textureType

textureMode

Usage

```
member(whichCastmember).shader(whichShader).textureMode
member(whichCastmember).model(whichModel).shader.textureMode
member(whichCastmember).model(whichModel).shaderList{[index]}.textureMode
```

Description

3D `#standard` shader property; specifies how the first texture layer is mapped on to the surface of the model. Use the `textureModeList` property to specify textures for layers other than the first layer. This property is ignored if the `#toon` modifier is applied to the model resource.

The possible values of this property are `#none`, `#wrapPlanar`, `#wrapCylindrical`, `#wrapSpherical`, `#reflection`, `#diffuseLight`, and `#specularLight`. For descriptions of these terms, see `textureModeList`.

Example

This statement sets the value of the `textureMode` property of the first texture layer of the shader of the model named Ball to `#wrapSpherical`:

```
-- Lingo
member("scene").model("Ball").shader.textureMode = #wrapSpherical

// Javascript
member("scene").getPropRef("model",1).getProp("shader").textureMode =
symbol("wrapSpherical");
```

See also

`textureModeList`

textureModeList

Usage

```
member(whichCastmember).shader(whichShader).textureModeList
member(whichCastmember).shader(whichShader).
textureModeList[textureLayerIndex]
member(whichCastmember).model(whichModel).shader.textureModeList
member(whichCastmember).model(whichModel).shader.
textureModeList[textureLayerndex]
```

Description

3D standard shader property; allows you to change how a textureLayer is mapped on to the surface of a model. This property is ignored if the `#toon` modifier is applied to the model resource. Possible values are as follows:

- `#none` uses the texture coordinate values originally defined for the model resource. This setting disables `wrapTransform and wrapTransformList[textureLayerIndex].`
- `#wrapPlanar` wraps the texture on the model surface as though it were being projected from an overhead projector. The shader's `wrapTransformList[textureLayerIndex]` is applied to the mapping space before the texture coordinates are generated in model space. With an identity `wrapTransformList[textureLayerIndex]` (the default), the planar mapping is oriented such that the texture is extruded along the Z axis with the texture's up direction along the Y axis.
- `#wrapCylindrical` wraps the texture around the surface as though the surface were placed in the middle of the texture and then the texture were rolled around the surface to form a cylinder. The `wrapTransformList[textureLayerIndex]` is applied to the mapping space before the texture coordinates are generated in model space. With an identity `wrapTransformList[textureLayerIndex]` (the default), the cylindrical mapping is oriented such that the texture is wrapped from the -Y axis, starting at the left edge of the texture, toward the +X axis, around the Z axis. The up direction of the texture is toward the +Z axis.

- `#wrapSpherical` wraps the texture around the surface as though the surface were placed in the middle of the texture and then all four corners of the texture were pulled up to meet at the top. The `wrapTransformList[textureLayerIndex]` is applied to the mapping space before the texture coordinates are generated in model space. With an identity `wrapTransformList[textureLayerIndex]`, the spherical mapping is located at the model space origin and oriented such that the texture is wrapped from the -Y axis, starting at the left edge of the texture, toward the +X axis, around the Z axis. The up direction of the texture is toward the +Z axis.

- `#reflection` is similar to `#wrapSpherical` except that the new texture coordinates are continuously reprojected on to the surface from a fixed orientation. When the model rotates, the texture coordinates will not rotate with it. Simulates light reflected on an object by its environment. This setting disables `wrapTransform`.

- `#diffuseLight` generates diffuse light mapping texture coordinate values, one per vertex, and stores the results in the referenced mesh. This setting disables `wrapTransform`.

- `#specularLight` generates specular light mapping texture coordinate values, one per vertex, and stores the results in the referenced mesh. This setting disables `wrapTransform`.

Example

In this example, a shader is set up to simulate a reflective garden ball. The shader's first textureLayer is set to a spherical mapping and the third textureLayer is set to use a #refection style mapping. The shader's textureList[3] entry will appear to reflected from the environment onto all models which use this shader.

```
-- Lingo
member("scene").shader("GardenBall).textureList[1] =
member("scene").texture("FlatShinyBall")
member("scene").shader("GardenBall").textureModeList[1] = #wrapSpherical
member("scene").shader("GardenBall").textureList[3] =
member("scene").texture("GardenEnvironment")
member("scene").shader("GardenBall").textureModeList[3] = #reflection

// Javascript
member("scene").getPropRef("shader",1).textureList[1] =
member("scene").getPropRef("texture",1);
member("scene").getPropRef("shader",1).textureModeList[1] = symbol("wrapSpherical");
member("scene").getPropRef("shader",1).textureList[3] =
member("scene").getPropRef("texture",1);
member("scene").getPropRef("shader",1).textureModeList[3] = symbol("reflection");
```

See also

`textureTransformList`, `wrapTransform`

textureRenderFormat

Usage

```
getRenderServices().textureRenderFormat
```

Description

3D `rendererServices` property; allows you to get or set the default bit format used by all textures in all 3d cast members. Use a texture's texture.RenderFormat property to override this setting for specific textures only. Smaller sized bit formats (i.e 16 bit variants such as #rgba5551) use less hardware accelerator video ram, allowing you to make use of more textures before being forced to switch to software rendering. Larger sized bit formats (i.e. 32 bit variants such as #rgba8888) generally look better. In order to use alpha transparency in a texture, the last bit must be nonzero. In order to get smooth transparency gradations the alpha channel must have more than 1 bit of precision.

Each pixel formats has four digits, with each digit indicating the degree of precision for red, green, blue, and alpha. The value you choose determines the accuracy of the color fidelity (precision of the alpha channel) and the amount of memory used by the hardware texture buffer. You can choose a value that improves color fidelity or a value that allows you to fit more textures on the card. You can fit twice as many 16-bit textures as 32-bit textures in the same space. If a movie uses more textures than fit on a card at a the same time, Director switches to `#software` rendering.

You can specify any of the following values for `textureRenderFormat`:

- `#rgba8888`: 32 bit color mode with 8 bits each for red, green, blue, and alpha
- `#rgba8880`: same as above, with no alpha value
- `#rgba5650`: 16-bit color mode with no alpha; 5 bits for red, 6 for green, 5 for blue
- `#rgba5550`: 16-bit color mode with no alpha; 5 bits each for red, green, and blue, with no alpha measure
- `#rgba5551`: 16-bit color mode with 5 bits each for red, green, and blue; 1 bit for alpha
- `#rgba4444`: 16-bit color mode with 4 bits each for red, green, blue, and alpha

The default value is `#rgba5551`.

Example

The following statement sets the global `textureRenderFormat` for the 3D member to `#rgba8888`. Each texture in this movie will be rendered in 32-bit color unless its `texture.renderFormat` property is set to a value other than `#default`.

```
-- Lingo
getRendererServices().textureRenderFormat = #rgba8888

// Javascript
getRendererServices().textureRenderFormat = symbol("rgba8888");
```

See also

`renderer`, `preferred3dRenderer`, `renderFormat`, `getRendererServices()`

textureRepeat

Usage

```
member(whichCastmember).shader(whichShader).textureRepeat
member(whichCastmember).model(whichModel).shader.textureRepeat
member(whichCastmember).model(whichModel).shaderList{[index]}.textureRepeat
```

Description

3D `#standard` shader property; controls the texture clamping behavior of the first texture layer of the shader. Use the `textureRepeatList` property to control this property for texture layers other than the first layer.

When `textureRepeat` is set to `TRUE` and the value of the x and/or y components of `shaderReference.textureTransform.scale` is less than 1, the texture is tiled (repeated) across the surface of the model.

When `textureRepeat` is set to `FALSE`, the texture will not tile. If the value of the x and/or y components of `shaderReference.textureTransform.scale` is less than 1, any area of the model not covered by the texture will be black. If the value of the x and/or y components of `shaderReference.textureTransform.scale` is greater than 1, the texture is cropped as it extends past the texture coordinate range.

The default value of this property is TRUE. This property is always TRUE when using the #software renderer.

Example

The following statement sets the textureRepeat property of the first shader used by the model named gbCyl3 to TRUE. The first texture in that shader will tile if the value of the x or y component of its textureTransform or textureTransformList property is less than 1.

```
-- Lingo
member("scene").model("gbCyl3").shader.textureRepeat = TRUE

// Javascript
member("scene").getPropRef("model",1).getProp("shader").textureRepeat = true;
```

See also

textureTransform, textureTransformList

textureRepeatList

Usage

```
shaderReference.textureRepeatList[textureLayerIndex]
member(whichCastmember).shader(whichShader).textureRepeatList[textureLayerIndex]
member(whichCastmember).shader[shaderListIndex].textureRepeatList[textureLayerIndex]
member(whichCastmember).model(whichModel).shader.textureRepeatList[textureLayerIndex]
member(whichCastmember).model(whichModel).shaderList[shaderListIndex].
textureRepeatList[textureLayerIndex]
```

Description

3D standard shader property; allows you to get or set the texture clamping behavior of any texture layer. When TRUE, the default, the texture in textureLayerIndex can be tiled (repeated) several times across model surfaces. This can be accomplished by setting shaderReference.textureTransform[textureLayerIndex].scale to be less than 1 in x or y. When this value is set to FALSE, the texture will apply to a smaller portion of model surfaces, rather than tile across those surfaces, when the shaderReference.textureTransform[textureLayerIndex].scale is less than 1 in x or y. Think of it as shrinking the source image within the frame of the original image and filling in black around the gap. Similarly, if shaderReference.textureTransform[textureLayerIndex].scale is set to be greater than 1 in x or y, the image will be cropped as the border of the texture is extended past the texture coordinate range.

Example

The following code will textureMap a sphere entirely with a granite texture repeated 4 times across the surface, and a logo image which covers just 1/4 of the surface.

```
-- Lingo
m = member(2).model("mySphere")
f = member(2).newTexture("granite", #fromCastmember, member("granite"))
g = member(2).newTexture("logo", #fromCastmember, member("logo"))
s = member(2).newShader("s", #standard)
s.textureList[1] = g
s.textureList[2] = f
s.textureRepeatList[2] = false
s.textureRepeatList[1] = true
s.textureTransformList[1].scale(0.5,0.5,1.0)
s.textureTransformList[2].scale(0.5,0.5,1.0)
s.textureModeList[2] = #wrapPlanar
```

```
s.blendFunctionList[2] = #add
m.shaderList = s

// Javascript
m = member(2).getPropRef("model","1");
f = member(2).newTexture("granite", symbol("fromCastmember"), member("granite"));
g = member(2).newTexture("logo", symbol("fromCastmember"), member("logo"));
s = member(2).newShader("s", symbol("standard"));
s.textureList[1] = g;
s.textureList[2] = f;
s.textureRepeatList[2] = false;
s.textureRepeatList[1] = true;
s.textureTransformList[1].scale(0.5,0.5,1.0);
s.textureTransformList[2].scale(0.5,0.5,1.0);
s.textureModeList[2] = symbol("wrapPlanar");
s.blendFunctionList[2] = symbol("add");
m.shaderList = s;
```

textureTransform

Usage

```
member(whichCastmember).shader(whichShader).textureTransform
member(whichCastmember).model(whichModel).shader.textureTransform
member(whichCastmember).model(whichModel).shaderList{[index]}.textureTransform
```

Description

3D `#standard` shader property; provides access to a transform which modifies the texture coordinate mapping of the first texture layer of the shader. Manipulate this transform to tile, rotate, or translate the texture before applying it to the surface of the model. The texture itself remains unaffected; the transform merely modifies how the shader applies the texture. The `textureTransform` property is applied to all texture coordinates regardless of the `textureMode` property setting. This is the last modification of the texture coordinates before they are sent to the renderer. The `textureTransform` property is a matrix that operates on the texture in textureImage space. Texture-Image space is defined to exist only on the X,Y plane.

To tile the image twice along its horizontal axis, use `shaderReference.textureTransform.scale(0.5, 1.0, 1.0)`. Scaling on the Z axis is ignored.

To offset the image by `point(xOffset,yOffset)`, use `shaderReference.textureTransform.translate(xOffset,yOffset,0.0)`. Translating by integers when the shader's `textureRepeat` property is `TRUE` will have no effect, because the width and height of the texture will be valued between 0.0 and 1.0 in that case.

To apply a rotation to a texture layer, use `shaderReference.textureTransform.rotate(0,0,angle)`. Rotations around the Z axis are rotated around the (0,0) 2d image point, which maps to the upper left corner of the texture. Rotations about the X and Y axes are ignored.

Just as with a model's transform, `textureTransform` modifications are layerable. To rotate the texture about a point(xOffset,yOffset) instead of point(0,0), first translate to point(0 - xOffset, 0 - yOffset), then rotate, then translate to point(xOffset, yOffset). The `textureTransform` is similar to the shader's `wrapTransform` property with the following exceptions. It is applied in 2d image space rather than 3d world space. As a result, only rotations about the Z axis and translations and scales on X and Y axes are effective. The transform is applied regardless of the `shaderReference.textureMode` setting. The `wrapTransform`, by comparison, is only effective when the `textureMode` is `#wrapPlanar`, `#wrapCylindrical`, or `#wrapSpherical`.

Example

This statement shows the `textureTransform` of the first texture in the first shader used by the model gbCyl3:

```
-- Lingo
put member("Scene").model("gbCyl3").shader.textureTransform
-- transform(1.0000, 0.0000, 0.0000,0.0000, 0.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000,
1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000)

// Javascript
trace(member("Scene").getPropRef("model",1).getProp("shader").textureTransform);
// <transform(1.0000, 0.0000, 0.0000,0.0000, 0.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000,
1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000)>
```

The following statement halves the height and width of the first texture used by the shader named gbCyl3. If the `textureRepeat` property of gbCyl3 is set to `TRUE`, four copies of the texture will be tiled across the shader.

```
-- Lingo
member("Scene").shader("gbCyl3").textureTransform.scale = vector(0.5, 0.5, 1)

// Javascript
member("Scene").getPropRef("shader",1).textureTransform.scale = vector(0.5,0.5,1);
```

This statement rotates the first texture used by the shader gbCyl3 by 90° from vector(0, 0, 0):

```
-- Lingo
member("Scene").shader("gbCyl3").textureTransform.rotation = vector(0, 0, 90)

// Javascript
member("Scene").getPropRef("shader",1).textureTransform.rotation = vector(0,0,90);
```

textureTransformList

Usage

```
shaderReference .textureTransformList[textureLayerIndex]
member(whichCastmember).shader(ShaderName).textureTransformList[textureLayerIndex]
member(whichCastmember).shader[shaderListIndex].textureTransformList[textureLayerIndex]
member(whichCastmember).model(modelName).shader.textureTransformList[textureLayerIndex]
member(whichCastmember).model(modelName).shaderList[shaderListIndex].
textureTransformList[textureLayerIndex]
```

Description

3D standard shader property; this property provides access to a transform which modifies the texture coordinate mapping of a texture layer. Manipulate this transform to tile, rotate, or translate a texture image before applying it to the surface of models. The texture itself remains unaffected, the transform merely modifies how the shader applies the texture.

To tile the image twice along its horizontal axis, use `textureTransformList[whichTextureLayer].scale(0.5, 1.0, 1.0)`. Scales in Z will be ignored since images are 2D in nature. Care must be taken to avoid 0.0 scales (even in Z), as that will negate the effect of the entire texture.

To offset the image by point(xOffset,yOffset), use `textureTransformList[whichTextureLayer].translate(xOffset,yOffset,0.0)`. Translating by integers when that texture layer's textureRepeat property is `TRUE` will have no effect, because the width and height of the texture will be valued between 0.0 and 1.0 in that case.

To apply a rotation to a texture layer, use `textureTransformList[whichTextureLayer].rotate(0,0,angle)`. Rotations around the Z axis are rotated around the (0,0) 2D image point, which maps to the upper left corner of the texture. Rotations about X and Y will be ignored since images are 2D by nature.

Just as with a model's transform, `textureTransform` modifications are layerable. To rotate the image about a point(xOffset,yOffset) instead of point(0,0), first translate to point(0 - xOffset, 0 - yOffset), then rotate, then translate to point(xOffset, yOffset).

The `textureTransformList` is similar to the shader `wrapTransformList` property with the following exceptions.

It is applied in 2D image space rather than 3D world space. As a result, only rotations in Z, and translations and scales in X and Y, are effective.

The transform is applied regardless of the `shaderReference.textureModeList[index]` setting. The `wrapTransform`, by comparison, is only effective when the `textureMode` is `#wrapPlanar`, `#wrapCylindrical`, or `#wrapSpherical`.

Example

This statement shows the textureTransform of the third texture in the first shader used by the model gbCyl3:

```
-- Lingo
put member("scene").model("gbCyl3").shader.textureTransformList[3]
-- transform(1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000,
1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000)

// Javascript
trace(member("scene").getPropRef("model",1).getProp("shader").textureTransformList[3])
//< transform(1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 0.0000, 0.0000, 0.0000,
0.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000)>
```

The following statement halves the height and width of the fifth texture used by the shader gbCyl3. If the `textureRepeatList[5]` value of gbCyl3 is set to `TRUE`, four copies of the texture will be tiled across the shader.

```
-- Lingo
member("scene").shader("gbCyl3").textureTransformList[5].scale = vector(0.5, 0.5, 1)

// Javascript
member("scene").getPropRef("shader","1").textureTransformList[5].scale = vector(0.5, 0.5,
1);
```

This statement rotates the fourth texture used by the shader gbCyl3 by 90° from vector(0, 0, 0):

```
-- Lingo
member("scene").shader("gbCyl3").textureTransformList[4].rotation = vector(0, 0, 90)

// Javascript
member("scene").getPropRef("shader","1 ").textureTransformList[4].rotation = vector(0, 0,
90);
```

These statements rotate the third texture used by the shader gbCyl3 by 90× around its center, assuming that textureList[3] is a 128x128 sized texture:

```
-- Lingo
s = member("scene").shader("gbCyl3")
s.textureTransformList[3].translate(-64,-64,0)
s.textureTransformList[3].rotate(0,0,90)
s.textureTransformList[3].translate(64,64,0)

// Javascript
s = member("scene").getPropRef("shader","1");
```

```
s.textureTransformList[3].translate(-64,-64,0);
s.textureTransformList[3].rotate(0,0,90);
s.textureTransformList[3].translate(64,64,0);
```

textureType

Usage

```
member(whichCastmember).textureType
```

Description

3D texture property; allows you to get or set the texture type for the default texture. Possible values are as follows:

- `#none` specifies that there is no texture type.
- `#default` uses the texture from the original shader as the texture.
- `#member` uses the image from the specified cast member as the texture.

The default value for this property is `#default`. You must specify `#member` for this property in order to use the `textureMember` property.

Example

The following statement sets the `textureType` property of the cast member Scene to `#member`.

```
member("Scene").textureType = #member
```

This makes it possible use a bitmap cast member as the source of the default texture by setting the `textureMember` property. The bitmap cast member is named "grass".

```
member("Scene").textureMember = "grass"
```

See also

`textureMember`

thumbNail

Usage

```
-- Lingo syntax
memberObjRef.thumbNail

// JavaScript syntax
memberObjRef.thumbNail;
```

Description

Member property; contains the image used to preview a cast member in the Cast window. Read/write during authoring only.

The image can be customized for any cast member.

Example

The following statement shows how to use a placeholder cast member to display another thumbnail on the Stage. The placeholder cast member is placed on the Stage, then the picture of that member is set to the thumbnail of member 10. This makes it possible to show a reduced image without having to scale or otherwise manipulate a graphic:

```
-- Lingo syntax
member("Placeholder").picture = member(10).thumbNail

// JavaScript syntax
member("Placeholder").picture = member(10).thumbNail;
```

See also

Member

tilt

Usage

```
-- Lingo syntax
spriteObjRef.tilt

// JavaScript syntax
spriteObjRef.tilt;
```

Description

QuickTime VR sprite property; the current tilt, in degrees, of the QuickTime VR movie.

This property can be tested and set.

Example

This statement shows how to get the tilt of the current QuickTime movie which is in sprite(3).

```
-- Lingo
putsprite(3).tilt

// Javascript
trace(sprite(3).tilt);
```

time (timeout object)

Usage

```
timeoutObject.time
```

Description

Timeout object property; the system time, in milliseconds, when the next timeout event will be sent by the given *timeoutObject*.

This is not the time until the next event, but the absolute time of the next timeout event.

Example

This handler determines the time remaining until the next `timeout` event will be sent by the timeout object `Update` by calculating the difference between its `time` property and the current value of the milliseconds and displaying the result in the field Time Until:

```
-- Lingo
on prepareFrame
    msBeforeUpdate = timeout("Update").time - the milliseconds
    secondsBeforeUpdate = msBeforeUpdate / 1000
    minutesBeforeUpdate = secondsBeforeUpdate / 60
    member("Time Until").text = string(minutesBeforeUpdate) && "minutes before next \
    timeout"
end

// Javascript
Function prepareFrame()
{
    var msBeforeUpdate = timeout(“Update”).time - _system.milliseconds;
    var secondsBeforeUpdate = msBeforeUpdate / 1000;
    var minutesBeforeUpdate = secondsBeforeUpdate / 60;
    member(“Time Until”).text = string(minutesBeforeUpdate) + “ minutes before next
timeout”;
}
```

See also

`milliseconds`, `period`, `persistent`, `target`, `timeout()`, `timeoutHandler`

timeoutHandler

Usage

`timeoutObject.timeoutHandler`

Description

System property; represents the name of the handler that will receive timeout messages from the given *`timeoutObject`*. Its value is a symbol, such as `#timeExpiredHandler`. The `timeoutHandler` is always a handler within the timeout object's `target` object, or in a movie script if the timeout object has no `target` specified.

This property can be tested and set.

Example

This statement displays the `timeoutHandler` of the timeout object Quiz Timer in the Message window:

```
-- Lingo
put timeout("Quiz Timer").timeoutHandler

// Javascript
trace(timeout(“Quiz Timer”).timeoutHandler);
```

See also

`target`, `timeout()`, `timeoutList`

timeoutList

Usage

```
-- Lingo syntax
_movie.timeoutList

// JavaScript syntax
_movie.timeoutList;
```

Description

Movie property; a linear list containing all currently active timeout objects. Read-only.

Use the `forget()` method to delete a timeout object.

Timeout objects are added to the `timeoutList` with the `new()` method.

Example

This statement deletes the third timeout object from the timeout list:

```
-- Lingo syntax
_movie.timeoutList[3].forget()

// JavaScript syntax
_movie.timeoutList[3].forget();
```

See also

`forget() (Window)`, `Movie`, `new()`, `forget() (Timeout)`, `timeout()`

timeScale

Usage

```
member(whichCastMember).timeScale
the timeScale of member whichCastMember
```

Description

Cast member property; returns the time unit per second on which the digital video's frames are based. For example, a time unit in a QuickTime digital video is 1/600 of a second.

This property can be tested but not set.

Example

This statement displays the timescale of a QuickTime digial video in sprite(3).

```
-- Lingo
put(sprite(3).timeScale)

// Javascript
trace(sprite(3).timeScale)
```

See also

`digitalVideoTimeScale`

title (DVD)

Usage

```
-- Lingo syntax
dvdObjRef.title

// JavaScript syntax
dvdObjRef.title;
```

Description

DVD property; specifies the current title. Read/write.

This property returns an integer that specifies the number of the current title.

Example

This statement returns the current title:

```
-- Lingo syntax
trace (member(1).title)-- 1

// JavaScript syntax
trace (member(1).title);// 1
```

See also

DVD

title (Window)

Usage

```
-- Lingo syntax
windowObjRef.title

// JavaScript syntax
windowObjRef.title;
```

Description

Window property; assigns a title to a window. Read/write.

Example

This statements assigns the title Planets to the fifth window:

```
-- Lingo syntax
_player.windowList[5].title = "Planets"

// JavaScript syntax
_player.windowList[5].title = "Planets";
```

See also

Window

titlebarOptions

Usage

```
-- Lingo syntax
windowObjRef.titlebarOptions

// JavaScript syntax
windowObjRef.titlebarOptions;
```

Description

Window property; specifies a list of properties that stores the title bar options of a window. Read/write.

The property list contains the following properties:

Property	Description
#icon	Specifies the cast member icon to use in the title bar. This property is only available if the title bar is visible (the `#visible` property is set to `TRUE`).
#visible	Specifies whether the title bar is visible. If `FALSE`, the title bar is not visible, and all other title bar and window properties are unaffected. If `TRUE`, the title bar is visible, and the window maintains the states of all other title bar and window properties. The default value is `TRUE`.
#closebox	Specifies whether a close box appears in the upper right corner of the window. If `TRUE`, a close box appears. If `FALSE`, a close box does not appear. The default value is `TRUE`.
#minimizebox	Specifies whether a minimize box appears in the upper right corner of the window. If `TRUE`, a minimize box appears. If `FALSE`, a minimize box does not appear. The default value is `TRUE`.
#maximizebox	Specifies whether a maximize box appears in the upper right corner of the window. If `TRUE`, a maximize box appears. If `FALSE`, a maximize box does not appear. The default value is `TRUE`.
#sideTitlebar	(Mac only) Specifies whether the title bar should appear on the side of the winow. If `TRUE`, the title bar appears on the side of the window. If `FALSE`, the title bar does not appear on the side of the window. The default value is `FALSE`.

These properties can also be accessed by using the Movie object's `displayTemplate` property.

Example

This statement displays in the Message window the available titlebar options for the window named Elements:

```
-- Lingo syntax
trace(window("Elements").titlebarOptions)

// JavaScript syntax
trace(window("Elements").titlebarOptions);
```

These statements set the `icon` property to the bitmap cast member named `smallIcon`:

```
-- Lingo syntax
window("Elements").titlebarOptions.icon = member("smallIcon")

// JavaScript syntax
window("Elements").titlebarOptions.icon = member("smallIcon");
```

See also

`appearanceOptions`, `displayTemplate`, `Window`

titleCount

Usage

```
-- Lingo syntax
dvdObjRef.titleCount

// JavaScript syntax
dvdObjRef.titleCount;
```

Description

DVD property; returns the number of available titles. Read-only.

The number of available titles ranges from 1 to 99.

Example

This statement returns the title count of the dvd object.

```
-- Lingo
put sprite(1).titleCount

// Javascript
trace(sprite(1).titleCount)
```

See also

DVD

toolXtraList

Usage

```
-- Lingo syntax
_player.toolXtraList

// JavaScript syntax
_player.toolXtraList;
```

Description

Player property; returns a linear list of all tool Xtra extensions available to the Director player. Read-only.

Example

This statement displays in the Message window all available tool Xtra extensions.

```
-- Lingo syntax
put(_player.toolXtraList)

// JavaScript syntax
put(_player.toolXtraList);
```

See also

mediaXtraList, Player, scriptingXtraList, transitionXtraList, xtraList (Player)

toon (modifier)

Usage

```
member(whichCastmember).model(whichModel).toon.toonModifierProperty
```

Description

3D modifier; once you have added the `#toon` modifier to a model you can get and set the `#toon` modifier properties.

The toon modifier draws a model using only a handful of colors, and resulting in a cartoon style of rendering of the model's surface. When the `#toon` modifier is applied, the model's shader `texture`, `reflectionMap`, `diffuseLightMap`, `specularLightMap`, and `glossMap` properties are ignored.

When the `#toon` modifier is used in conjunction with the `#inker` modifier, the rendered effect is cumulative and varies depending on which modifier was first applied. The list of modifiers returned by the `modifier` property will list #inker or #toon (whichever was added first), but not both. The toon modifier can not be used in conjunction with the `#sds` modifier.

The `#toon` modifier has the following properties:

Note: *For more detailed information about the following properties see the individual property entries.*

- `style` allows you to get or set the style applied to color transitions. The following are the possible values:

  `#toon` gives sharp transitions between available colors.

  `#gradient` gives smooth transitions between available colors.

  `#blackAndWhite` gives sharp transition between black and white.

- `colorSteps` allows you to get or set the number of different colors used for lighting calculations. When setting this value it is rounded down to nearest power of 2. Allowed values are 2, 4, 8, and 16. The default is 2.

- `shadowPercentage` allows you to get or set the percentage of the colors (`colorSteps`) defined for lighting used to render the shadowed portion of the model's surface. Possible values range from 0 to 100. The default is 50.

- `shadowStrength` allows you to get or set the level of darkness applied to the shadowed portion of the model's surface. Possible values are any non-negative floating-point number. The default value is 1.0.

- `highlightPercentage` allows you to get or set the percentage of the colors defined for lighting (`colorSteps`) used to render the highlighted portion of the model's surface. Possible values range from 0 to 100. The default is 50.

- `highlightStrength` allows you to get or set the level of brightness applied to the highlighted portion of the model's surface. Possible values are any non-negative floating-point number. The default value is 1.0.

- `lineColor` allows you to get or set the color of lines drawn by the inker. Possible values are any valid Lingo color object. The default value is `rgb (0, 0, 0)`, which is black.

- `creases` allows you to get or set whether lines are drawn in creases. This is a Boolean value; the default value is `True`.

- `creaseAngle`, if `creases` is set to `TRUE`, allows you to get or set how sensitive the line drawing function of the toon modifier is to the presence of creases.

- `boundary` allows you to get or set whether lines are drawn around the boundary of the surface. This is a Boolean value; the default value is `True`.

- `lineOffset` allows you to get or set where lines are drawn relative to the shaded surface and the camera. Negative lines move lines toward the camera. Positive values move lines away from the camera. Possible values are floating-point numbers from -100.0 to 100.0. The default value is -2.0.
- `useLineOffset` allows you to get or set whether `lineOffset` is on or off. This is a Boolean value; the default value is `False`.
- `silhouettes` allows you to get or set whether lines are drawn to define the edges along the border of a model, outlining its shape. This is a Boolean value; the default value is `True`.

See also

`addModifier`, `modifiers`, `sds (modifier)`, `inker (modifier)`

top

Usage

```
-- Lingo syntax
spriteObjRef.top

// JavaScript syntax
spriteObjRef.top;
```

Description

Sprite property; returns or sets the top vertical coordinate of the bounding rectangle of a sprite as the number of pixels from the upper left corner of the Stage. Read/write.

Example

This statement checks whether the top of sprite 3 is above the top of the Stage and calls the handler `offtopedge` if it is:

```
-- Lingo syntax
if (sprite(3).top < 0) then
    offTopEdge()
end if

// JavaScript syntax
if (sprite(3).top < 0) {
    offTopEdge();
}
```

See also

`bottom`, `height`, `left`, `locH`, `locV`, `right`, `Sprite`, `width`

topSpacing

Usage

```
chunkExpression.topSpacing
```

Description

Text cast member property; allows you to specify additional spacing applied to the top of each paragraph in the *`chunkExpression`* portion of the text cast member.

The value itself is an integer, with less than 0 indicating less spacing between paragraphs and greater than 0 indicating more spacing between paragraphs.

The default value is 0, which results in default spacing between paragraphs.

Example

This statement sets the `topSpacing` of the second paragraph in text cast member "myText" to 20:

```
-- Lingo
member(1).paragraph[2].topSpacing = 20

// Javascript
member(1).getPropRef("paragraph",2).topSpacing = 20;
```

See also

bottomSpacing

traceLoad

Usage

```
-- Lingo syntax
_movie.traceLoad

// JavaScript syntax
_movie.traceLoad;
```

Description

Movie property; specifies the amount of information that is displayed about cast members as they load. Read/write.

Valid values for traceLoad are as follows.

- 0—Displays no information (default).
- 1—Displays cast members' names.
- 2—Displays cast members' names, the number of the current frame, the movie name, and the file seek offset (the relative amount the drive had to move to load the media).

Example

This statement causes the movie to display the names of cast members as they are loaded:

```
-- Lingo syntax
_movie.traceLoad = 1

// JavaScript syntax
_movie.traceLoad = 1;
```

See also

Movie

traceLogFile

Usage

```
-- Lingo syntax
_movie.traceLogFile

// JavaScript syntax
_movie.traceLogFile;
```

Description

Movie property; specifies the name of the file in which the Message window display is written. Read/write.

You can close the file by setting the `traceLogFile` property to `EMPTY` (Lingo) or an empty string " " (JavaScript syntax). Any output that would appear in the Message window is written into this file. You can use this property for debugging when running a movie in a projector and when authoring.

Example

This statement instructs Lingo to write the contents of the Message window in the file "Messages.txt" in the same folder as the current movie:

```
-- Lingo syntax
_movie.traceLogFile = _movie.path & "Messages.txt"

// JavaScript syntax
_movie.traceLogFile = _movie.path + "Messages.txt";
```

This statement closes the file that the Message window display is being written to:

```
-- Lingo syntax
_movie.traceLogFile = ""

// JavaScript syntax
_movie.traceLogFile = "";
```

See also

Movie

traceScript

Usage

```
-- Lingo syntax
_movie.traceScript

// JavaScript syntax
_movie.traceScript;
```

Description

Movie property; specifies whether the movie's trace function is on (`TRUE`) or off (`FALSE`).
Read/write.

When `traceScript` is on, the Message window displays each line of script that is being executed.

Example

This statement turns the `traceScript` property on.

```
-- Lingo syntax
_movie.traceScript = TRUE

// JavaScript syntax
_movie.traceScript = true;
```

See also

Movie

trackCount (Member)

Usage

```
-- Lingo syntax
memberObjRef.trackCount()

// JavaScript syntax
memberObjRef.trackCount();
```

Description

Digital video cast member property; returns the number of tracks in the specified digital video cast member.

This property can be tested but not set.

Example

This statement determines the number of tracks in the digital video cast member Jazz Chronicle and displays the result in the Message window:

```
-- Lingo syntax
put(member("Jazz Chronicle").trackCount())

// JavaScript syntax
trace(member("Jazz Chronicle").trackCount());
```

trackCount (Sprite)

Usage

```
-- Lingo syntax
spriteObjRef.trackCount()

// JavaScript syntax
spriteObjRef.trackCount();
```

Description

Digital video sprite property; returns the number of tracks in the specified digital video sprite.

This property can be tested but not set.

Example

This statement determines the number of tracks in the digital video sprite assigned to channel 10 and displays the result in the Message window:

```
-- Lingo syntax
```

```
put(sprite(10).trackCount())

// JavaScript syntax
trace(sprite(10).trackCount());
```

trackEnabled

Usage

```
-- Lingo syntax
spriteObjRef.trackEnabled(whichTrack)

// JavaScript syntax
spriteObjRef.trackEnabled(whichTrack);
```

Description

Digital video sprite property; indicates the status of the specified track of a digital video. This property is `TRUE` if the track is enabled and playing. This property is `FALSE` if the track is disabled and no longer playing or is not updating.

This property cannot be set. Use the `setTrackEnabled` property instead.

Example

This statement checks whether track 2 of digital video sprite 1 is enabled:

```
-- Lingo syntax
put(sprite(1).trackEnabled(2))

// JavaScript syntax
put(sprite(1).trackEnabled(2));
```

See also

`setTrackEnabled()`

trackNextKeyTime

Usage

```
-- Lingo syntax
spriteObjRef.trackNextKeyTime(whichTrack)

// JavaScript syntax
spriteObjRef.trackNextKeyTime(whichTrack);
```

Description

Digital video sprite property; indicates the time of the keyframe that follows the current time in the specified digital video track.

This property can be tested but not set.

Example

This statement determines the time of the keyframe that follows the current time in track 5 of the digital video assigned to sprite channel 15 and displays the result in the Message window:

```
-- Lingo syntax
```

```
put(sprite(15).trackNextKeyTime(5))

// JavaScript syntax
put(sprite(15).trackNextKeyTime(5));
```

trackNextSampleTime

Usage

```
-- Lingo syntax
spriteObjRef.trackNextSampleTime(whichTrack)

// JavaScript syntax
spriteObjRef.trackNextSampleTime(whichTrack);
```

Description

Digital video sprite property; indicates the time of the next sample that follows the digital video's current time. This property is useful for locating text tracks in a digital video.

This property can be tested but not set.

Example

This statement determines the time of the next sample that follows the current time in track 5 of the digital video assigned to sprite 15:

```
-- Lingo syntax
put(sprite(15).trackNextSampleTime(5))

// JavaScript syntax
put(sprite(15).trackNextSampleTime(5));
```

trackPreviousKeyTime

Usage

```
-- Lingo syntax
spriteObjRef.trackPreviousKeyTime(whichTrack)

// JavaScript syntax
spriteObjRef.trackPreviousKeyTime(whichTrack);
```

Description

Digital video sprite property; reports the time of the keyframe that precedes the current time.

This property can be tested but not set.

Example

This statement determines the time of the keyframe in track 5 that precedes the current time in the digital video sprite assigned to channel 15 and displays the result in the Message window:

```
-- Lingo syntax
put(sprite(15).trackPreviousKeyTime(5))

// JavaScript syntax
put(sprite(15).trackPreviousKeyTime(5));
```

trackPreviousSampleTime

Usage

```
-- Lingo syntax
spriteObjRef.trackPreviousSampleTime(whichTrack)

// JavaScript syntax
spriteObjRef.trackPreviousSampleTime(whichTrack);
```

Description

Digital video sprite property; indicates the time of the sample preceding the digital video's current time. This property is useful for locating text tracks in a digital video.

This property can be tested but not set.

Example

This statement determines the time of the sample in track 5 that precedes the current time in the digital video sprite assigned to channel 15 and displays the result in the Message window:

```
-- Lingo syntax
put(sprite(15).trackPreviousSampleTime(5))

// JavaScript syntax
put(sprite(15).trackPreviousSampleTime(5));
```

trackStartTime (Member)

Usage

```
-- Lingo syntax
memberObjRef.trackStartTime(whichTrack)

// JavaScript syntax
memberObjRef.trackStartTime(whichTrack);
```

Description

Digital video cast member property; returns the start time of the specified track of the specified digital video cast member.

This property can be tested but not set.

Example

This statement determines the start time of track 5 in the digital video cast member Jazz Chronicle and displays the result in the Message window:

```
-- Lingo syntax
put(member("Jazz Chronicle").trackStartTime(5))

// JavaScript syntax
put(member("Jazz Chronicle").trackStartTime(5));
```

trackStartTime (Sprite)

Usage

```
-- Lingo syntax
spriteObjRef.trackStartTime(whichTrack)

// JavaScript syntax
spriteObjRef.trackStartTime(whichTrack);
```

Description

Digital video sprite property; sets the starting time of a digital video movie in the specified sprite channel. The value of `trackStartTime` is measured in ticks.

This property can be tested but not set.

Example

In the Message window, the following statement reports when track 5 in sprite channel 10 starts playing. The starting time is 120 ticks (2 seconds) into the track.

```
-- Lingo syntax
put(sprite(10).trackStartTime(5))

// JavaScript syntax
put(sprite(10).trackStartTime(5))
```

See also

duration (Member), playRate, currentTime (QuickTime, AVI)

trackStopTime (Member)

Usage

```
-- Lingo syntax
memberObjRef.trackStopTime(whichTrack)

// JavaScript syntax
memberObjRef.trackStopTime(whichTrack);
```

Description

Digital video cast member property; returns the stop time of the specified track of the specified digital video cast member. It can be tested but not set.

Example

This statement determines the stop time of track 5 in the digital video cast member Jazz Chronicle and displays the result in the Message window:

```
-- Lingo syntax
put(member("Jazz Chronicle").trackStopTime(5))

// JavaScript syntax
put(member("Jazz Chronicle").trackStopTime(5));
```

trackStopTime (Sprite)

Usage

```
-- Lingo syntax
spriteObjRef.trackStopTime(whichTrack)

// JavaScript syntax
spriteObjRef.trackStopTime(whichTrack);
```

Description

Digital video sprite property; returns the stop time of the specified track of the specified digital video sprite.

When a digital video movie is played, `trackStopTime` is when playback halts or loops if the `loop` property is turned on.

This property can be tested but not set.

Example

This statement determines the stop time of track 5 in the digital video assigned to sprite 6 and displays the result in the Message window:

```
-- Lingo syntax
put(sprite(6).trackStopTime(5))

// JavaScript syntax
put(sprite(6).trackStopTime(5));
```

See also

playRate, currentTime (QuickTime, AVI), trackStartTime (Member)

trackText

Usage

```
-- Lingo syntax
spriteObjRef.trackText(whichTrack)

// JavaScript syntax
spriteObjRef.trackText(whichTrack);
```

Description

Digital video sprite property; provides the text that is in the specified track of the digital video at the current time. The result is a string value, which can be up to 32K characters long. This property applies to text tracks only.

This property can be tested but not set.

Example

This statement assigns the text in track 5 of the digital video assigned at the current time to sprite 20 to the field cast member Archives:

```
-- Lingo syntax
member("Archives").text = string(sprite(20).trackText(5))

// JavaScript syntax
member("Archives").text = sprite(20).trackText(5).toString();
```

trackType (Member)

Usage

```
-- Lingo syntax
memberObjRef.trackType(whichTrack)

// JavaScript syntax
memberObjRef.trackType(whichTrack);
```

Description

Digital video cast member property; indicates which type of media is in the specified track of the specified cast member. Possible values are `#video`, `#sound`, `#text`, and `#music`.

This property can be tested but not set.

Example

The following handler checks whether track 5 of the digital video cast member Today's News is a text track and then runs the handler `textFormat` if it is:

```
-- Lingo syntax
on checkForText
    if member("Today's News").trackType(5) = #text then
        textFormat
    end if
end

// JavaScript syntax
function checkForText() {
    var tt = member("Today's News").trackType(5);
    if (tt == "text") {
        textFormat();
    }
}
```

trackType (Sprite)

Usage

```
-- Lingo syntax
spriteObjRef.trackType(whichTrack)

// JavaScript syntax
spriteObjRef.trackType(whichTrack);
```

Description

Digital video sprite property; returns the type of media in the specified track of the specified sprite. Possible values are `#video`, `#sound`, `#text`, and `#music`.

This property can be tested but not set.

Example

The following handler checks whether track 5 of the digital video sprite assigned to channel 10 is a text track and runs the handler `textFormat` if it is:

```
-- Lingo syntax
```

```
on checkForText
    if sprite(10).trackType(5) = #text then
        textFormat
    end if
end

// JavaScript syntax
function checkForText() {
    var tt = sprite(10).trackType(5);
    if (tt == "text") {
        textFormat();
    }
}
```

trails

Usage

```
sprite(whichSprite).trails
the trails of sprite whichSprite
```

Description

Sprite property; for the sprite specified by *whichSprite*, turns the trails ink effect on (1 or TRUE) or off (0 or FALSE). For the value set by Lingo to last beyond the current sprite, the sprite must be a scripted sprite.

To erase trails, animate another sprite across these pixels or use a transition.

Example

This statement turns on trails for sprite 7:

```
-- Lingo
sprite(7).trails = 1

// Javascript
sprite(7).trails = 1;
```

See also

directToStage

transform (property)

Usage

```
member(whichCastmember).node(whichNode).transform
member(whichCastmember).node(whichNode).transform.transformProperty
member(whichCastmember).model(whichModel).bonesPlayer.bone[boneID].transform
member(whichCastmember).model(whichModel).bonesPlayer.bone[boneID].transform.transformProp
erty
```

Description

3D property and command; allows you to get or set the transform associated with a particular node or a specific bone within a model using the bonesPlayer modifier. As a command, transform provides access to the various commands and properties of the transform object. A node can be a camera, group, light or model object.

For node objects, this property defaults to the identity transform. A node's transform defines the position, rotation and scale of the node relative to is parent object. If a node's parent is the World group object, then the `transform` property of the node has the same value as is returned by the `getWorldTransform()` command.

For bones within models using the `bonesPlayer` modifier, this property defaults in value to the transform assigned to the bone upon creation of the model file. The transform of a bone represents the bone's rotation relative to its parent bone and its position relative to its original joint position. The original joint position is determine upon creation of the model file.

You can use the following transform commands and properties with the `transform` property of node objects:

Note: *This section only contains summaries, see the individual entries for more detailed information.*

- `preScale` applies scaling before the current positional, rotational, and scale offsets held by the transform.
- `preTranslate` applies a translation before the current positional, rotational, and scale offsets held by the transform.
- `preRotate` applies a rotation before the current positional, rotational, and scale offsets held by the transform.
- `scale` (command) applies scaling after the current positional, rotational, and scale offsets held by the transform.
- `scale` (transform) allows you to get or set the degree of scaling of the transform.
- `translate` applies a translation after the current positional, rotational, and scale offsets held by the transform.
- `rotate` applies a rotation after the current positional, rotational, and scale offsets held by the transform.
- `position` (transform) allows you to get or set the positional offset of the transform.
- `rotation` (transform) allows you to get or set the rotational offset of the transform.

If you want to modify the `transform` property of a bone within a model, then you must store a copy of the original transform of the bone, modify the stored copy using the above commands and properties, then reset the bone's `transform` property so that it is equal to the modified transform. For example:

```
t = member("character").model("biped").bonesPlayer.bone[38].transform.duplicate()
t.translate(25,0,-3)
member("character").model("biped").bonesPlayer.bone[38].transform = t
```

Parameters

None.

Example

This Lingo shows the transform of the model box, followed by the position and rotation properties of the transform:

```
put member("3d world").model("box").transform
-- transform(1.000000,0.000000,0.000000,0.000000, 0.000000,1.000000,0.000000,0.000000,
0.000000,0.000000,1.000000,0.000000, -94.144844,119.012825,0.000000,1.000000)
put member("3d world").model("box").transform.position
-- vector(-94.1448, 119.0128, 0.0000)
put member("3d world").model("box").transform.rotation
--vector(0.0000, 0.0000, 0.0000)
```

See also

`interpolateTo()`, `scale (transform)`, `rotation (transform)`, `position (transform)`, `bone`, `worldTransform`, `preRotate`, `preScale()`, `preTranslate()`

transitionType

Usage

```
member(whichCastMember).transitionType
the transitionType of member whichCastMember
```

Description

Transition cast member property; determines a transition's type, which is specified as a number. The possible values are the same as the codes assigned to transitions for the `puppetTransition` command.

Example

This statement sets the type of transition cast member 3 to 51, which is a pixel dissolve cast member:

```
member(3).transitionType = 51
```

transitionXtraList

Usage

```
-- Lingo syntax
_player.transitionXtraList

// JavaScript syntax
_player.transitionXtraList;
```

Description

Player property; returns a linear list of all transition Xtra extensions available to the Director player. Read-only.

Example

This statement displays in the Message window all available transition Xtra extensions.

```
-- Lingo syntax
put(_player.transitionXtraList)

// JavaScript syntax
put(_player.transitionXtraList);
```

See also

mediaXtraList, Player, scriptingXtraList, toolXtraList, xtraList (Player)

translation

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.translation

// JavaScript syntax
memberOrSpriteObjRef.translation;
```

Description

QuickTime cast member and sprite property; controls the offset of a QuickTime sprite's image within the sprite's bounding box.

This offset is expressed in relation to the sprite's default location as set by its `center` property. When `center` is set to `TRUE`, the sprite is offset relative to the center of the bounding rectangle; when `center` is set to `FALSE`, the sprite is offset relative to the upper left corner of the bounding rectangle.

The offset, specified in pixels as positive or negative integers, is set as a Director list: [*xTrans*, *yTrans*]. The *xTrans* parameter specifies the horizontal offset from the sprite's default location; the *yTrans* parameter specifies the vertical offset. The default setting is [0,0].

When the sprite's `crop` property is set to `TRUE`, the `translation` property can be used to mask portions of the QuickTime movie by moving them outside the bounding rectangle. When the `crop` property is set to `FALSE`, the `translation` property is ignored, and the sprite is always positioned at the upper left corner of the sprite's rectangle.

This property can be tested and set.

Example

The following frame script assumes that the `center` property of the cast member of a 320-pixel-wide QuickTime sprite in channel 5 is set to `FALSE`, and its `crop` property is set to `TRUE`. It keeps the playhead in the current frame until the movie's horizontal translation point has moved to the right edge of the sprite, in 10-pixel increments. This has a wipe right effect, moving the sprite out of view to the right. When the sprite is out of view, the playhead continues to the next frame.

```
-- Lingo syntax
on exitFrame
    horizontalPosition = sprite(5).translation[1]
    if horizontalPosition < 320 then
        sprite(5).translation = sprite(5).translation + [10, 0]
        _movie.go(_movie.frame)
    end if
end

// JavaScript syntax
function exitFrame() {
    var horizontalPosition = sprite(5).translation[1];
    if (horizontalPosition < 320 ) {
        sprite(5).translation = sprite(5).translation + list(10, 0);
        _movie.go(_movie.frame);
    }
}
```

transparent

Usage

```
member(whichCastmember).shader(whichShader).transparent
member(whichCastmember).model(whichModel).shader.transparent
member(whichCastmember).model(whichModel).shaderList[shaderListIndex].transparent
```

Description

3D standard shader property; lets you get or set whether a model is blended using alpha values (`TRUE`) or is rendered as opaque (`FALSE`). The default value for this property is `TRUE` (alpha-blended).

The functionality of `shader.blend` is dependent upon this property.

All shaders have access to the `#standard` shader properties; in addition to these standard shader properties shaders of the types `#engraver`, `#newsprint`, and `#painter` have properties unique to their type. For more information, see the `newShader`.

Example

The following statement causes the model Pluto to be rendered opaque. The setting of the `blend` property for the model's shader will have no effect.

```
-- Lingo
member("scene").model("Pluto").shader.transparent = FALSE

// Javascript
member("scene").getPropRef("model",1).getProp("shader").transparent = false;
```

See also

`blendFactor`, `blend (3D)`

triggerCallback

Usage

```
-- Lingo syntax
spriteObjRef.triggerCallback

// JavaScript syntax
spriteObjRef.triggerCallback;
```

Description

QuickTime VR sprite property; contains the name of the handler that runs when the user clicks a hotspot in a QuickTime VR movie. The handler is sent two arguments: the `me` parameter and the ID of the hotspot that the user clicked.

The value that the handler returns determines how the movie processes the hotspot. If the handler returns `#continue`, the QuickTime VR sprite continues to process the hotspot normally. If the handler returns `#cancel`, the default behavior for the hotspot is canceled.

Set this property to 0 to clear the callback.

The QuickTime VR sprite receives the message first.

To avoid a decrease in performance, set the `triggerCallback` property only when necessary.

This property can be tested and set.

Example

The following statement sets the callback handler for a QuickTime VR sprite to the handler named `MyHotSpot-Callback` when the playhead first enters the sprite span. Every time that hotspot is triggered, the `MyHotSpot-Callback` handler is executed. When the playhead leaves the sprite span, the callback is canceled.

```
-- Lingo syntax
property pMySpriteNum, spriteNum

on beginSprite(me)
    pMySpriteNum = spriteNum
```

```
    sprite(pMySpriteNum).triggerCallback = #MyHotSpotCallback
end

on MyHotSpotCallback(me, hotSpotID)
    put "Hotspot" && hotSpotID && "was just triggered"
end

on endSprite me
    sprite(pMySpriteNum).triggerCallback = 0
end

// JavaScript syntax
function beginSprite() {
    pMySpriteNum = this.spriteNum;
    sprite(this.pMySpriteNum).triggerCallback = symbol("MyHotSpotCallback");
}

function MyHotSpotCallback(hotSpotID) {
    trace("Hotspot " + hotSpotID + " was just triggered");
}

function endSprite() {
    sprite(pMySpriteNum).triggerCallback = 0;
}
```

trimWhiteSpace

Usage

```
-- Lingo syntax
memberObjRef.trimWhiteSpace

// JavaScript syntax
memberObjRef.trimWhiteSpace;
```

Description

Cast member property. Determines whether the white pixels around the edge of a bitmap cast member are removed or left in place. This property is set when the member is imported. It can be changed in Lingo or in the Bitmap tab of the Property inspector.

Example

The following statement causes the white spaces to be trimmed in the text member.

```
-- Lingo
member(1).trimWhiteSpace;

// Javascript
member(1).trimWhiteSpace;
```

tunnelDepth

Usage

```
member(whichTextmember).tunnelDepth
member(whichCastMember).modelResource(whichExtruderModelResource).tunnelDepth
```

Description

A 3D extruder model resource property, as well as a text cast member property. Using this property allows you to get or set the extrusion depth (the distance between the front and back faces) of a 3D text model resource Possible values are floating point numbers between 1.0 and 100.0. The default value is 50.0.

It is recommended that you see `extrudeToMember` entry for more information about working with extruder model resources and text cast members.

Example

In this example, the cast member logo is a text cast member. The following statement sets the tunnelDepth of logo to 5. When logo is displayed in 3D mode, its letters will be very shallow.

```
-- Lingo
member("logo").tunnelDepth = 5

// Javascript
member(“logo”).tunnelDepth = 5;
```

In this example, the model resource of the model Slogan is extruded text. The following statement sets the `tunnelDepth` of Slogan's model resource to 1000. Slogan’s letters will be extremely deep.

```
-- Lingo
member("scene").model("Slogan").resource.tunnelDepth = 1000

// Javascript
member(“scene”).getPropRef(“model”,1).getProp(“resource”).tunnelDepth = 1000;
```

See also

`extrude3D`

tweened

Usage

```
sprite(whichSprite).tweened
the tweened of sprite whichSprite
```

Description

Sprite property; determines whether only the first frame in a new sprite is created as a keyframe (`TRUE`), or whether all frames in the new sprite are created as keyframes (`FALSE`).

This property does not affect playback and is useful only during Score recording.

This property can be tested and set.

Example

When this statement is issued, newly created sprites in channel 25 have a keyframe only in the first frame of the sprite span:

```
-- Lingo
sprite(25).tweened = 1

// Javascript
sprite(25).tweened = 1;
```

tweenMode

Usage

```
member(whichCastmember).modelResource(whichModelResource).tweenMode
modelResourceObjectReference.tweenMode
```

Description

3D particle property; allows you to get or set whether the color of a particle varies according to it's speed or age. The `tweenMode` property can have the following values:

- `#velocity` alters the color of the particle between `colorRange.start` and `colorRange.end` based on the velocity of the particle.
- `#age` alters the color of the particle by linearly interpolating the color between `colorRange.start` and `colorRange.end` over the lifetime of the particle. This is the default setting for this property.

Example

In this example, ThermoSystem is a model resource of the type #particle. This statement sets the ThermoSystem's tweenMode to #velocity, so its slower particles will not reach the color specified by colorRange.end, while its faster particles will:

```
-- Lingo
member(8).modelResource("thermoSystem").tweenMode = #velocitytype

// Javascript
member(8).getPropRef("modelResource",1).tweenMode =symbol("velocitytype");
```

type (light)

Usage

```
member(whichCastmember).light(whichLight).type
```

Description

3D light property; the light type of the referenced light. This property's possible values are as follows:

- `#ambient` lights of this type cast their light evenly on all surfaces. The intensity of ambient lights is not affected by distance from the light source.
- `#directional` lights appear to shine in a particular direction, but are not as focused as lights of type `#spot`. The intensity of directional lights decreases with distance from the light source.
- `#point` lights shine in all directions from a specific location in the 3D world. The effect is similar to a bare light bulb hanging in a room. The intensity of point lights decreases with distance from the light source.
- `#spot` Lights of this type cast their light from a particular point and within the cone defined by the light's forward direction and `spotAngle` property. The intensity of spot lights declines with distance from the light source using the values defined in the light's `attenuation` property.

Example

The following statement displays the `type` property of the light named MainLight:

```
-- Lingo
put member("3D").motion("MainLight").type
-- #spot
```

```
// Javascript
trace(member("3D").getPropRef("motion",1).type);
// symbol("spot")
```

See also

spotAngle, attenuation

type (Member)

Usage

```
-- Lingo syntax
memberObjRef.type

// JavaScript syntax
memberObjRef.type;
```

Description

Member property; indicates a cast member's type. Read-only.

The `type` property can be one of the following values:

#animgif	#palette
#bitmap	#physics
#button	#picture
#cursor	#QuickTimeMedia
#digitalVideo	#realMedia
#DVD	#script
#empty	#shape
#field	#shockwave3D
#filmLoop	#sound
#flash	#swa
#flashcomponent	#text
#font	#transition
#havok	#vectorShape
#movie	#windowsMedia
#ole	

This list includes those types of cast members that are available in Director and the Xtra extensions that come with it. You can also define custom cast member types for custom cast members.

For movies created in Director 5, 6 and 6.5, the type property returns #field for field cast members and #richText for text cast members. In all versions from Director 7, #field is returned for field cast members and #text is returned for text cast members.

Example

The following handler checks whether the cast member Today's News is a field cast member and displays an alert if it is not:

```
-- Lingo syntax
on checkFormat
    if (member("Today's News").type <> #field) then
        _player.alert("Sorry, this cast member must be a field.")
    end if
end

// JavaScript syntax
function checkFormat() {
    if (member("Today's News").type != "field") {
        _player.alert("Sorry, this cast member must be a field.");
    }
}
```

See also

Member

type (model resource)

Usage

`member(whichCastmember).modelResource(whichModelResource).type`

Description

3D model resource property; the resource type of the referenced model resource. This property's possible values are:

- `#box` indicates that this model resource is a primitive box resource created using the `newModelResource` command.
- `#cylinder` indicates that this model resource is a primitive cylinder resource created using the `newModelResource` command.
- `#extruder` indicates that this model resource is a primitive text extruder resource created using the `extrude3d` command.
- `#mesh` indicates that this model resource is a primitive mesh generator resource created using the `newMesh` command.
- `#particle` indicates that this model resource is a primitive particle system resource created using the `newModelResource` command.
- `#plane` indicates that this model resource is a primitive plane resource created using the `newModelResource` command.
- `#sphere` indicates that this model resource is a primitive sphere resource created using the `newModelResource` command.
- `#fromFile` indicates that this model resource was created external to Director and was loaded from an external file or a cast member.

Example

The following statement displays the `type` property of the model resource named Helix.

```
put member("helix models").modelResource("Helix").type
-- #fromFile
```

See also

newModelResource, newMesh, extrude3D

type (motion)

Usage

```
member(whichCastmember).motion(whichMotion).type
```

Description

3D motion property; the motion type of the referenced motion object. This property's possible values are:

- `#bonesPlayer` indicates that this motion is a bones based animation and it requires the use of the `#bonesPlayer` modifier for playback.
- `#keyFramePlayer` indicates that this motion is a keyframed animation and it requires the use of the `#keyFramePlayer` modifier for playback.
- `#none` indicates that this motion has no mapped movement and it is suitable for use by either the `#bonesPlayer` or the `#keyFramePlayer` modifier for playback. The default motion object found in every 3D cast member is of this type.

Example

The following statement displays the `type` property of the motion named Run.

```
put member("scene").motion("Run").type
-- #bonesPlayer
```

The following statement displays the `type` property of the motion named DefaultMotion.

```
put member("scene").motion("DefaultMotion").type
-- #none
```

See also

bonesPlayer (modifier), keyframePlayer (modifier)

type (shader)

Usage

```
member(whichCastmember).shader(whichShader).type
```

Description

3D shader property; the shader type of the referenced shader object. This property's possible values are:

- `#standard` indicates that this is a standard shader.
- `#painter` indicates that this is a painter shader.
- `#newsprint` indicates that this is a newsprint shader.
- `#engraver` indicates that this is an engraver shader.

Example

This statement shows that the shader used by the model named box2 is a painter shader:

```
put member("Scene").model("box2").shader.type
-- #painter
```

See also

newShader

type (sprite)

Usage

```
sprite(whichSprite).type
the type of sprite whichSprite
```

Description

Sprite property; clears sprite channels during Score recording by setting the `type` sprite property value for that channel to 0.

Note: *Switch the member of a sprite only to another member of the same type to avoid changing the sprite's properties when the member type is switched.*

This property can be tested and set.

Example

This statement clears sprite channel 1 when issued during a Score recording session:

```
sprite(1).type = 0
```

type (texture)

Usage

```
member(whichCastmember).shader(whichShader).type
```

Description

3D texture property; the texture type of the referenced texture object. This property's possible values are:

- `#fromCastMember` indicates that this is texture was created from a Director cast member supporting the `image` property using the `newTexture` command.
- `#fromImageObject` indicates that this is texture was created from an image object using the `newTexture` command.
- `#importedFromFile` indicates that this texture was created external to Director and created upon file import or cast member loading.

Example

This statement shows that the texture used by the shader for the model named Pluto was created from an image object:

```
put member("scene").model("Pluto").shader.texture.type
-- #fromImageObject
```

See also

newTexture

type (Window)

Usage

```
-- Lingo syntax
windowObjRef.type

// JavaScript syntax
windowObjRef.type;
```

Description

Window property; specifies the window type. Read/write.

If the type property is set, all properties pertaining to the new window are set accordingly.

This property can be one of the following values:

Property	Description
#document	Specifies that the window will appear with a standard title bar, a close box, a minimize box, and a maximize box. These types of windows can be moved.
#tool	Specifies that the window will appear with a shorter title bar and only a small close box in the upper right corner. These types of windows no longer receive activate or deactivate events, because #tool windows are always active. These types of windows will always later with each other, and will always appear on top of #document windows.
#dialog	Specifies that the window will appear with a standard title bar, a close box, and no icon. These types of windows are modal, and will always appear on top of all other windows.

These properties can also be accessed by using the Movie object's displayTemplate property.

Window behaviors also depend on the values of the type property and the Movie object's dockingEnabled property

- If dockingEnabled is TRUE and type is set to #document, the MIAW will look and act like a document window in Director. The window will appear in the "view port" area and be dockable with the Stage, Score, and Cast windows, media editors, and message windows. However, the window will not be able to group with any of these windows.
- If dockingEnabled is TRUE and type is set to #tool, the MIAW will look and act like a tool window in Director. The window will be able to group with all tool windows except the Property inspector and the Tool palette.
- If dockingEnabled is TRUE and type is set to #fullscreen or #dialog, the type is ignored and the window will be an authoring window.

Example

This statement sets the type of the window named Planets to #tool.

```
-- Lingo syntax
window("Planets").type = #tool

// JavaScript syntax
window("Planets").type = "tool";
```

See also

appearanceOptions, displayTemplate, dockingEnabled, titlebarOptions, Window

updateLock

Usage

```
-- Lingo syntax
_movie.updateLock

// JavaScript syntax
_movie.updateLock;
```

Description

Movie property; determines whether the Stage is updated during Score recording (FALSE) or not (TRUE). Read/write.

You can keep the Stage display constant during a Score recording session by setting updateLock to TRUE before script updates the Score. If updateLock is FALSE, the Stage updates to show a new frame each time the frame is entered.

You can also use updateLock to prevent unintentional Score updating when leaving a frame, such as when you temporarily leave a frame to examine properties in another frame.

Although this property can be used to mask changes to a frame during run time, be aware that changes to field cast members appear immediately when the content is modified, unlike changes to location or members with other sprites, which are not updated until this property is turned off.

Example

The following statement displays the value of the updateLock property in the message window.

```
-- Lingo
put _movie.updateLock

// Javascript
trace(_movie.updateLock);
```

See also

Movie

updateMovieEnabled

Usage

```
the updateMovieEnabled
```

Description

System property; specifies whether changes made to the current movie are automatically saved (TRUE) or not saved (FALSE, default) when the movie branches to another movie.

This property can be tested and set.

Example

This statement instructs Director to save changes to the current movie whenever the movie branches to another movie:

```
-- Lingo
the updateMovieEnabled = TRUE

// Javascript
_movie.updateMovieEnabled = true;
```

URL

Usage

```
-- Lingo syntax
memberObjRef.URL

// JavaScript syntax
memberObjRef.URL;
```

Description

Cast member property; specifies the URL for Shockwave Audio (SWA) and Flash movie cast members.

For Flash movie members, this property is synonymous with the `pathName` member property.

The `URL` property can be tested and set. For SWA members, it can be set only when the SWA streaming cast member is stopped.

Example

This statement makes a file on an Internet server the URL for SWA cast member Benny Goodman:

```
-- Lingo syntax
on mouseDown
    member("Benny Goodman").URL = "http://audio.adobe.com/samples/classic.swa"
end

// JavaScript syntax
function mouseDown() {
    member("Benny Goodman").URL = "http://audio.adobe.com/samples/classic.swa"
}
```

useAlpha

Usage

```
-- Lingo syntax
memberObjRef.useAlpha
imageObjRef.useAlpha

// JavaScript syntax
memberObjRef.useAlpha;
imageObjRef.useAlpha;
```

Description

Bitmap cast member and image object property; for 32-bit cast members and image objects with alpha channel information, determines whether Director uses the alpha information when drawing the image on to the Stage (TRUE), or whether Director ignores the alpha information when drawing to the Stage (FALSE).

Example

This toggles the alpha channel of cast member "foreground" on and off:

```
-- Lingo syntax
member("foreground").useAlpha = not(member("foreground").useAlpha)

// JavaScript syntax
switch(member("foreground").useAlpha) {
    case 0:
        member("foreground").useAlpha = 1;
        break;
    case 1:
        member("foreground").useAlpha = 0;
        break;
    }
}
```

useDiffuseWithTexture

Usage

```
member(whichCastmember).shader(whichShader).useDiffuseWithTexture
```

Description

3D standard shader property; allows you to get or set whether the diffuse color is used to modulate the texture (TRUE) or not (FALSE).

When set to TRUE, this property works in conjunction with the blendFunction and blendConstant properties: when blendFunction is set to #blend, the diffuse color is weighed with the texture color to determine the final color. For example, if blendFunction is set to #blend, and blendConstant is set to 100.0, the final color is the pure texture color. If we change blendConstant to 0.0, the final color is the diffuse color. If we change blendConstant to 10.0, the final color is 10% texture color, and 90% diffuse color.

The default value for this property is FALSE.

All shaders have access to the #standard shader properties; in addition to these standard shader properties shaders of the types #engraver, #newsprint, and #painter have properties unique to their type. For more information, see newShader.

Example

In this example, the shaderList of the model MysteryBox contains six shaders. Each shader has a texture list which contains up to eight textures. The diffuseColor property of the cast member (Level2) is set to rgb(255, 0, 0). The blendFunction property of all six shaders is set to #blend, and the blendConstant property of all six shaders is set to 80. This statement sets the useDiffuseWithTexture property of all shaders used by MysteryBox to TRUE. A little bit of red will be blended into the surface of the model. This property is affected by the settings of the blendFunction, blendFunctionList, blendSource, blendSourceList, blendConstant, and blendConstantList properties.

```
-- Lingo
```

```
member("Level2").model("MysteryBox").shaderlist.useDiffuseWithTexture = TRUE

// Javascript
member("Level2").getPropRef("model",1).getProp("shaderlist").useDiffuseWithTexture = true;
```

See also

blendFunction, blendConstant

useFastQuads

Usage

```
-- Lingo syntax
_movie.useFastQuads

// JavaScript syntax
_movie.useFastQuads;
```

Description

Movie property; determines whether to use faster (TRUE) or slower (FALSE, default) quad calculation operations. Read/write.

When set to TRUE, Director uses a faster, less precise method for calculating quad operations. Fast quads calculations are good for simple rotation and skew sprite effects.

When set to FALSE, Director uses the slower, default quad calculation method that provides more visually pleasing results when using quads for distortion and other arbitrary effects.

Simple sprite rotation and skew operations always use the fast quad calculation method, regardless of this setting. Setting useFastQuads to TRUE will not result in an increase in the speed of these simple operations.

Example

This statement tells Director to use its faster quad calculation code for all quad operations in the movie:

```
- Lingo syntax
_movie.useFastQuads = TRUE

// JavaScript syntax
_movie.useFastQuads = true;
```

See also

Movie

useHypertextStyles

Usage

```
-- Lingo syntax
memberObjRef.useHypertextStyles

// JavaScript syntax
memberObjRef.useHypertextStyles;
```

Description

Text cast member property; controls the display of hypertext links in the text cast member.

When `useHypertextStyles` is `TRUE`, all links are automatically colored blue with underlines, and the pointer (cursor) changes to a pointing finger when it is over a link.

Setting this property to `FALSE` turns off the automatic formatting and pointer change.

Example

This behavior toggles the formatting of hypertext on and off in text cast member "myText":

```
--Lingo syntax
on mouseUp
    member("myText").usehypertextStyles = not(member("myText").usehypertextStyles)
end

// JavaScript syntax
function mouseUp() {
    member("myText").usehypertextStyles = !(member("myText").usehypertextStyles)
}
```

useLineOffset

Usage

```
member(whichCastmember).model(whichModel).toon.useLineOffset
member(whichCastmember).model(whichModel).inker.useLineOffset
```

Description

3D `toon` and `inker` modifier property; indicates whether the modifier's `lineOffset` property is used by the modifier when it draws lines on the surface of the model.

The default value of this property is `FALSE`.

Example

The following statement sets the `useLineOffset` property of the `toon` modifier for the model named Teapot to `FALSE`. The `toon` modifier's `lineOffset` property will have no effect.

```
-- Lingo syntax
member("tp").model("Teapot").toon.useLineOffset = FALSE

// JavaScript syntax
member("tp").getPropRef("model",1).getProp("toon").useLineOffset = false;
```

See also

`lineOffset`

userData

Usage

```
member(whichCastmember).model(whichModel).userData
member(whichCastmember).light(whichLight).userData
member(whichCastmember).camera(whichCamera).userData
```

```
member(whichCastmember).group(whichCamera).userData
```

Description

3D property; returns the `userData` property list of a model, group, camera, or light. The default value of this property for an object that was created outside of Director is a list of all the properties that were assigned to the model's `userData` property in the 3D modeling tool. The default value of this property for objects created inside of Director is an empty property list [:], unless the object was created using any of the clone commands. If a cloning command was used to create the object then the new object's `userData` property defaults to a value equal to that of the original source object.

To modify the elements of this list you must use the `addProp` and `deleteProp` commands documented in the main Lingo Dictionary.

Example

This statement displays the `userData` property of the model named New Body:

```
put member("Car").model("New Body").userData
-- [#driver: "Bob", #damage: 34]
```

This statement adds the property `#health` with the value 100 to the `userData` property list for the model named Player:

```
member("scene").model("Player").userData.addProp(#health,100)
```

userName

Usage

```
-- Lingo syntax
_player.userName

// JavaScript syntax
_player.userName;
```

Description

Player property; a string containing the user name entered when Director was installed. Read-only.

This property is available in the authoring environment only. It could be used in a movie in a window (MIAW) tool that is personalized to show the user's information.

Example

The following handler places the user's name and serial number in a display field when the window is opened. (A movie script in the MIAW is a good location for this handler.)

```
-- Lingo syntax
on prepareMovie
    displayString = _player.userName & RETURN & _player.organizationName & RETURN &
_player.serialNumber
    member("User Info").text = displayString
end

// JavaScript syntax
function prepareMovie() {
    var displayString = _player.userName + "\n" + _player.organizationName+ "\n" +
_player.serialNumber;
    member("User Info").text = displayString;
```

```
}
```

See also

Player

userName (RealMedia)

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.userName

// JavaScript syntax
memberOrSpriteObjRef.userName;
```

Description

RealMedia sprite and cast member property; allows you to set the user name required to access a protected RealMedia stream. For security reasons, you cannot use this property to retrieve a previously specified user name. If a user name has been set, the value of this property is the string `"********"`. The default value of this property is an empty string, which means no user name has been specified.

Example

The following examples show that the user name for the RealMedia stream in the cast member Real or sprite 2 has been set.

```
-- Lingo syntax
put(sprite(2).userName) -- "********"
put(member("Real").userName) -- "********"

// JavaScript syntax
put(sprite(2).userName); // "********"
put(member("Real").userName); // "********"
```

The following examples show that the user name for the RealMedia stream in the cast member Real or sprite 2 has never been set.

```
-- Lingo syntax
put(sprite(2).userName) -- ""
put(member("Real").userName) -- ""

// JavaScript syntax
put(sprite(2).userName); // ""
put(member("Real").userName); // ""
```

The following examples set the user name for the RealMedia stream in the cast member Real and sprite 2 to Marcelle.

```
-- Lingo syntax
member("Real").userName = "Marcelle"
sprite(2).userName = "Marcelle"

// JavaScript syntax
member("Real").userName = "Marcelle";
sprite(2).userName = "Marcelle";
```

See also

password

useTargetFrameRate

Usage

```
sprite(which3dSprite).useTargetFrameRate
```

Description

3D sprite property; determines whether the `targetFrameRate` property of the sprite is enforced. If the `useTargetFrameRate` property is set to `TRUE`, the polygon count of the models in the sprite are reduced if necessary to achieve the specified frame rate.

Example

These statements set the `targetFrameRate` property of sprite 3 to 45 and enforce the frame rate by setting the `useTargetFrameRate` property of the sprite to `TRUE`:

```
-- Lingo syntax
sprite(3).targetFrameRate = 45
sprite(3).useTargetFrameRate = TRUE

// JavaScript syntax
sprite(3).targetFrameRate = 45;
sprite(3).useTargetFrameRate = true;
```

See also

`targetFrameRate`

vertex

Usage

```
-- Lingo syntax
memberObjRef.vertex[whichVertexPosition]

// JavaScript syntax
memberObjRef.vertex[whichVertexPosition];
```

Description

Chunk expression; enables direct access to parts of a vertex list of a vector shape cast member.

Use this chunk expression to avoid parsing different chunks of the vertex list. It's possible to both test and set values of the vertex list using this type of chunk expression.

Example

The following code shows how to determine the number of vertex points in a member:

```
-- Lingo syntax
put(member("Archie").vertex.count) -- 2

// JavaScript syntax
put(member("Archie").vertex.count); // 2
```

To obtain the second vertex for the member, you can use code like this:

```
-- Lingo syntax
```

```
put(member("Archie").vertex[2]) -- point(66.0000, -5.0000)

// JavaScript syntax
put(member("Archie").vertex[2]); // point(66.0000, -5.0000)
```

You can also set the value in a control handle:

```
-- Lingo syntax
member("Archie").vertex[2].handle1 = point(-63.0000, -16.0000)

// JavaScript syntax
member("Archie").vertex[2].handle1 = point(-63.0000, -16.0000);
```

See also

vertexList

vertexList

Usage

```
-- Lingo syntax
memberObjRef.vertexList

// JavaScript syntax
memberObjRef.vertexList;
```

Description

Cast member property; returns a linear list containing property lists, one for each vertex of a vector shape. The property list contains the location of the vertex and the control handle. There are no control handles if the location is (0,0).

Each vertex can have two control handles that determine the curve between this vertex and the adjacent vertices. In `vertexList`, the coordinates of the control handles for a vertex are kept relative to that vertex, rather than absolute in the coordinate system of the shape. If the first control handle of a vertex is located 10 pixels to the left of that vertex, its location is stored as (-10, 0). Thus, when the location of a vertex is changed with Lingo, the control handles move with the vertex and do not need to be updated (unless the user specifically wants to change the location or size of the handle).

When modifying this property, be aware that you must reset the list contents after changing any of the values. This is because when you set a variable to the value of the property, you are placing a copy of the list, not the list itself, in the variable. To effect a change, use code like this:

```
- Get the current property contents
currVertList = member(1).vertexList
-- Add 25 pixels to the horizontal and vertical positions of the first vertex in the list
currVertList[1] .vertex = currVertList[1] .vertex + point(25, 25)
-- Reset the actual property to the newly computed position
member(1).vertexList = currVertList
```

Example

This statement displays the vertextList value for an arched line with two vertices:

```
-- Lingo syntax
put(member("Archie").vertexList)-- [[#vertex: point(-66.0000, 37.0000), #handle1: point(-
70.0000, -36.0000), #handle2: point(-62.0000, 110.0000)], [#vertex: point(66.0000, -5.0000),
#handle1: point(121.0000, 56.0000), #handle2: point(11.0000, -66.0000)]]
```

```
// JavaScript syntax
put(member("Archie").vertexList);//[[#vertex: point(-66.0000, 37.0000),#handle1: point(-
70.0000, -36.0000),#handle2: point(-62.0000, 110.0000)], [#vertex: point(66.0000, -
5.0000),#handle1: point(121.0000, 56.0000), #handle2: point(11.0000, -66.0000)]]
```

See also

addVertex(), count(), deleteVertex(), moveVertex(), moveVertexHandle(), originMode, vertex

vertexList (mesh generator)

Usage

```
member(whichCastmember).modelResource(whichModelResource).vertexList
```

Description

3D property; when used with a model resource whose type is #mesh, allows you to get or set the vertexList property for the model resource.

The vertexList is a linear list of each vertex used in the mesh. A single vertex may be shared by numerous faces of the mesh. You can specify a list of any size for this property, but it will store only the number of items specified when using the newMesh() command to create the #mesh model resource.

Example

This statement sets the vertexList of the model resource named Triangle:

```
member("Shapes").modelResource("Triangle").vertexList = [vector(0,0,0), vector(20,0,0),
vector(20, 20, 0)]
```

See also

newMesh, face, vertices

vertexList (mesh deform)

Usage

```
member(whichCastmember).model(whichModel).meshDeform.mesh[index].vertexList
```

Description

3D property; when used with a model with the #meshDeform modifier attached, it allows you to get or set the vertexList property for the specified mesh within the referenced model.

The vertexList is a linear list of each vertex used in the specified mesh. A single vertex may be shared by numerous faces of the mesh.

If a model makes use of the #sds or #lod modifiers in addition to the #meshDeform modifier, then it is important to know that the value of this property will change under the influence of the #sds or #lod modifiers.

Example

This statement displays the #meshDeform modifier's vertexList for the first mesh in the model named Triangle:

```
put member("Shapes").model("Triangle").meshDeform.mesh[1].vertexList
-- [vector(0,0,0), vector(20,0,0), vector(20, 20, 0)]
```

See also

face, vertices, mesh (property)

vertices

Usage

```
member(whichCastmember).modelResource(whichModelResource).face[faceIndex].vertices
```

Description

3D face property; when used with a model resource whose type is #mesh, this property allows you to get or set which vertices from the resource's vertexList property to use for the mesh face specified by *faceIndex*.

This property is a linear list of three integers corresponding to the index positions of the three vertices, as found in the mesh's vertexList property, that comprise the specified face.

The vertices must be specified in the list using counterclockwise winding in order to achieve an outward pointing surface normal.

If you make changes to this property or use the generateNormals() command, you will need to call the build() command in order to rebuild the mesh.

Example

This example displays the vertexList of the mesh model resource named SimpleSquare, then it displays the vertices property for the second face of that mesh:

```
put member("3D").modelResource("SimpleSquare").vertexList
-- [vector( 0.0000, 0.0000, 0.0000), vector( 0.0000, 5.0000, 0.0000), vector( 5.0000, 0.0000,
0.0000), vector( 5.0000, 5.0000, 0.0000)]
put member("3D").modelResource("SimpleSquare").face[1].vertices
-- [3, 4, 1]
```

See also

face, vertexList (mesh deform), generateNormals()

video (QuickTime, AVI)

Usage

```
member(whichCastMember).video
the video of member whichCastMember
```

Description

Digital video cast member property; determines whether the graphic image of the specified digital video cast member plays (TRUE or 1) or not (FALSE or 0).

Only the visual element of the digital video cast member is affected. For example, when video is set to FALSE, the digital video's soundtrack, if present, continues to play.

Example

This statement turns off the video associated with the cast member Interview:

```
-- Lingo syntax
```

```
member("Interview").video = FALSE

// JavaScript syntax
member("Interview").video = false;
```

See also

setTrackEnabled(), trackEnabled

video (RealMedia, Windows Media)

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.video

// JavaScript syntax
memberOrSpriteObjRef.video;
```

Description

RealMedia and Windows Media property; allows you to set or get whether the sprite or cast member renders video (TRUE or 1) or only plays the sounds (FALSE or 0). Read/write.

Integer values other than 1 or 0 are treated as TRUE.

Use this property to suppress the video while playing the audio component of a RealMedia or Windows Media cast member, or to toggle the video on and off during playback.

Example

The following examples show that the video property for sprite 2 and the cast member Real is set to TRUE.

```
-- Lingo syntax
put(sprite(2).video) -- 1
put(member("Real").video) -- 1

// JavaScript syntax
put(sprite(2).video); // 1
put(member("Real").video); // 1
```

The following examples set the video property to FALSE for the RealMedia video element of sprite 2 and the cast member Real.

```
-- Lingo syntax
sprite(2).video = FALSE
member("Real").video = FALSE

// JavaScript syntax
sprite(2).video = 0;
member("Real").video = 0;
```

videoFormat

Usage

```
-- Lingo syntax
dvdObjRef.videoFormat
```

```
// JavaScript syntax
dvdObjRef.videoFormat;
```

Description

DVD property. Returns a symbol that indicates the video format. Read-only.

Possible symbols include the following:

Symbol	Description
#MPEG1	The video format is MPEG-1.
#MPEG2	The video format is MPEG-2.
#unknown	The video format is unknown.

Example

This statement gives the video format of the dvd object sprite.

```
-- Lingo syntax
put sprite(1).videoFormat

// JavaScript syntax
trace(sprite(1).videoFormat)
```

See also

DVD

videoForWindowsPresent

Usage

```
the videoForWindowsPresent
```

Description

System property; indicates whether AVI software is present on the computer.

This property can be tested but not set.

Example

This statement checks whether Video for Windows is missing and branches the playhead to the Alternate Scene marker if it isn't:

```
if the videoForWindowsPresent= FALSE then go to "Alternate Scene"
```

See also

QuickTimeVersion()

viewH

Usage

```
-- Lingo syntax
```

```
memberOrSpriteObjRef.viewH

// JavaScript syntax
memberOrSpriteObjRef.viewH;
```

Description

Cast member and sprite property; controls the horizontal coordinate of a Flash movie and vector shape's view point, specified in pixel units. The values can be floating-point numbers. The default value is 0.

A Flash movie's view point is set relative to its origin point.

Setting a positive value for `viewH` shifts the movie to the left inside the sprite; setting a negative value shifts the movie to the right. Therefore, changing the `viewH` property can have the effect of cropping the movie or even of removing the movie from view entirely.

This property can be tested and set.

Note: *This property must be set to the default value if the scaleMode property is set to #autoSize, or the sprite will not display correctly.*

Example

This handler accepts a sprite reference as a parameter and moves the view of a Flash movie sprite from left to right within the sprite's bounding rectangle:

```
-- Lingo syntax
on panRight whichSprite
    repeat with i = 120 down to -120
        sprite(whichSprite).viewH = i
        _movie.updateStage()
    end repeat
end

// JavaScript syntax
function panRight(whichSprite) {
    var i = 120;
    while(i > -121) {
        sprite(whichSprite).viewH = i;
        _movie.updateStage();
        i--;
    }
}
```

See also

scaleMode, viewV, viewPoint, viewScale

viewPoint

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.viewPoint

// JavaScript syntax
memberOrSpriteObjRef.viewPoint;
```

Description

Cast member property and sprite property; controls the point within a Flash movie or vector shape that is displayed at the center of the sprite's bounding rectangle in pixel units. The values are integers.

Changing the view point of a cast member changes only the view of a movie in the sprite's bounding rectangle, not the location of the sprite on the Stage. The view point is the coordinate within a cast member that is displayed at the center of the sprite's bounding rectangle and is always expressed relative to the movie's origin (as set by the `originPoint`, `originH`, and `originV` properties). For example, if you set a Flash movie's view point at point (100,100), the center of the sprite is the point within the Flash movie that is 100 Flash movie pixel units to the right and 100 Flash movie pixel units down from the origin point, regardless of where you move the origin point.

The `viewPoint` property is specified as a Director point value: for example, point (100,200). Setting a Flash movie's view point with the `viewPoint` property is the same as setting the `viewH` and `viewV` properties separately. For example, setting the `viewPoint` property to point (50,75) is the same as setting the `viewH` property to 50 and the `viewV` property to 75.

Director point values specified for the `viewPoint` property are restricted to integers, whereas `viewH` and `viewV` can be specified with floating-point numbers. When you test the `viewPoint` property, the point values are truncated to integers. As a general guideline, use the `viewH` and `viewV` properties for precision; use the `originPoint` property for speed and convenience.

This property can be tested and set. The default value is point (0,0).

Note: *This property must be set to the default value if the scaleMode property is set to #autoSize, or the sprite will not display correctly.*

Example

This handler makes a specified Flash movie sprite move down and to the right in increments of five Flash movie pixel units:

```
-- Lingo syntax
on panAcross(whichSprite)
    repeat with i = 1 to 10
        sprite(whichSprite).viewPoint = sprite(whichSprite).viewPoint + point(i * -5, i * -
5)
        _movie.updateStage()
    end repeat
end

// JavaScript syntax
function panAcross(whichSprite) {
    var i = 1;
    while(i < 11) {
        sprite(whichSprite).viewPoint = sprite(whichSprite).viewPoint + point(i * -5, i * -
5);
        _movie.updateStage();
        i++
    }
}
```

See also

`scaleMode`, `viewV`, `viewH`, `viewScale`

viewScale

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.viewScale

// JavaScript syntax
memberOrSpriteObjRef.viewScale;
```

Description

Cast member property and sprite property; sets the overall amount to scale the view of a Flash movie or vector shape sprite within the sprite's bounding rectangle. You specify the amount as a percentage using a floating-point number. The default value is 100.

The sprite rectangle itself is not scaled; only the view of the cast member within the rectangle is scaled. Setting the `viewScale` property of a sprite is like choosing a lens for a camera. As the `viewScale` value decreases, the apparent size of the movie within the sprite increases, and vice versa. For example, setting `viewScale` to 200% means the view inside the sprite will show twice the area it once did, and the cast member inside the sprite will appear at half its original size.

One significant difference between the `viewScale` and `scale` properties is that `viewScale` always scales from the center of the sprite's bounding rectangle, whereas `scale` scales from a point determined by the Flash movie's `originMode` property.

This property can be tested and set.

Note: *This property must be set to the default value if the scaleMode property is set to #autoSize, or the sprite will not display correctly.*

Example

This sprite script sets up a Flash movie sprite and doubles its view scale:

```
-- Lingo syntax
property spriteNum

on beginSprite me
    sprite(spriteNum).viewScale = 200
end

// JavaScript syntax
function beginSprite() {
    sprite(this.spriteNum).viewScale = 200;
}
```

See also

`scaleMode`, `viewV`, `viewPoint`, `viewH`

viewV

Usage

```
-- Lingo syntax
memberOrSpriteObjRef.viewV
```

```
// JavaScript syntax
memberOrSpriteObjRef.viewV;
```

Description

Cast member and sprite property; controls the vertical coordinate of a Flash movie and vector shape's view point, specified in pixel units. The values can be floating-point numbers. The default value is 0.

A Flash movie's view point is set relative to its origin point.

Setting a positive value for `viewV` shifts the movie up inside the sprite; setting a negative value shifts the movie down. Therefore, changing the `viewV` property can have the effect of cropping the movie or even of removing the movie from view entirely.

This property can be tested and set.

Note: *This property must be set to the default value if the scaleMode property is set to #autoSize, or the sprite will not display correctly.*

Example

This handler accepts a sprite reference as a parameter and moves the view of a Flash movie sprite from the top to the bottom within the sprite's bounding: rectangle:

```
-- Lingo syntax
on panDown(whichSprite)
    repeat with i = 120 down to -120
        sprite(whichSprite).viewV = i
        _movie.updateStage()
    end repeat
end

// JavaScript syntax
function panDown(whichSprite) {
    var i = 120;
    while(i > -121) {
        sprite(whichSprite).viewV = i;
        _movie.updateStage();
        i--;
    }
}
```

See also

scaleMode, viewV, viewPoint, viewH

visible

Usage

```
-- Lingo syntax
windowObjRef.visible

// JavaScript syntax
windowObjRef.visible;
```

Description

Window property; determines whether a window is visible (`TRUE`) or not (`FALSE`). Read/write.

Example

This statement makes the window named Control_Panel visible:

```
-- Lingo syntax
window("Control_Panel").visible = TRUE

// JavaScript syntax
window("Control_Panel").visible = true;
```

See also

Window

visible (sprite)

Usage

```
sprite(whichSprite).visible
the visible of sprite whichSprite
```

Description

Sprite property; determines whether the sprite specified by *whichSprite* is visible (TRUE) or not (FALSE). This property affects all sprites in the channel, regardless of their position in the Score.

Note: *Setting the visible property of a sprite channel to FALSE makes the sprite invisible and prevents only the mouse-related events from being sent to that channel. The beginSprite, endSprite, prepareFrame, enterFrame, and exitFrame events continue to be sent regardless of the sprite's visibility setting. Clicking the Mute button on that channel in the Score, however, will set the visible property to FALSE and prevent all events from being sent to that channel. Muting disables a channel, while setting a sprite's visible property to FALSE merely affects a graphic property.*

This property can be tested and set. If set to FALSE, this property will not automatically reset to TRUE when the sprite ends. You must set the visible property of the sprite to TRUE in order to see any other members using that channel.

Example

This statement makes sprite 8 visible:

```
sprite(8).visible = TRUE
```

visibility

Usage

```
member(whichCastmember).model(whichModel).visibility
modelObjectReference.visibility
```

Description

3D property; allows you to get or set the visibility property of the referenced model. This property determines how the model's geometry is drawn. It can have one of the following values:

- #none specifies that no polygons are drawn and the model is invisible.
- #front specifies that only those polygons facing the camera are drawn.This method is referred to as back face culling and optimizes rendering speed. This is the default setting for the property.

• `#back` specifies that only those polygons facing away from the camera are drawn. Use this setting when you want to draw the inside of a model, or for models which are not drawing correctly, possibly because they were imported from a file format that used a different handiness value when computing normals.

• `#both` specifies that both sides of all polygons are drawn.Use this setting when you want to see the plane regardless of the viewing direction, and for models that are not drawing correctly.

Example

The following statement shows that the `visibility` property of the model Monster02 is set to `#none`. The model is invisible.

```
-- Lingo syntax
put member("3D").model("Monster02").visibility
-- #none

// JavaScript syntax
trace(member("3D").getPropRef("model",1).visibility)
// symbol("none")
```

volume (DVD)

Usage

```
-- Lingo syntax
dvdObjRef.volume

// JavaScript syntax
dvdObjRef.volume;
```

Description

DVD property. Determines the current DVD sound volume . Read/write.

The volume must be an integer in the range of 0 (silent) to 100 (full volume).

On Windows the volume scale is logarithmic. On Mac the scale is linear.

Example

This statement sets the volume of DVD member

```
-- Lingo syntax
member(1).volume = 20

// JavaScript syntax
member(1).volume = 20;
```

See also

DVD

volume (Member)

Usage

```
-- Lingo syntax
memberObjRef.volume
```

```
// JavaScript syntax
memberObjRef.volume;
```

Description

Shockwave Audio (SWA) cast member property; determines the volume of the specified SWA streaming cast member. Values range from 0 to 255.

This property can be tested and set.

Example

This statement sets the volume of an SWA streaming cast member to half the possible volume:

```
-- Lingo syntax
member("SWAfile").volume = 128

// JavaScript syntax
member("SWAfile").volume = 128;
```

volume (Sound Channel)

Usage

```
-- Lingo syntax
soundChannelObjRef.volume

// JavaScript syntax
soundChannelObjRef.volume;
```

Description

Sound Channel property; determines the volume of a sound channel. Read/write.

Sound channels are numbered 1, 2, 3, and so on up to 8. Channels 1 and 2 are the channels that appear in the Score.

The value of the `volume` property ranges from 0 (mute) to 255 (maximum volume). A value of 255 indicates the full volume set for the machine, as controlled by the Sound object's `soundLevel` property, and lower values are scaled to that total volume. This property allows several channels to have independent settings within the available range.

The lower the value of the `volume` sound property, the more static or noise you're likely to hear. Using `soundLevel` may produce less noise, although this property offers less control.

To see an example of `volume` used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

Example

This statement sets the volume of sound channel 2 to 130, which is a medium sound level setting:

```
-- Lingo syntax
sound(2).volume = 130

// JavaScript syntax
sound(2).volume = 130;
```

See also

Sound Channel, soundLevel

volume (Sprite)

Usage

```
-- Lingo syntax
spriteObjRef.volume

// JavaScript syntax
spriteObjRef.volume;
```

Description

Sprite property; controls the volume of a digital video movie or Windows Media cast member specified by name or number. The values range from 0 to 256. Values of 0 or less mute the sound. Values exceeding 256 are loud and introduce considerable distortion.

Example

This statement sets the volume of the QuickTime movie playing in sprite channel 7 to 256, which is the maximum sound volume:

```
-- Lingo syntax
sprite(7).volume = 256

// JavaScript syntax
sprite(7).volume = 256;
```

See also

soundLevel

volume (Windows Media)

Usage

```
-- Lingo syntax
windowsMediaObjRef.volume

// JavaScript syntax
windowsMediaObjRef.volume;
```

Description

Windows Media sprite property; determines the volume of a Windows Media sprite.

The value of this property is an integer that ranges from 0 (mute) to 7 (loud).

You can also set this property using the Control > Volume menu in Director.

Example

This statement sets the volume of sprite 7 to 2:

```
-- Lingo syntax
sprite(7).volume = 2

// JavaScript syntax
sprite(7).volume = 2;
```

See also

Windows Media

warpMode

Usage

```
-- Lingo syntax
spriteObjRef.warpMode

// JavaScript syntax
spriteObjRef.warpMode;
```

Description

QuickTime VR sprite property; specifies the type of warping performed on a panorama.

Possible values are `#full`, `#partial`, and `#none`.

This property can be tested and set. When tested, if the values for the static and motion modes differ, the property's value is the value set for the current mode. When set, the property determines the warping for both the static and motion modes.

Example

This sets the warpMode of sprite 1 to #full:

```
-- Lingo syntax
sprite(1).warpMode = #full

// JavaScript syntax
sprite(1).warpMode = symbol("full");
```

width

Usage

```
-- Lingo syntax
memberObjRef.width
imageObjRef.width
spriteObjRef.width

// JavaScript syntax
memberObjRef.width;
imageObjRef.width;
spriteObjRef.width;
```

Description

Member, Image, and Sprite property; for vector shape, Flash, animated GIF, RealMedia, Windows Media, bitmap, and shape cast members, determines the width, in pixels, of a cast member. Read-only for cast members and image objects, read/write for sprites.

This property does not affect field and button cast members.

Example

This statement assigns the width of member 50 to the variable `theHeight`:

```
-- Lingo syntax
theHeight = member(50).width

// JavaScript syntax
var theHeight = member(50).width;
```

This statement sets the width of sprite 10 to 26 pixels:

```
-- Lingo syntax
sprite(10).width = 26

// JavaScript syntax
sprite(10).width = 26;
```

This statement assigns the width of sprite number i + 1 to the variable `howWide`:

```
-- Lingo syntax
howWide = sprite(i + 1).width

// JavaScript syntax
var howWide = sprite(i + 1).width;
```

See also

`height`, `image (Image)`, `Member`, `Sprite`

width (3D)

Usage

```
member(whichCastmember).modelResource(whichModelResource).width
modelResourceObjectReference.width
```

Description

3D property; allows you to get or set the width of the plane for a model resource whose type is `#box` or `#plane`. This property must be greater than 0.0, and has a default setting of 1.0. For objects whose type is `#box`, the default value of `width` is 50.0. For objects whose type is `#plane`, the default setting is 1.0. `width` is measured along the X axis.

Example

This statement sets the width of the model resource Grass plane to 250.0:

```
-- Lingo syntax
member("3D World").modelResource("Grass plane").width = 250.0

// JavaScript syntax
Member("3D World").getPropRef("modelResource",1).width = 250.0;
```

widthVertices

Usage

```
member(whichCastmember).modelResource(whichModelResource).
widthVertices
modelResourceObjectReference.widthVertices
```

Description

3D property; allows you to get or set the number of vertices (as an integer) on the X axis of a model resource whose type is `#box` or `#plane`. This property must be greater than or equal to 2, and has a default value of 2.

Example

The following statement sets the `widthVertices` property of the model resource Tower to 10. Eighteen polygons (2 * (10-1) triangles) will be used to define the geometry of the model resource along its x-axis.

```
member("3D World").modelResource("Tower").widthVertices = 10
```

wind

Usage

```
member(whichCastmember).modelResource(whichModelResource).wind
modelResourceObjectReference.wind
```

Description

3D property; allows you to get or set the `wind` property of a model resource whose type is `#particle`, as a vector.

This `wind` property defines the direction and strength of the wind force applied to all particles during each simulation step. The default value for this property is vector(0, 0, 0), which specifies that no wind is applied.

Example

```
-- Lingo syntax
put member("3D").modelResource("fog bank").wind
-- vector(10.5,0,0)

// JavaScript syntax
trace(member("3D").getPropRef("modelResource","1").wind)
// vector(10.5,0,0)
```

window

Usage

```
-- Lingo syntax
_player.window[windowNameOrNum]

// JavaScript syntax
_player.window[windowNameOrNum];
```

Description

Player property; provides indexed or named access to the Window objects created by the Director player. Read-only.

The *`windowNameOrNum`* argument is either a string that specifies the name of the window to access or an integer that specifies the index position of the window to access.

The functionality of this property is identical to the top level `window()` method.

Example

This statements sets the variable named `myWindow` to the third window object:

```
-- Lingo syntax
```

```
myWindow = _player.window[3]

// JavaScript syntax
var myWindow = _player.window[3];
```

See also

Player, window()

windowBehind

Usage

```
-- Lingo syntax
windowObjRef.windowBehind

// JavaScript syntax
windowObjRef.windowBehind;
```

Description

Window property; returns a reference to the window that is behind all other windows. Read-only.

Example

These statements set the variable `backWindow` to the window behind all other windows, and then moves that window to the front:

```
-- Lingo syntax
backWindow = _player.windowList[5].windowBehind
backWindow.moveToFront()

// JavaScript syntax
var backWindow = _player.windowList[5].windowBehind;
backWindow.moveToFront();
```

See also

moveToBack(), moveToFront(), Window, windowInFront, windowList

windowInFront

Usage

```
-- Lingo syntax
windowObjRef.windowInFront

// JavaScript syntax
windowObjRef.windowInFront;
```

Description

Window property; returns a reference to the window that is in front of all other windows. Read-only.

Example

These statements set the variable `frontWindow` to the window in front of all other windows, and then moves that window to the back:

```
-- Lingo syntax
frontWindow = _player.windowList[5].windowInFront
frontWindow.moveToBack()

// JavaScript syntax
var frontWindow = _player.windowList[5].windowInFront
frontWindow.moveToBack();
```

See also

moveToBack(), moveToFront(), Window, windowBehind, windowList

windowList

Usage

```
-- Lingo syntax
_player.windowList

// JavaScript syntax
_player.windowList;
```

Description

Player property; displays a list of references to all known movie windows. Read-only.

The Stage is also considered a window.

Example

This statement displays in the Message window a list of all known movie windows:

```
-- Lingo syntax
trace(_player.windowList)

// JavaScript syntax
trace(_player.windowList);
```

See also

Player

wordWrap

Usage

```
-- Lingo syntax
memberObjRef.wordWrap

// JavaScript syntax
memberObjRef.wordWrap;
```

Description

Cast member property; determines whether line wrapping is allowed (TRUE) or not (FALSE).

Example

This statement turns line wrapping off for the field cast member Rokujo:

```
--Lingo syntax
member("Rokujo").wordWrap = FALSE

// JavaScript syntax
member("Rokujo").wordWrap = false;
```

worldPosition

Usage

```
member(whichCastmember).model(whichModel).worldPosition
member(whichCastmember).light(whichLight).worldPosition
member(whichCastmember).camera(whichCamera).worldPosition
member(whichCastmember).group(whichGroup).worldPosition
```

Description

3D property; allows you to get and not set the position of the specified node in world coordinates. A node can be a model, group, camera, or light. This property is equivalent in result to using `getWorldTransform().position` command. The position of a node is represented by a vector object.

Example

This statement shows that the position of the model named Mars, in world coordinates, is the vector (-1333.2097, 0.0000, -211.0973):

```
--Lingo syntax
put member("scene").model("Mars").worldPosition
-- vector(-1333.2097, 0.0000, -211.0973)

// JavaScript syntax
trace(member("scene").getProp("model",1).worldPosition)
// vector(-1333.2097, 0.0000, -211.0973)
```

See also

`getWorldTransform()`, `position (transform)`

worldTransform

Usage

```
member(whichMember).model(whichModel).bonesPlayer.bone[index].worldTransform
```

Description

3D bonesplayer property; allows you to get the world relative transform of a specific bone, as opposed to using the `transform` property which returns the bone's parent relative transform. The `worldTransform` property can only be used with bonesplayer modified models.

Example

This statement stores a bone's world relative transform in the variable `finalTransform`:

```
--Lingo syntax
finalTransform = member("3D").model("biped").bonesPlayer.bone[3].worldTransform

// JavaScript syntax
```

```
finalTransform =
member("3D").getPropRef("model",1).getProp("bonesPlayer").bone[3].worldTransform;
```

See also

bone, getWorldTransform(), transform (property)

wrapTransform

Usage

```
member( whichCastmember ).shader( ShaderName ).wrapTransform
member( whichCastmember ).shader[ ShaderIndex ].wrapTransform
member( whichCastmember ).model[modelName].shader.wrapTransform
member( whichCastmember ).model.shaderlist[ shaderListIndex ].wrapTransform
```

Description

3D standard shader property; this property provides access to a transform that modifies the texture coordinate mapping for the shader's `texture`. Rotate this transform to alter how the texture is projected onto a model surface. The texture remains unaffected; the transform modifies only the orientation of how the shader applies the texture.

Note: *Note: This command only has an effect when the shader's textureModeList is set to is #planar, #spherical, or #cylindrical.*

Example

These statements set the `transformMode` of the shader named "shad2" to `#wrapCylindrical`, then rotates that cylindrical projection by 90° about the x-axis so that the cylindrical mapping wraps around the *y*-axis instead of the *z*-axis:

```
--Lingo syntax
s = member("Scene").shader("shad2")
s.textureMode= #wrapCylindrical
s.wrapTransform.rotate(90.0, 0.0, 0.0)

// JavaScript syntax
var s = member("Scene").getPropRef("shader",1);
s.textureMode = symbol("wrapCylindrical");
s.wrapTransform.rotate(90.0,0.0,0.0);
```

wrapTransformList

Usage

```
member( whichCastmember ).shader( ShaderName ).wrapTransformList[ textureLayerIndex ]
member( whichCastmember ).shader[ shaderListIndex ].wrapTransformList[ textureLayerIndex ]
member( whichCastmember ).model( modelName ).shader.wrapTransformList[ textureLayerIndex ]
member( whichCastmember ).model( modelName ).shaderList[ shaderListIndex ].
wrapTransformList[ textureLayerIndex ]
```

Description

3D standard shader property; this property provides access to a transform that modifies the texture coordinate mapping of a specified texture layer. Rotate this transform to alter how the texture is projected onto model surfaces. The texture itself remains unaffected; the transform modifies only the orientation of how the shader applies the texture.

Note: *wrapTransformList[textureLayerIndex] only has an effect when textureModeList[`textureLayerIndex`] is set to #planar, #spherical, or #cylindrical.*

Example

In this example, line 2 sets the `transformMode` of the third texture layer of the shader named "shad2" to `#wrapCylindrical`. Line 3 rotates that cylindrical projection by 90° about the x-axis so that the cylindrical mapping wraps around the *y*-axis instead of the *z*-axis.

```
--Lingo syntax
s = member("Scene").shader("shad2")
s.textureModeList[3] = #wrapCylindrical
s.wrapTransformList[3].rotate(90.0, 0.0, 0.0)

// JavaScript syntax
var s = member("Scene").getPropRef("shader",1);
s.textureModeList[3] = symbol("wrapCylindrical");
s.wrapTransformList[3].rotate(90.0, 0.0, 0.0);
```

See also

`newShader`, `textureModeList`

x (vector)

Usage

```
member(whichCastmember).vector.x
member(whichCastmember).vector[1]
```

Description

3D property; allows you to get or set the *x* component of a vector.

Example

This statement shows the *x* component of a vector:

```
vec = vector(20, 30, 40)
put vec.x
-- 20.0000
```

xAxis

Usage

```
member(whichCastmember).transform.xAxis
```

Description

3D transform property; allows you to get but not set the vector representing the transform's canonical *x*-axis in transform space.

Example

The first line of this example sets the transform of the model ModCylinder to the identity transform. The next two lines show that the *x*-axis of ModCylinder is the vector (1.0000, 0.0000, 0.0000). This means that the *x*-axis of ModCylinder is aligned with the *x*-axis of the world. The next line rotates ModCylinder 90° around its y-axis. This rotates the axes of ModCylinder as well. The last two lines show that the *x*-axis of ModCylinder is now the vector (0.0000, 0.0000, -1.0000). This means that the *x*-axis of ModCylinder now is aligned with the negative *z*-axis of the world.

```
-- Lingo syntax
member("Engine").model("ModCylinder").transform.identity()
put member("Engine").model("ModCylinder").transform.xAxis
-- vector( 1.0000, 0.0000, 0.0000 )
member("Engine").model("ModCylinder").rotate(0, 90, 0)
put member("Engine").model("ModCylinder").transform.xAxis
-- vector( 0.0000, 0.0000, -1.0000 )

// JavaScript syntax
member("Engine").getPropRef("model",1).transform.identity();
trace(member("Engine").getPropRef("model",1).transform.xAxis)
// vector( 1.0000, 0.0000, 0.0000 )
member("Engine").getPropRef("model",1).rotate(0, 90, 0);
trace(member("Engine").getPropRef("model",1).transform.xAxis)
// vector( 0.0000, 0.0000, -1.0000 )
```

xtra

Usage

```
-- Lingo syntax
_player.xtra[xtraNameOrNum]

// JavaScript syntax
_player.xtra[xtraNameOrNum];
```

Description

Player property; provides indexed or named access to the Xtra extensions available to the Director player. Read-only.

The *xtraNameOrNum* argument is either a string that specifies the name of the Xtra extension to access or an integer that specifies the index position of the Xtra extension to access.

The functionality of this property is identical to the top level `xtra()` method.

Example

This statement sets the variable `myXtra` to the Speech Xtra extension:

```
-- Lingo syntax
myXtra = _player.xtra["SpeechXtra"]

// JavaScript syntax
var myXtra = _player.xtra["SpeechXtra"];
```

See also

Player, xtra()

xtraList (Movie)

Usage

```
-- Lingo syntax
_movie.xtraList

// JavaScript syntax
_movie.xtraList;
```

Description

Movie property; displays a linear property list of all Xtra extensions in the Movies/Xtras dialog box that have been added to the movie. Read-only.

Two possible properties can appear in `xtraList`:

- `#filename`—Specifies the filename of the Xtra extension on the current platform. It is possible to have a list without a `#filename` entry, such as when the Xtra extension exists only on one platform.
- `#packageurl`—Specifies the location, as a URL, of the download package specified by `#packagefiles`.
- `#packagefiles`—Set only when the Xtra extension is marked for downloading. The value of this property is another list containing a property list for each file in the download package for the current platform. The properties in this subproperty list are `#name` and `#version`, which contain the same information as found in `xtraList (Player)`.

Example

This statement displays the `xtraList` in the Message window:

```
-- Lingo syntax
put(_movie.xtraList)

// JavaScript syntax
put(_movie.xtraList);
```

See also

Movie, xtraList (Player)

xtraList (Player)

Usage

```
-- Lingo syntax
_player.xtraList

// JavaScript syntax
_player.xtraList;
```

Description

Player property; displays a linear property list of all available Xtra extensions and their file versions. Read-only.

This property is useful when the functionality of a movie depends on a certain version of an Xtra extension.

There are two possible properties that can appear in `xtraList`:

- `#filename`—Specifies the filename of the Xtra extension on the current platform. It is possible to have a list without a `#filename` entry, such as when the Xtra extension exists only on one platform.
- `#version`—Specifies the same version number that appears in the Properties dialog box (Windows) or Info window (Mac) when the file is selected on the desktop. An Xtra extension may not necessarily have a version number.

Example

This statement displays in the Message window all Xtra extensions that are available to the Director Player.

```
-- Lingo syntax
trace(_player.xtraList)

// JavaScript syntax
trace(_player.xtraList);
```

See also

mediaXtraList, Player, scriptingXtraList, toolXtraList, transitionXtraList

y (vector)

Usage

```
member(whichCastmember).vector.y
member(whichCastmember).vector[2]
```

Description

3D property; allows you to get or set the *y* component of a vector.

Example

This statement shows the y component of a vector:

```
vec = vector(20, 30, 40)
put vec.y
-- 30.0000
```

yAxis

Usage

```
member(whichCastmember).transform.yAxis
```

Description

3D transform property; allows you to get but not set the vector representing the transform's canonical *y*-axis in transform space.

Example

The first line of this example sets the transform of the model ModCylinder to the identity transform. The next two lines show that the Y axis of ModCylinder is the vector (0.0000, 1.0000, 0.0000). This means that the *y*-axis of ModCylinder is aligned with the *y*-axis of the world. The next line rotates ModCylinder 90° around its *x*-axis. This rotates the axes of ModCylinder as well. The last two lines show that the *y*-axis of ModCylinder is now the vector (0.0000, 0.0000, 1.0000). This means that the *y*-axis of ModCylinder now is aligned with the positive *z*-axis of the world.

```
member("Engine").model("ModCylinder").transform.identity()
put member("Engine").model("ModCylinder").transform.yAxis
-- vector( 0.0000, 1.0000, 0.0000 )
member("Engine").model("ModCylinder").rotate(90, 0, 0)
put member("Engine").model("ModCylinder").transform.yAxis
-- vector( 0.0000, 0.0000, 1.0000 )
```

yon

Usage

```
member(whichCastmember).camera(whichCamera).yon
```

Description

3D property; allows you to get or set the distance from the camera defining where along the camera's Z axis the view frustum is clipped. Objects at a distance greater than `yon` are not drawn.

The default value for this property is 3.40282346638529e38.

Example

This statement sets the `yon` property of camera 1 to 50000:

```
-- Lingo syntax
member("3d world").camera[1].yon = 50000

// JavaScript syntax
member("3d world").getPropRef("camera",1).yon = 50000
```

See also

hither

z (vector)

Usage

```
member(whichCastmember).vector.z
member(whichCastmember).vector[3]
```

Description

3D property; allows you to get or set the *z* component of a vector.

Example

This statement shows the z component of a vector:

```
vec = vector(20, 30, 40)
```

```
put vec.z
-- 40.0000
```

zAxis

Usage

```
member(whichCastmember).transform.zAxis
```

Description

3D transform property; allows you to get but not set the vector representing the transform's canonical *z*-axis in transform space.

Example

The first line of this example sets the transform of the model ModCylinder to the identity transform. The next two lines show that the *z*-axis of ModCylinder is the vector (0.0000, 0.0000, 1.0000). This means that the *z*-axis of ModCylinder is aligned with the *z*-axis of the world. The next line rotates ModCylinder 90° around its *y*-axis. This rotates the axes of ModCylinder as well. The last two lines show that the *z*-axis of ModCylinder is now the vector (1.0000, 0.0000, 0.0000). This means that the *z*-axis of ModCylinder now is aligned with the *x*-axis of the world.

```
-- Lingo syntax
member("Engine").model("ModCylinder").transform.identity()
put member("Engine").model("ModCylinder").transform.zAxis
-- vector( 1.0000, 0.0000, 0.0000 )
member("Engine").model("ModCylinder").rotate(0, 90, 0)
put member("Engine").model("ModCylinder").transform.zAxis
-- vector( 0.0000, 0.0000, -1.0000 )

// JavaScript syntax
member("Engine").getPropRef("model",1).transform.identity();
trace(member("Engine").getPropRef("model",1).transform.zAxis)
//vector( 1.0000, 0.0000, 0.0000 )
member("Engine").getPropRef("model",1).rotate(0, 90, 0);
trace(member("Engine").getPropRef("model",1).transform.zAxis)
// vector( 0.0000, 0.0000, -1.0000 )
```

Chapter 15: Physics Engine

The Physics Xtra is a high-performance tool that helps developers create 3D worlds in which objects interact. The Xtra performs calculations to determine the results of collisions, factoring in object properties such as mass, velocity, and rotation. Forces can be applied, and objects can be connected to each other with constraints. The constraints available are 6 degree of freedom joints, linear joints, angular joints, and spring joints.

Additonally, terrains and raycasting are supported. A terrain is similar to a bumpy plane that is infinite in two dimensions and defines an elevation along the third. Raycasting is the mechanism of collision detection with rays. Raycasting can be done against all types of rigid bodies and terrains.

With this Xtra, developers can focus on game play and user interaction, and not worry about creating a real-time physics engine with Lingo scripts.

The Physics (dynamiks) Xtra is a fully integrated rigid body physics simulation engine for Adobe® Director®. The dynamics Xtra is supported on Windows and MAC platforms.

The physics engine for Director has the following features. Refer to the rest of the chapter for their details.

Physics World properties

angularDamping

Usage

```
world.angularDamping
```

Access: Get/Set

Type: float

Default value: 0.000

Description

Represents the damping quantity by which the angular forces are reduced.

Example

```
--Lingo Syntax
member("PhysicsWorld").angularDamping = 10.0

//Javascript syntax
member("PhysicsWorld").angularDamping = 10.0;
```

See also

linearDamping

contactTolerance

Usage

```
world.contactTolerance
```

Access: Get/Set

Type: float

Description

The penetration depth between rigid bodies for the collision to be detected.

Note: *To get the expected result when simulating a collision between bodies, tweak the contact tolerance value.*

Example

This statement sets the penetration depth between rigid bodies for the collision to be detected.

```
--Lingo Syntax
member("PhysicsWorld").contactTolerance = 0.01

//JavaScript Syntax
member("PhysicsWorld").contactTolerance = 0.01;
```

Note:

- Ideally, the contact tolerance should be 2% of the rigid body dimensions.
- If the contact tolerance for each rigid body is not specified, the value set for the world is used.
- Higher values for contact tolerance will lead to numerical errors and instability.

friction

Usage

```
world.friction
```

Access: Get/Set

Type: float

Value 0 - 1

Description

This represents the default value for co-efficient of friction for all rigid bodies in the world. The friction property takes values from 0-1.

Example

```
--Lingo Syntax
member("PhysicsWorld").friction = 0.5

//Javascript syntax
member("PhysicsWorld").friction = 0.5;
```

See also

restitution

gravity

Usage

```
world.gravity
```

Access: Get/Set

Type: vector

Description

This represents the acceleration due to gravity that all bodies in the physics world will experience.

Example

```
--Lingo Syntax
member("PhysicsWorld").gravity = vector(0,-9.81,0)

//Javascript syntax
member("PhysicsWorld").gravity = vector(0,-9.81,0);
```

IsInitialized

Usage

```
world.isInitialized
```

Access: get

Type: Boolean

Description

The current initialization status of the world. If init() method on world succeeds, then this property is set to 1.

Example

```
--Lingo Syntax
If not member("PhysicsWorld").isInitalized then
-- Initialize the physics world here
end if

//JavaScript Syntax
If (member("PhysicsWorld").isInitialized == 0) {
    // Initialize World here
}
```

See also

init()

linearDamping

Usage

```
world.linearDamping
```

Access: Get/Set

Type: float

Default value: 0.000

Description

Represents the loss of linear velocity at the end of a time step.

Example

```
--Lingo Syntax
member("PhysicsWorld").linearDamping = 10.0
```

```
//Javascript syntax
member("PhysicsWorld").linearDamping = 10.0;
```

See also

angularDamping

restitution

Usage

```
world.restitution
```

Access: Get/Set

Type: float

Value 0 - 1

Description

This represents the default value for co-efficient of restitution for all rigid bodies in the world. The restitution property takes values from 0-1.

Example

```
--Lingo Syntax
member("PhysicsWorld").restitution = 0.5
```

```
//Javascript syntax
member("PhysicsWorld").restitution = 0.5;
```

See also

friction

scalingFactor

Usage

```
world.scalingFactor
```

Access: Get

Type: vector

Description

Represents the scaling used for conversion if the 3D world and the physics world are of different units. scalingFactor is a vector which takes the scaling for length, mass and time.

Note: *You cannot set the value for this property in Director 11. This feature will be available in future releases.*

Example

```
--Lingo Syntax
put member("PhysicsWorld").scalingFactor
```

```
//Javascript syntax
put(member('PhysicsWorld').scalingFactor)
```

See also

`init()`

sleepMode

Usage

```
world.sleepMode
```

Access: Get/Set

Values: #energy, #linearvelocity

Default value: #energy

Description

This gets or sets the sleep mode for rigidbodies.

#energy The kinetic energy of the rigid body is used to determine its sleep state.

#linearvelocity The linear velocity of the rigid body is used to determine its sleep state.

Example

```
--Lingo Syntax
member("PhysicsWorld").sleepMode = #energy
```

```
//Javascript syntax
member("PhysicsWorld").sleepMode = symbol("energy");
```

See also

`sleepThreshold`

sleepThreshold

Usage

```
world.sleepThreshold
```

Access: Get/Set

Type: float

Default value: 0.000

Description

This represents the threshold below which the Rigid body becomes inactive. This value is either the kinetic energy or the linear velocity of the rigid body depending on the value set for the sleepMode property. This is only applicable to dynamic objects.

Example

```
--Lingo Syntax
member("PhysicsWorld").sleepThreshold = 10.0

//Javascript syntax
member("PhysicsWorld").sleepThreshold = 10.0;
```

See also

sleepMode

subSteps

Usage

```
world.subSteps
```

Access: Get /Set

Type: int

Description

Represents the number of substeps for each simulate call.

Example

```
--Lingo Syntax
member("PhysicsWorld").subSteps = 5

//Javascript syntax
member("PhysicsWorld").subSteps = 5;
```

See also

init(),timeStep,timeStepMode

timeStep

Usage

```
world.timeStep
```

Access: Get/Set

Type: float

Description

Used only in the equal time step mode where it specifies the step time for the physics world.

Example

```
--Lingo Syntax
member("PhysicsWorld"). timeStep = 0.0167

//Javascript syntax
member("PhysicsWorld"). timeStep = 0.0167;
```

See also

init(),subSteps,timeStepMode

timeStepMode

Usage

```
world.timeStepMode
```

Access: Get/Set

Description

This returns the current time step mode as specified in the init() call to the world. This can be either #automatic or #equal

#equal - Steps the physics world by time specified in the init call's timeStep parameter.

#automatic - Steps the physics world by the actual time elapsed.

Example

This statement sets the current time step mode for the physics world to "equal"

```
--Lingo Syntax
member("PhysicsWorld").timeStepMode = #equal

//JavaScript Syntax
member("PhysicsWorld").timeStepMode = symbol("equal");
```

See also

init(),timeStep,subSteps

Physics World methods

- destroy()
- getSimulationTime()
- init()
- PauseSimulation()
- ResumeSimulation ()
- simulate()

destroy()

Usage

```
<void>world.destroy()
```

Description

This stops the simulation for the world and frees all resources in the world. To restart simulation, Init() has to be called on the world.

Parameters

None

Example:

```
--Lingo Syntax
member("PhysicsWorld").destroy()

//Javascript syntax
member("PhysicsWorld").destroy();
```

Note: *If destroy() is not called and the movie is replayed in Director Authoring, then an error occurs when any Physics object is created.*

See also

`init()`

getSimulationTime()

Usage

```
<float>world.getSimulationTime()
```

Description

Returns the elapsed simulation time of the physics world.

Parameters

None

Example

```
--Lingo Syntax
nFloatTime = member("PhysicsWorld").getSimulationTime()

//Javascript syntax
var nFloatTime = member("PhysicsWorld").getSimulationTime();
```

See also

`simulate()`

init()

Usage

```
world.init(string 3dmembername, vector scalingFactor, symbol timeStepMode, float timeStep,
int substepCount).
world.init(3dmemberref, vector scalingFactor, symbol timeStepMode,float timeStep, int
substepCount).
```

Description

This function initializes the physics world with the specified 3D cast member. Each physics cast member should be associated with a unique 3D cast member represented by the "3dmembername".

- You must end the init() function with world.destroy() before calling another init() function.
- Do not use the resetworld() method between the init() and destroy() methods in a Physics simulation.
- It is recommended that you do not apply Physics to models with keyframe animation or bonesplayer animation.

Note: *A -2 error is thrown if the world is not initialized before accessing a method or property of a physics object.*

Parameters

Parameter	Description
3dmembername	Required. String/member reference that specifies a unique 3D cast member
ScalingFactor	Required. Vector that specifies the scaling factor of the 3d world in physics.
Timestep	Value in milliseconds.
Timestepmode	Required. #equal - Steps the physics world by time specified in the init call's timeStep parameter. #automatic - Steps the physics world by the actual time elapsed.
subStepcount	Required. Integer value that is calculated for accuracy.

Example:

```
--Lingo Syntax
member("PhysicsWorld").init("3dmembername",vector(1.0,1.0,1.0),#equal,0.33,5)
member("PhysicsWorld").init(member("3dWorld"),vector(1.0,1.0,1.0),#equal,0.33,5)
member("PhysicsWorld").init("3dmembername",vector(1.0,1.0,1.0),#automatic,0,5)
member("PhysicsWorld").init(member("3dWorld"),vector(1.0,1.0,1.0),#automatic,0,5)

//Javascript syntax
member("PhysicsWorld").init("3dmembername",vector(1.0,1.0,1.0),symbol("equal"),0.167,5);
member("PhysicsWorld").init(member("3dWorld"),vector(1.0,1.0,1.0),symbol("equal"),0.33,
5);
member("PhysicsWorld").init("3dmembername",vector(1.0,1.0,1.0),symbol("automatic"),0,5);
member("PhysicsWorld").init(member("3dWorld"),vector(1.0,1.0,1.0),symbol("automatic"),0,
5);
```

See also

destroy(),subSteps,timeStep,timeStepMode

PauseSimulation()

Usage

```
world.PauseSimulation()
```

Description

This pauses the physics simulation for this physics world. Until ResumeSimulation () call is made, the simulate() call will have no effect.

Parameters

None

Example

```
--Lingo Syntax
member("PhysicsWorld").PauseSimulation()

//Javascript syntax
member("PhysicsWorld").PauseSimulation();
```

See also

ResumeSimulation ()

ResumeSimulation ()

Usage

```
<void>world.ResumeSimulation ()
```

Description

This resumes the simulation of the physics world that has been paused by the world.PauseSimulation() call.

Parameters

None

Example

```
--Lingo Syntax
member("PhysicsWorld").ResumeSimulation ()

//Javascript syntax
member("PhysicsWorld").ResumeSimulation ();
```

See also

PauseSimulation()

simulate()

Usage

```
world.simulate()
```

Description

Call steps the physics world by some time. This step time depends on the timeStepMode set during the init call.

timeStepMode can have the following two values:

#equal The simulate() call will step the physics world by time specified in the init call's timeStep parameter.

#automatic The simulate() call will step the physics world by the actual time elapsed.

- All changes to the Physics world are refected only when the simulate() method is called. For example, if you apply impulse using the applyimpulse() method to a rigid body, the value of linearVelocity does not change until you call the simulate() method.
- It is recommended that you call the simulate() method in every exitFrame() for proper simulation.

Note: *A -3 error is displayed when the simulate call fails.*

Parameters

None

Example

```
--Lingo Syntax
member("PhysicsWorld").simulate()

//Javascript syntax
member("PhysicsWorld").simulate();
```

See also

init(),subSteps,timeStep,timeStepMode

Rigid Body management methods

- createRigidBody()
- deleteRigidBody()
- getRigidBody()
- getRigidBodies()
- getSleepingBodies()

Note: *Ensure that the names for the rigid body and terrain are different. When you use the same name for the rigid body and terrain, a -4 error indicating an invalid parameter is returned.*

createRigidBody()

Usage

```
<RigidBody> world.createRigidBody(string rigidbodyname, string
3Dmodelname,symbolBodyProxy,symbol bodyType,symbol flipNormals)
```

Description

This method creates a rigid body with the specified proxy shape.

The #meshdeform modifier must be added to the model before creating a rigid body. A -21 error is displayed when the meshdeform modifier is not added to the 3D model.

Parameters

Parameter	Description
3dModelName	Required. String that specifies the name of the 3d Model on which the rigid body is created.
rigidBodyName	Required. String that specifies the name of the rigid body.

Parameter	Description
symbolBodyProxy	Required. Symbol that takes the following values. #box #Sphere #convexShape #concaveShape
bodyType	Required. #static - If a static or immovable body needs to be created. ***Note:*** When you move a static body by changing its position, the position of the rigid body changes but the position of the 3D model is unaltered. #dynamic - If a movable body needs to be created.
flipNormals	Optional. This parameter inverts the face normals. For bodies using #concaveshape proxy, collisions are detected in the positive direction of the face normals. Penetration is observed from the opposite side. To invert this behavior, use Flip-Normals. This parameter works only with #convexshape and #concaveshape proxies. It throws an error for other proxies.

A reference to the rigid body is returned. If the creation fails, void is returned.

Note: *A rigid body using #concaveshape proxy must be #static. A -28 error is thrown if you try to create a concave dynamic body.*

Example

```
--Lingo syntax
objRigidBody =
member("PhysicsWorld").createRigidBody("RigidBodyA","ModelA",#sphere,#static)
objRigidBody = member("PhysicsWorld").createRigidBody("RigidBodyA","ModelA",#box,#dynamic)
objRigidBody =
member("PhysicsWorld").createRigidBody("RigidBodyA","ModelA",#convexshape,#dynamic)

objRigidBody
=member("PhysicsWorld").createRigidBody("RigidBodyA","ModelA",#concaveshape,#static,#flipN
ormals) --The last parameter,#flipNormals,is optional.

//JavaScript Syntax
var objRigidBody =
member("PhysicsWorld").createRigidBody("RigidBodyA","ModelA",symbol("sphere"),symbol("stat
ic"));
var objRigidBody =
member("PhysicsWorld").createRigidBody("RigidBodyA","ModelA",symbol("box"),symbol("dynamic
"));
var objRigidBody =
member("PhysicsWorld").createRigidBody("RigidBodyA","ModelA",symbol("convexshape"),symbol(
"dynamic'));

var objRigidBody =
member("PhysicsWorld").createRigidBody("RigidBodyA","ModelA",symbol("concaveshape"),symbol
("static"),symbol("flipNormals"));//The last parameter, symbol("flipNormals"),is optional.
```

See also

deleteRigidBody()

deleteRigidBody()

Usage

```
<void>world.deleteRigidBody(string rigidbodyname)
<void>world.deleteRigidBody(RigidBodyref r)
```

Description

Removes the rigid body from the physics world.

Note: *Ensure that you do not delete the 3D model associated with the rigid body before you delete the rigid body.*

Parameters

One of the following parameters is required.

Parameter	Description
Rigidbodyname	String specifying the rigid body name.
Rigibodyref	Reference to the rigid body created using "createrigidbody()"

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").createRigidBody("RigidBodyA","ModelA",#sphere,#static)
member("PhysicsWorld").deleteRigidBody("RigidBodyA")
```

or

```
objRB = member("PhysicsWorld").createRigidBody("RigidBodyA","ModelA",#sphere,#static)
member("PhysicsWorld").deleteRigidBody(objRB);
```

```
//Javascript Syntax
var objRB = member("PhysicsWorld").createRigidBody("RigidBodyA",
"ModelA",symbol("sphere"),symbol("static"));
member("PhysicsWorld").deleteRigidBody("RigidBodyA");
```

or

```
var objRB = member("PhysicsWorld").createRigidBody("RigidBodyA",
"ModelA",symbol("sphere"),symbol("static"));
member("PhysicsWorld").deleteRigidBody(objRB);
```

See also

createRigidBody()

getRigidBody()

Usage

```
<RigidBody>world.getRigidBody(string rigidBodyName)
```

Description

Returns the rigid body specified by the rigidbody name. Return void if the rigid body with the specified name does not exist.

Parameters

Parameter	Description
Rigidbodyname	Required. String specifying the rigid body name.

Returns the reference to the rigid body.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
```

getRigidBodies()

Usage

```
<list of RigidBodies>world.getRigidBodies()
```

Description

Returns a list of rigid bodies currently present in the physics world.

Parameters

None

Returns a list containing reference to the rigid bodies for the physics world.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBodies()

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBodies();
```

getSleepingBodies()

Usage

```
<list of RigidBodies> world.getSleepingBodies()
```

Description

Returns a list of sleeping rigid bodies.

Parameters

None

Returns a list containing reference to the sleeping rigid bodies for the physics world.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getSleepingBodies()

//Javascript Syntax
var objRB = member("PhysicsWorld").getSleepingBodies ();
```

Rigid Body properties

angularDamping

Usage

```
r.angularDamping
```

Access: Get/Set

Type: float

Description

This gets/sets the angular damping for the body. This is the angular counterpart of linear damping.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.angularDamping
```

```
objRB.angularDamping = 0.5

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
put(objRB.angularDamping);
objRB.angularDamping = 0.5;
```

See also

linearDamping

angularMomemtum

Usage

```
r.angularMomemtum
```

Access: Get/Set

Type: vector

Description

This gets/sets the angular momemtum of the rigid body.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.linearVelocity
objRB.angularMomentum = vector(-100,0,0)

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
put(objRB.linearVelocity);
objRB.angularMomentum =  vector(-100,0,0);
```

See also

linearMomemtum

angularVelocity

Usage

```
r.angularVelocity
```

Access: Get/Set

Type: vector

Description

This gets/sets the angular velocity vector of the rigid body.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB. angularVelocity
objRB.angularVelocity = vector(-100,0,0)

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
put(objRB. angularVelocity);
```

```
objRB.angularVelocity =  = vector(-100,0,0);
```

See also

linearVelocity

centerOfMass

Usage

```
r.centerOfMass
```

Access: Get/Set

Type: vector

Descritpion

This represents the center of mass of the rigid body in rigid body coordinates.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.centerOfMass
```

```
//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
put(objRB.centerOfMass);
```

See also

mass

contactTolerance

Usage

```
r.contacttolerance
```

Access: Get/set

Type: float

Description

This sets the inter-penetration depth for the rigid body when it collides with another body.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
objRB.contactTolerance = 0.01
```

```
//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
objRB.contactTolerance = 0.01;
```

friction

Usage

```
r.friction
```

Access: Get/Set

Type: float

Description

This gets/sets the coefficient of friction for the rigid body. The friction property takes values from 0-1.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.friction
objRB.friction= 0.5
```

```
//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody ("RigidBodyA");
put(objRB.friction);
objRB.friction= 0.5;
```

See also

restitution

isPinned

Usage

```
r.ispinned
```

Access: Get/Set

Type: boolean

Description

This gets/sets the pinning of the body. If true , the body is pinned to its position in its current orientation. External forces do not change the position or orientation of the body.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.isPinned
objRB.isPinned = TRUE
```

```
//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
put(objRB.isPinned);
objRB.isPinned = true;
```

isSleeping

Usage

```
r.isSleeping
```

Access: Get/Set

Type: boolean

Description

Returns or sets the state of a rigid body as sleeping.

Note: *A rigid body's state of rest depends on its shape. For example, a rigid body created using a box proxy takes a longer time to sleep compared to a sphere.*

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.isSleeping
```

```
//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody ("RigidBodyA");
put(objRB. isSleeping)
```

linearDamping

Usage

```
r.linearDamping
```

Access: Get/Set

Type: float

Description

This gets/sets the current value of linear damping for the body. This is the amount by which the body slows while moving linearly in the world.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.linearDamping
objRB.linearDamping = 0.5
```

```
//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
put(objRB.linearDamping);
objRB.linearDamping = 0.5;
```

See also

angularDamping

linearMomemtum

Usage

```
r.linearMomemtum
```

Access: Get/Set

Type: vector

Description

This gets/sets the linear momentum of the rigid body.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.linearMomentum
objRB.linearMomentum = vector(-100,0,0)
```

```
//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
put(objRB.linearMomentum);
objRB.linearMomentum =  vector(-100,0,0);
```

See also

angularMomemtum

linearVelocity

Usage

```
r.linearVelocity
```

Access: Get/Set

Type: vector

Description

This gets/sets the linear velocity vector of the rigid body.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.linearVelocity
objRB.linearVelocity = vector(-100,0,0)
```

```
//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
put(objRB. linearVelocity);
objRB.linearVelocity = vector(-100,0,0);
```

See also

angularVelocity

mass

Usage

```
r.mass
```

Access: Get/ Set

Type: float

Description

Sets or gets the mass of the rigid body. The default value for mass is zero. Set a value greater than zero for this property. Not specifying a value will lead to inaccuracies in the simulation.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.mass
objRB.mass = objRB.mass + 20
```

```
//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
put(objRB.mass);
```

```
objRB.mass = objRB.mass + 20;
```

See also

centerOfMass

model

Usage

```
r.model
```

Access: Get

Description

Returns the 3D model that is associated with this rigid body.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
objModel = objRB.model
put objModel.name
```

```
//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
var objModel = objRB.model
put(objModel.name)
```

name

Usage

```
r.name
```

Access: Get

Type: string

Description

Returns the name of the rigid body.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.name
```

```
//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
put(objRB.name);
```

orientation

Usage

```
r.orientation
```

Access: Get/Set

Type: list

Description

Gets/Sets the orientation of the rigid body in the physics world. The orientation property is a list with the direction vector, and the angle between the rigid body's axes and the world axis.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
objRB.orientation = [vector(10,0,0),45]

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
objRB.orientation = list(vector(10,0,0),45);
```

Note: *Changing the position or orientation of a static rigid body will not change the model position. The model position or orientation has to be changed explicitly.*

position

Usage

```
r.position
```

Access: Get/Set

Type: vector

Description

Gets/sets the position of the rigid body in the physics world. The position represents a vector(x,y,z) in world coordinates.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
objRB.position = vector(0,0,0)

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
objRB.position = vector(0,0,0);
```

Note: *Changing the position or orientation of a static rigid body will not change the model position. The model position or orientation has to be changed explicitly.*

properties

Usage

```
r.properties
```

Access: Get

Type: property list

Description

This returns a property list representing the properties of this rigid body. For example , if it is a sphere, it returns the radius; for a box ,the property list contains the dimensions of the box. It returns an empty property list for #concaveshape and #convexshape.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.properties

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
put(objRB.properties);
```

restitution

Usage

```
r.restitution
```

Access: Get/Set

Type: float

Description

This gets/sets the co-efficient of restitution for the rigid body. The restitution property takes values from 0-1.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.restitution
objRB.restitution = 0.5

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
put(objRB.restitution);
objRB.restitution = 0.5;
```

shape

Usage

```
r.shape
```

Access: Get

Values #sphere, #box, #convexshape, or #concaveshape

Description

Returns the shape of the rigid body.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
 put objRB.shape

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
put(objRB.shape);
```

sleepMode

Usage

```
r.sleepMode
```

Access: Get/Set

Description

This gets or sets the sleep mode for a rigid body. This can be either #energy or #linearvelocity. Default value is #energy.

#energy The kinetic energy of the rigid body is used to determine its sleep state.

#linearvelocity The linear velocity of the rigid body is used to determine its sleep state.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
objRB.sleepMode = #energy

//Javascript syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
objRB.sleepMode = symbol("energy");
```

See also

sleepThreshold

sleepThreshold

Usage

```
r.sleepThreshold
```

Access: Get/Set

Type: float

Description

This represents the threshold below which the rigid body becomes inactive. This value is either the kinetic energy or the linear velocity of the rigid body depending on the value set for the sleepMode property. This is only applicable to dynamic objects. Default value is 0.000.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
objRB.sleepMode = #energy
objRB.sleepThreshold = 10.0

//Javascript syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
objRB.sleepMode = symbol("energy");
objRB.sleepThreshold = 10.0;
```

See also

sleepMode

type

Usage

```
r.type
```

Access: Get

Description

Returns the type of the rigid body. This can be either #static or #dynamic.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
put objRB.type

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody ("RigidBodyA");
put(objRB.type);
```

userData

Usage

```
r.userData
```

Access: Get/Set

Type: Object

Description

This allows the user to get/set any user defined data and associate it with this rigid body.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
--Assigning Integer Value
objRB.userData = 0.5
--Assigning String Value
objRB.userData = "My User data Value"
--Assigning Reference Value
objRB.userData = member("PhysicsWorld').getRigidBody("RigidBodyB")

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
--Assigning Integer Value
objRB.userData = 0.5;
--Assigning String Value
objRB.userData = "My User data Value";
--Assigning Reference Value
objRB.userData = member("PhysicsWorld').getRigidBody("RigidBodyB");
```

Rigid Body methods

- applyAngularImpulse()
- applyTorque()
- attemptMoveTo()

applyForce()

Usage

```
<int>r.applyForce(vector vForce, vector vposition)
```

Description

This applies the given force at the position specified by vector pos in rigid body co-ordinates. If the position is not specified, then the force is applied at the center of mass of the rigid body.

Parameters

Parameter	Description
vForce	Required. Vector value that specifies the magnitude and direction of the force applied.
vPosition	Optional. Vector that specifies the point where the force is applied.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
objRB.applyForce(vector(-100,0,0),vector(0,0,0))

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
objRB.applyForce(vector(-100,0,0),vector(0,0,0));
```

applyLinearImpulse()

Usage

```
< int >r.applyLinearImpulse(vector vImpulse, vector vposition)
```

Description

This applies the given Linear Impulse at the position specified by vector pos in rigid body co-ordinates. If the position is not specified, then the impulse is applied at the center of mass of the rigid body.

Parameters

Parameter	Description
vImpulse	Required. Vector value that specifies the magnitude and direction of the force applied.
vPosition	Optional. Vector that specifies the point where the force is applied.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
objRB.applyLinearImpulse(vector(-100,0,0),vector(0,0,0))

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
objRB.applyLinearImpulse(vector(-100,0,0),vector(0,0,0));
```

applyAngularImpulse()

Usage

```
< int >r.applyAngularImpulse(vector vImpulse)
```

Description:

This will apply the given Angular Impulse at the center of mass.

Parameters

Parameter	Description
vImpulse	Required. Vector that specifies the angular Impulse applied.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
objRB.applyAngularImpulse(vector(-100,100,0))

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
objRB.applyAngularImpulse (vector(-100,0,0));
```

applyTorque()

Usage

```
< int >r.applyTorque(vector vTorque)
```

Description

This will apply the specified torque to the rigid body at its center of mass.

Parameters

Parameter	Description
vTorque	Required. Vector that specifies the torque applied.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
objRB.applyTorque(vector(-100,100,0))

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody("RigidBodyA");
objRB.applyTorque(vector(-100,0,0));
```

attemptMoveTo()

Usage

```
< int >r.attemptMoveTo(vector vposition)
```

Description

This call will attempt to move the rigid body to the specified position if there is no collision. If collision is detected at the specified position, then the object will not be moved at all.

Parameters

Parameter	Description
vPosition	Required. Vector that specifies the point to which the object should be moved.

Example

```
--Lingo Syntax
objRB = member("PhysicsWorld").getRigidBody("RigidBodyA")
objRB.attemptMoveTo(vector(-100,0,0))

//Javascript Syntax
var objRB = member("PhysicsWorld").getRigidBody ("RigidBodyA");
objRB.atemptMoveTo(vector(-100,0,0));
```

Constraint methods

ConstraintDesc

Usage

```
ConstraintDesc(sname,ObjectA,ObjectB,vPointA,vPointB,fStiffness,fDamping)
```

vPointA and vPointB must be on or inside the rigid bodies. They are specified in the object coordinates. The behavior is undefined if the points are not on the bodies.

Description

A constraint descriptor object needs to be created before a spring, angular joint, or linear joint is created.

Note: *A -5 error is displayed when you try to create a ConstraintDesc with an existing name.*

Parameters

All constraints are created using a Constraint Descriptor. It is a type with the following contents:

Parameter	Description
string Name	Name of the constraint
ObjectA	Rigidbody A to which the constraint is connected
ObjectB	Rigidbody B to which the constraint is connected.
vector PoCA	Point at which the constraint is attached to Rigidbody A
vector PoCB	Point at which the constraint is attached to Rigidbody B
float Stiffness	Value of the stiffness of the constraint
float damping	Value of the damping of the constraint.

Example

```
--Lingo Syntax
-- Spring constraint Descriptor between two rigid bodies' b1 and b2.
SpringDesc =
ConstraintDesc("SpringDesc",b1,b2,vector(6.0,0.0,0.0),vector(8.0,0.0,0.0),500,1)
-- Linear Joint constraint Descriptor between a rigid body b1 and point in world.
LJointDesc =
ConstraintDesc("LJoinDesc",b1,void,vector(6.0,0.0,0.0),vector(8.0,0.0,0.0),500,1)
```

```
//JavaScript Syntax
//Spring constraint Descriptor between two rigid bodies' b1 and b2.
var SpringDesc =
ConstraintDesc("SpringDesc",b1,b2,vector(6.0,0.0,0.0),vector(8.0,0.0,0.0),500,1);
```

Note:

- To create a constraint between a point in the world and a rigid body, set the value of ObjectB as void.
- A constraint descriptor cannot be created between a point and a static rigid body, between two static rigid bodies, or between two points in world.

See also

`createAngularJoint()`, `createLinearJoint()`, `createSpring()`, `createD6Joint`

createAngularJoint()

Usage

```
<AngularJoint> world.createAngularJoint(ConstraintDesc desc , float length)
```

Description

This creates an angular joint. An angular joint is free to move in the angular direction. It is constrained in the linear direction. The length to be maintained as constrained is specified as the second parameter.

Parameters

Parameter	Description
ConstraintDesc	Required. Reference to the constraint descriptor object.
restlength	Required. Float value that specifies the rest length of the joint.

Example

```
--Lingo Syntax
```

```
-- Angular Joint constraint Descriptor between two rigid bodies' b1 and b2.
AJointDesc =
ConstraintDesc("AJointDesc",b1,b2,vector(6.0,0.0,0.0),vector(8.0,0.0,0.0),500,1)
--Create a angular joint with a rest length of 5.
member("PhysicsWorld").createAngularJoint(AJointDesc, 5.0)
```

```
//JavaScript Syntax
//Angular Joint constraint Descriptor between two rigid bodies' b1 and b2.
var AJointDesc =
ConstraintDesc("AJointDesc",b1,b2,vector(6.0,0.0,0.0),vector(8.0,0.0,0.0),500,1);
//Create a angular Joint with a rest length of 5.
member("PhysicsWorld").createAngularJoint (AJointDesc,5.0);
```

See also

ConstraintDesc, createLinearJoint(), createSpring(), createD6Joint

createLinearJoint()

Usage

```
<LinearJoint> world.createLinearJoint(ConstraintDesc desc , list orientation)
```

Description:

This creates a linear joint. A linear joint is free to move in the linear direction. It is constrained in the angular direction. The angles to be maintained as constrained is specified as the second parameter.

Parameters

Parameter	Description
ConstraintDesc	Required. Reference to the constraint descriptor object.
Orientation	Required. List value that specifies the orientation of the joint.

Example

```
--Lingo Syntax
-- Angular Joint constraint Descriptor between two rigid bodies' b1 and b2.
LJointDesc =
ConstraintDesc("LJointDesc",b1,b2,vector(6.0,0.0,0.0),vector(8.0,0.0,0.0),500,1)
--Create a linear joint with angular constraint in x,y axis at an angle of 45.
member("PhysicsWorld").createLinearJoint(LJointDesc,[vector(1,1,0),45])
```

```
//JavaScript Syntax
//Linear Joint constraint Descriptor between two rigid bodies' b1 and b2.
var LJointDesc =
ConstraintDesc("LJointDesc",b1,b2,vector(6.0,0.0,0.0),vector(8.0,0.0,0.0),500,1);
//Create a linear joint with angular constraint in x,y axis at an angle of 45.
member("PhysicsWorld").createLinearJoint (LJointDesc, list(vector(1,1,0),45));
```

See also

createAngularJoint(), ConstraintDesc, createSpring(), createD6Joint

createSpring()

Usage

```
<Spring> world.createSpring( ConstraintDesc desc, symbol forceExertionMode , float
restLength)
```

Description

Creates a spring with the details specified in the constraint descriptor.

Parameters

Parameter	Description
ConstraintDesc	Required. Reference to the constraint descriptor object.
forceexertionmode	Takes the following symbol values. #kDuringCompression - Restoring force will be applied only when the actual length is less than the rest length between objectA and objectB. #kDuringExpansion - Restoring force will be applied only when the actual length is more than the rest length between objectA and objectB. #kBoth - Restoring force will be applied in both the above cases.
restlength	Required. Float value that specifies the rest length of the spring.

Example

```
--Lingo Syntax
-- Spring constraint Descriptor between two rigid bodies' b1 and b2.
SpringDesc =
ConstraintDesc("SpringDesc",b1,b2,vector(6.0,0.0,0.0),vector(8.0,0.0,0.0),500,1)
--Create a spring which has expansion and compression with a rest length of 5.
member("PhysicsWorld").createSpring(SpringDesc,#kboth,5.0)

//JavaScript Syntax
//Spring constraint Descriptor between two rigid bodies' b1 and b2.
SpringDesc =
ConstraintDesc("SpringDesc",b1,b2,vector(6.0,0.0,0.0),vector(8.0,0.0,0.0),500,1);
//Create a spring which has expansion and compression with a rest length of 5.
member("PhysicsWorld").createSpring(SpringDesc,symbol("kboth"),5.0);
```

See also

createAngularJoint(), createLinearJoint(), ConstraintDesc, createD6Joint

deleteConstraint()

Usage

```
<int>deleteConstraint(string constraintname)
<int>deleteConstraint(Constraint constraintref)
```

Description

This deletes the specified constraint using the name of the constraint.

Parameters

Parameter	Description
ConstraintName	Required. String Name of the constraint.

or

Parameter	Description
Constraintref	Required. Reference to the constraint object.

Example

```
--Lingo Syntax
member("PhysicsWorld").deleteConstraint("ConstraintDesc")

//JavaScript Syntax
member("PhysicsWorld").deleteConstraint("ConstraintDesc");
```

deleteSpring()

Usage

```
<int>deleteSpring(string springname)
<int>deleteSpring(Spring spring)
```

Description

This deletes the specified string using the name of the spring.

Parameters

Parameter	Description
Springname	Required. String value that specifies the name of the created spring.

Example

```
--Lingo Syntax
-- Spring constraint descriptor between two rigid bodies' b1 and b2.
SpringDesc =
ConstraintDesc("SpringDesc",b1,b2,vector(6.0,0.0,0.0),vector(8.0,0.0,0.0),500,1)
--Create a spring which has expansion and compression with a rest length of 5.
member("PhysicsWorld").createSpring(SpringDesc,#kboth,5.0)
--Deletes the spring
member("PhysicsWorld").deleteSpring("SpringDesc")

//JavaScript Syntax
//Spring constraint descriptor between two rigid bodies' b1 and b2.
SpringDesc =
ConstraintDesc("SpringDesc",b1,b2,vector(6.0,0.0,0.0),vector(8.0,0.0,0.0),500,1);
//Create a spring which has expansion and compression with a rest length of 5.
member("PhysicsWorld").createSpring(SpringDesc,symbol("kboth"),5.0);
//Deletes the spring
member("PhysicsWorld").deleteSpring("SpringDesc");
```

getConstraint()

Usage

```
<AngularJoint / LinearJoint>getConstraint(string constraintname)
```

Description

This retuns the constraint of the specified name.

Parameters

Parameter	Description
ConstraintName	Required. String Name of the constraint.

Returns the reference to the joint.

Example

```
--Lingo Syntax
objJoint = member("PhysicsWorld").getConstraint("ConstraintDesc")

//JavaScript Syntax
var objJoint = member("PhysicsWorld").getConstraint("ConstraintDesc");
```

See also

ConstraintDesc

getConstraints()

Usage

```
<list of Constraints> world.getConstraints(symbol constraintType)
```

Description

This returns all the constraints of the specified type. This can either be #linearjoint, #angularjoint, or #d6joint.

Parameters

Parameter	Description
constraintType	Optional. Symbol that specifies the type of constraint. If not provided, will return all constraints, except springs, in the physics world.

Example

```
--Lingo Syntax
--Returns all the constraints in the world.
lstJoint = member("PhysicsWorld").getConstraints()
--Returns all the Linear joint constraints in the world.
lstJoint = member("PhysicsWorld").getConstraints(#linearjoint)

//JavaScript Syntax
var lstJoint = member("PhysicsWorld").getConstraints(symbol("angularjoint"));
```

See also

ConstraintDesc

getSpring()

Usage

```
<Spring > world.getSpring(string springname)
```

Description

Returns the spring object of the specified name.

Parameters

Parameter	Description
Springname	Required. String value that specifies the name of the created spring.

Example

```
--Lingo Syntax
objSpring = member("PhysicsWorld").getSpring("SpringName")
```

```
//JavaScript Syntax
var objSpring = member("PhysicsWorld").getSpring("SpringName");
```

See also

ConstraintDesc

getSprings()

Usage

```
<list of Springs> world.getSprings();
```

Description

This returns a list of all the springs in the physics world.

Collision Functions: By default the collision between all the bodies in the world are enabled. These functions can be used to enable/disable the collision between pairs of bodies.

Parameters

None

Example

```
--Lingo Syntax
lSprings= member("PhysicsWorld").getSprings()
```

```
//JavaScript Syntax
var lSprings = member("PhysicsWorld").getSprings();
```

Constraint properties

- constraintType
- damping
- name
- objectA
- objectB
- pointA
- pointB
- properties
- stiffness

constraintType

Usage

```
c.constraintType
```

Access: Get

Description

Returns the type of the constraint. This can be #linearjoint, #angularjoint, or #6djoint.

Example

```
--Lingo Syntax
objJoint = member("PhysicsWorld").getConstraint("lJoint01")
put objJoint.constraintType
```

```
//JavaScript Syntax.
objJoint = member("PhysicsWorld").getConstraint("lJoint01");
put(objJoint.constraintType);
```

damping

Usage

```
c.damping
```

Access: Get / Set

Type: float

Description

Returns the damping of the constraint.

Example

```
--Lingo Syntax
objJoint = member("PhysicsWorld").getSpring("lJoint01")
put objJoint.damping
```

```
//JavaScript Syntax.
objJoint = member("PhysicsWorld").getSpring("lJoint01");
put(objJoint.damping);
```

name

Usage

```
c.name
```

Access: Get

Type: string

Description

Returns the name of the constraint.

Example

```
--Lingo Syntax
objConstraint = member("PhysicsWorld").getConstraint("ConstraintA")
put objConstraint.name
```

```
//Javascript Syntax
var objConstraint = member("PhysicsWorld").getConstraint("ConstraintA");
put(objConstraint.name);
```

objectA

Usage

```
s.objectA
```

Access: Get

Type: RigidBody

Description

Returns the name of rigid body "A" attached to the joint.

Example

```
--Lingo Syntax
--Change one end of the Joint to already created rigidbody - rigidbodyX
objJoint = member("PhysicsWorld").getConstraint("lJoint01")
put objJoint.objectA
```

```
//JavaScript Syntax.
--Change one end of the Joint to already created rigidbody - rigidbodyX
objJoint = member("PhysicsWorld").getConstraint("lJoint01");
put(objJoint.objectA);
```

objectB

Usage

```
s.objectB
```

Access: Get

Type: RigidBody

Description

Returns the name of rigid body "B" attached to the spring

Example

```
--Lingo Syntax
--Change one end of the Joint to already created rigidbody - rigidbodyX
objJoint = member("PhysicsWorld").getConstraint ("lJoint01")
put objJoint.objectB
--Change one end of the Joint to a worldPoint
objJoint = member("PhysicsWorld").getConstraint("lJoint01")
put objJoint.objectB
```

```
//JavaScript Syntax.
--Change one end of the joint to already created rigidbody - rigidbodyX
objJoint = member("PhysicsWorld").getConstraint("lJoint01");
put(objJoint.objectB);
```

pointA

Usage

```
s.pointA
```

Access: Get

Type: vector

Description

Returns the point at which the joint is attached to object A.

Example

```
--Lingo Syntax
objJoint = member("PhysicsWorld").getConstraint("lJoint01")
put objJoint.pointA

//JavaScript Syntax.
objJoint = member("PhysicsWorld").getConstraint("lJoint01");
put(objJoint.pointA);
```

pointB

Usage

```
s.pointB
```

Access: Get

Type: vector

Description

Returns the point at which the joint is attached to object B.

Example

```
--Lingo Syntax
objJoint = member("PhysicsWorld").getConstraint("lJoint01")
put objJoint.pointB

//JavaScript Syntax.
objJoint = member("PhysicsWorld").getConstraint("lJoint01");
put(objJoint.pointB);
```

properties

Usage

```
c.properties
```

Access: Get / Set

Type: Property List

Description

Returns a property list representing the properties of the joint. For example, the linear joint returns orientation, and the angular joint returns length.

Example

```
--Lingo Syntax
objJoint = member("PhysicsWorld").getConstraint("lJoint01")
put objJoint.properties

//JavaScript Syntax.
objJoint = member("PhysicsWorld").getConstraint("lJoint01");
put(objJoint.properties);
```

stiffness

Usage

```
s.stiffness
```

Access: Get /Set

Type: float

Description

Returns the stiffness of the constraint.

Example

```
--Lingo Syntax
objJoint = member("PhysicsWorld").getSpring("lJoint01")
put objJoint.stiffness
```

```
//JavaScript Syntax.
objJoint = member("PhysicsWorld").getSpring("lJoint01");
put(objJoint.stiffness);
```

Spring properties

damping

Usage

```
s.damping
```

Access: Get/Set

Type: float

Description

Represents the damping of the spring.

Example

```
--Lingo Syntax
objSpring = member("PhysicsWorld").getSpring("SpringDesc")
put objSpring.damping
```

```
//JavaScript Syntax
var objSpring = member("PhysicsWorld").getSpring("SpringDesc");
put(objSpring.damping);
```

flags

Usage

```
s.flags
```

Access: Get/Set

Description

Represents the compression/expansion forces applied to the spring. Returns either #kDuringCompression , #kDuringExpansion or #kBoth

Example

```
--Lingo Syntax
objSpring = member("PhysicsWorld").getSpring("SpringDesc")
put objSpring.flags
```

```
//JavaScript Syntax
var objSpring = member("PhysicsWorld").getSpring("SpringDesc");
put(objSpring.flags);
```

name

Usage

```
s.name
```

Access: Get

Type: string

Description

Returns the name of the spring.

Example

```
--Lingo Syntax
objSpring = member("PhysicsWorld").getSpring("SpringA")
put objSpring.name
```

```
//Javascript Syntax
var objSpring = member("PhysicsWorld").getSpring("SpringA");
put(objSpring.name);
```

objectA

Usage

```
s.objectA
```

Access: Get/Set

Type: RigidBody

Description

Returns the name of rigid body "A" attached to the spring

Example

```
--Lingo Syntax
--Change one end of the spring to already created rigidbody - rigidbodyX
objSpring = member("PhysicsWorld").getSpring("Spring01")
put objSpring.objectA
objSpring.objectA = rigidbodyX
--Change one end of the spring to a worldPoint
objSpring = member("PhysicsWorld").getSpring("Spring01")
put objSpring.objectA
objSpring.objectA = void
```

```
//JavaScript Syntax.
--Change one end of the spring to already created rigidbody - rigidbodyX
objSpring = member("PhysicsWorld").getSpring("Spring01");
put(objSpring.objectA);
objSpring.objectA = rigidbodyX;
```

objectB

Usage

```
s.objectB
```

Access: Get/set

Type: RigidBody

Description

Returns the name of rigid body "B" attached to the spring.

Example

```
--Lingo Syntax
--Change one end of the spring to already created rigidbody - rigidbodyX
objSpring = member("PhysicsWorld").getSpring("Spring01")
put objSpring.objectB
objSpring.objectB = rigidbodyX

--Change one end of the spring to a worldPoint
objSpring = member("PhysicsWorld").getSpring("Spring01")
put objSpring.objectB
objSpring.objectB = void
```

```
//JavaScript Syntax.
--Change one end of the spring to already created rigidbody - rigidbodyX
objSpring = member("PhysicsWorld").getSpring("Spring01");
put(objSpring.objectB);
objSpring.objectB = rigidbodyX;
```

pointA

Usage

```
s.pointA
```

Access: Get/Set

Type: vector

Description

Returns the point at which the spring is attached to object A.

Example

```
--Lingo Syntax
objSpring = member("PhysicsWorld").getSpring("Spring01")
put objSpring.pointA

//JavaScript Syntax.
objSpring = member("PhysicsWorld").getSpring("Spring01");
put(objSpring.pointA);
```

pointB

Usage

```
s.pointB
```

Access: Get/Set

Type: vector

Description

Returns the point at which the spring is attached to object B.

Example

```
--Lingo Syntax
objSpring = member("PhysicsWorld").getSpring("Spring01")
put objSpring.pointB

//JavaScript Syntax.
objSpring = member("PhysicsWorld").getSpring("Spring01");
put(objSpring.pointB);
```

restLength

Usage

```
s.restLength
```

Access: Get/Set

Type: float

Description

Represents the length of the spring at rest.

Example

```
--Lingo Syntax
objSpring = member("PhysicsWorld").getSpring("SpringDesc")
put objSpring.restLength

//JavaScript Syntax
var objSpring = member("PhysicsWorld").getSpring("SpringDesc");
put(objSpring.restLength);
```

stiffness

Usage

```
s.stiffNess
```

Access: Get/Set

Type: float

Description

Represents the stiffness of the spring.

Example

```
--Lingo Syntax
objSpring = member("PhysicsWorld").getSpring("SpringDesc")
put objSpring.stiffness
```

```
//JavaScript Syntax
var objSpring = member("PhysicsWorld").getSpring("SpringDesc");
put(objSpring.stiffness);
```

Collision and Collision Callback methods

- disableCollision()
- disableCollisionCallback()
- enableCollision()
- enableCollisionCallback()
- getCollisionCallbackDisabledPairs()
- getCollisionDisabledPairs()
- registerCollisionCallback()
- removeCollisionCallback()

disableCollision()

Usage

```
<int>world .disableCollision(RigidBody a, RigidBody b)
<int>world .disableCollision(string rigidbodyname1, string rigidbodyname2)
```

Description

If A, B are void, this will disable collision between all rigid bodies in the world.

If B is void, then it will disable collision of body A with all the bodies.

If A, B are specified, it will disable collision between these two bodies if they had been enabled earlier.

Note: *disableCollision( ) disables all the callbacks.. However, using enableCollision( ) will not enable the callbacks automatically. You will have to call enableCollisionCallback( ) separately for enabling the callbacks.*

Note:

Parameters

Depends on the number of rigid bodies for which collision has to be disabled.

Example

```
--Lingo Syntax
--All collisions are disabled.
member("PhysicsWorld").disableCollision()
```

```
--Collisions disabled for rigid body A
member("PhysicsWorld").disableCollision(rigidbodyA)
--Collisions disabled for rigidbody A and rigidbody B
member("PhysicsWorld").disableCollision(rigidbodyA,rigidbodyB)

//JavaScript Syntax
//All collisions are disabled.
member("PhysicsWorld").disableCollision();
//Collisions disabled for rigid body A
member("PhysicsWorld").disableCollision(rigidbodyA);
//Collisions disabled for rigidbody A and rigidbody B
member("PhysicsWorld").disableCollision(rigidbodyA,rigidbodyB);
```

See also

enableCollision()

disableCollisionCallback()

Usage

```
<int>world .disableCollisionCallback(RigidBody a, RigidBody b)
<int>world . disableCollisionCallback (string rigidbodyname1, string rigidbodyname2)
```

Description

If no parameters are passed, collision callback reports for all rigid bodies that were created before this call are disabled.

If parameters are passed to this method, collision callbacks are disabled only for the rigid body(s) specified.

Parameters

Depends on the number of rigid bodies for which collision callbacks have to be disabled.

Example

```
--Lingo Syntax
--All collisions callbacks are disabled for rigid bodies created before this call.
member("PhysicsWorld").disableCollisionCallback()
--Collisions callback disabled for rigid body A with all other bodies.
member("PhysicsWorld").disableCollisionCallback(rigidbodyA)
--Collisions  callback disabled between rigidbody A and rigidbody B.
member("PhysicsWorld").disableCollisionCallback(rigidbodyA,rigidbodyB)

//JavaScript Syntax
//All collision callbacks are disabled.
member("PhysicsWorld").disableCollisionCallback();
//Collision callbacks disabled for rigid body A
member("PhysicsWorld").disableCollisionCallback(rigidbodyA);
//Collision callbacks disabled between rigidbody A and rigidbody B
member("PhysicsWorld").disableCollisionCallback(rigidbodyA,rigidbodyB);
```

See also

enableCollisionCallback(),getCollisionCallbackDisabledPairs(),getCollisionDisabledPairs(),
registerCollisionCallback(),removeCollisionCallback()

enableCollision()

Usage

```
<int>world .enableCollision(RigidBody a, RigidBody b)
```

```
<int>world .enableCollision(string rigidbodyname1, string rigidbodyname2)
```

Description

If no parameters are passed, all collisions are enabled. Otherwise, collisions are enabled only for the rigid body(s) specified.

Parameters

Depends on the number of rigid bodies for which collision has to be enabled.

Example

```
--Lingo Syntax
--All collisions are enabled.
member("PhysicsWorld").enableCollision()
--Collisions enabled for rigid body A with all other bodies.
member("PhysicsWorld").enableCollision(rigidbodyA)
--Collisions enabled between rigidbody A and rigidbody B.
member("PhysicsWorld").enableCollision(rigidbodyA,rigidbodyB)

//JavaScript Syntax
//All collisions are enabled.
member("PhysicsWorld").enableCollision();
//Collisions enabled for rigid body A
member("PhysicsWorld").enableCollision(rigidbodyA);
//Collisions enabled for rigidbody A and rigidbody B
member("PhysicsWorld").enableCollision(rigidbodyA,rigidbodyB);
```

See also

disableCollision()

enableCollisionCallback()

Usage

```
<int>world .enableCollisionCallback(RigidBody a, RigidBody b)
<int>world . enableCollisionCallback(string rigidbodyname1, string rigidbodyname2)
```

Description

If no parameters are passed, collision callback reports for all rigid bodies that were created before this call is enabled.

If parameters are passed to this method, collision callbacks are enabled only for the rigid body(s) specified.

Parameters

Depends on the number of rigid bodies for which collision callbacks has to be enabled.

Example

```
--Lingo Syntax
--All collisions callbacks are enabled for rigid bodies created before this call.
member("PhysicsWorld").enableCollisioncallback()
--Collisions callback enabled for rigid body A with all other bodies.
member("PhysicsWorld").enableCollisioncallback(rigidbodyA)
--Collisions  callback enabled between rigidbody A and rigidbody B.
member("PhysicsWorld").enableCollisioncallback(rigidbodyA,rigidbodyB)

//JavaScript Syntax
//All collision callbacks are enabled.
member("PhysicsWorld").enableCollisioncallback();
//Collision callbacks enabled for rigid body A
member("PhysicsWorld").enableCollisioncallback(rigidbodyA);
```

```
//Collision callbacks enabled for rigidbody A and rigidbody B
member("PhysicsWorld").enableCollisioncallback(rigidbodyA,rigidbodyB);
```

See also

disableCollisionCallback(),getCollisionDisabledPairs(),getCollisionCallbackDisabledPairs()
,registerCollisionCallback(),removeCollisionCallback()

getCollisionCallbackDisabledPairs()

Usage

```
<[[RigidBody,RigidBody]…]>world .getCollisionCallbackDisabledPairs()
```

Description

This returns the list of all pairs of rigid bodies for which collision callbacks have been disabled.

Parameters

None

Example

```
--Lingo Syntax
lstPairs = member("PhysicsWorld").getCollisionCallbackDisabledPairs()
lstPair1 = lstPairs[1]

//JavaScript Syntax
var lstPairs = member("PhysicsWorld").getCollisionCallbackDisabledPairs();
var lstPair1 = lstPairs(0);
```

See also

enableCollisionCallback(),disableCollisionCallback(),getCollisionDisabledPairs(),registerC
ollisionCallback(),removeCollisionCallback()

getCollisionDisabledPairs()

Usage

```
<[[RigidBody,RigidBody]…]>world .getCollisionDisabledPairs()
```

Description

This returns the list of all pairs of rigid bodies for which collision has been disabled.

Parameters

None

Example

```
--Lingo Syntax
lstPairs = member("PhysicsWorld").getCollisionDisabledPairs()
lstPair1 = lstPairs[1]

//JavaScript Syntax
var lstPairs = member("PhysicsWorld").getCollisionDisabledPairs();
var lstPair1 = lstPairs(0);
```

See also

enableCollisionCallback(),disableCollisionCallback(),getCollisionCallbackDisabledPairs(),
registerCollisionCallback(),removeCollisionCallback()

registerCollisionCallback()

Usage

```
<int>world .registerCollisionCallback( #collisionCallback,<script reference>)
```

Description

This function registers the callback function that is to be called to report the collisions. There can be only one collision callback handler for the physics scene. The information about the collisions is passed to this function in a Collision Report list, which contains the following information:

Parameters

Parameter	Description
CollisionCallback	Required. This specifies the name of the collision callback handler.
Script reference	Optional. This is a behavior or parent script instance. If this parameter is not provided, it is assumed that the callback function is defined in a movie script.

Structure of a Collision Report:

ObjectA	Object A involved in collision
ObjectB	Object B involved in collision
list (vector<float,float,float>)	The contact points between the object A and object B
list (vector<float,float,float>)	The contact normals for each of the above points.

Note: *Inside the collision callback handler, if there is any script that causes runtime errors (such as property not found), the error messages are not displayed, and the handler execution is aborted at the point of error. All the subsequent statements are not executed.*
Avoid modifications to the 3D scene within the collision callback handler, like resetworld(), deleteModel(), deleterigidBody(),etc.

Example

```
--Lingo Syntax
member("PhysicsWorld").registercollisionCallback(#collisioncallback)
--Movie Script
On collisionCallback collisionReport
CollisionA = collisionReport[1]
R1 = CollisionA.objectA
R2 = CollisionA.objectB
lstPoints = CollisionA.contactPoints
lstNormals = CollisionA.contactNormals
end

//JavaScript Syntax
member("PhysicsWorld").registerCallback(symbol("collisioncallback"));
//Movie Script
function collisionCallback(collisionreport)
{
var CollisionA = collisionReport(0);
var R1 = CollisionA.objectA;
var R2 = CollisionA.objectB;
var lstPoints = CollisionA.contactPoints;
var lstNormals = CollisionA.contactNormals;
}
```

See also

enableCollisionCallback(),disableCollisionCallback(),getCollisionDisabledPairs(),getCollisionDisabledPairs(),removeCollisionCallback()

removeCollisionCallback()

Usage

```
<int>world .removeCollisionCallback()
```

Description:

This function removes the callback registered earlier.

Parameters

None

Example

```
--Lingo Syntax
member("PhysicsWorld").removeCollisionCallback()

//JavaScript Syntax
member("PhysicsWorld").removeCollisionCallback();
```

See also

enableCollisionCallback(),disableCollisionCallback(),getCollisionDisabledPairs(),getCollisionCallbackDisabledPairs(),registerCollisionCallback()

Terrain methods

- createTerrain
- createTerrainDesc
- deleteTerrain
- getTerrain
- getTerrains

Note: *Ensure that the names for the rigid body and terrain are different. When you use the same name for the rigid body and terrain, a -4 error indicating an invalid parameter is returned.*

createTerrain

Usage

```
createTerrain(terrainName, terrainDesc, position, orientation, rowScale, columnScale, heightScale)
```

Description

This method creates the terrain as a static rigid body using the height-map information provided.

Note: *For better performance while creating a mesh and heightMap using a heightmap image , breaking the mesh into smaller pieces and stitching them back will help boost performance.*

Parameters

Parameter	Description
terrainName	Required. String that is the name of the terrain.
terrainDesc	Required. Reference to the terrain descriptor.
position	Required. Vector that sets the position of the terrain.
orientation	Required. Vector that sets the orientation of the terrain.
rowScale	Required. float value that specifies the row scale factor.
columnScale	Required. float value that specifies the column scale factor.
heightScale	Required. Float value that specifies the height scale factor.

Example

```
--Lingo Syntax
objTerrain =
member("PhysicsWorld").createTerrain("myterrain",terrainDesc,position,orientation,1,1,1)

//JavaScript Syntax
var objTerrain=
member("PhysicsWorld").createTerrain("myterrain",terrainDesc,position,orientation,1,1,1);
```

See also

createTerrainDesc,terrainDesc

createTerrainDesc

Usage

```
createTerrainDesc(elevationMatrix,friction,restitution)
```

Description

A terrain descriptor object needs to be created before a terrain is created.

Parameters

Parameter	Description	Values
ElevationMatrix	Required. Lingo Matrix that represents the height map information of the terrain.	
Friction	Required. Float value that represents the coefficient of friction.	0-1
Restitution	Required. Float value that represents the coeffficient of restitution.	0-1

Example

```
--Lingo Syntax
tDesc = member("PhysicsWorld").createTerrainDesc(myMatrix,0.4,0.5)

//JavaScript Syntax
//Spring constraint Descriptor between two rigid bodies' b1 and b2.
var tDesc = member("PhysicsWorld").createTerrainDesc(myMatrix,0.4,0.5);
```

See also

createTerrain,terrainDesc

deleteTerrain

Usage

```
world.deleteterrain(String terrainName)
world.deleteterrain(terrain refTerrain)
```

Description

This method deletes the terrain from the physics scene. The method can take the terrain name or reference as the parameter.

Parameters

Parameter	Description
terrainName	Required. String value that specifies the name of the created terrain.

or

Parameter	Description
refTerrain	required. Reference to the created terrain object.

Example

```
--Lingo Syntax
member("PhysicsWorld").deleteTerrain("myterrain")
```

```
//JavaScript Syntax
member("PhysicsWorld").deleteTerrain("myTerrain");
```

getTerrain

Usage

```
<terrain refTerrain> world.getterrain(String "myterrain")
```

Description

This method returns the terrain object of the specified name.

Parameters

Parameters	Description
myTerrain	: required. String that specifies the name of the terrain.

Example

```
--Lingo Syntax
objTerrain = member("PhysicsWorld").getTerrain("myTerrain")
```

```
//JavaScript Syntax
var objTerrain = member("PhysicsWorld").getTerrain("myTerrain");
```

See also

createTerrainDesc,terrainDesc

getTerrains

Usage

```
<list lstTerrain> world.getterrains()
```

Description

This method returns the list of terrain objects in the physics world.

Parameters

No parameter.

Example

```
--Lingo Syntax
lstTerrain = member("PhysicsWorld").getTerrains()
```

```
//JavaScript Syntax
var lstTerrain = member("PhysicsWorld").getTerrains();
```

Terrain properties

- columnScale
- contactTolerance
- heightScale
- name
- orientation
- position
- rowScale
- terrainDesc

columnScale

Syntax

```
t.columnScale
```

Access: Get

Description

Represents the column scale factor of the terrain.

Example

```
--Lingo Syntax
objTerrain = Member("PhysicsWorld").getTerrain("myTerrain")
put objTerrain.columnScale
```

```
//JavaScript Syntax
Var objTerrain = Member("PhysicsWorld").getTerrain("myTerrain");
Put(objTerrain.columnScale);
```

contactTolerance

Syntax

```
t.contactTolerance
```

Access: Get/set

Type: float

Description

This is the penetration depth between the terrain and other rigid body/terrain for collision to be detected.

Example

```
--Lingo Syntax
objTerrain = Member("PhysicsWorld").getTerrain("myTerrain")
put objTerrain.contactTolerance
```

```
//JavaScript Syntax
Var objTerrain = Member("PhysicsWorld").getTerrain("myTerrain");
Put(objTerrain.contactTolerance);
```

heightScale

Syntax

```
t.heightScale
```

Access: Get

Description

Represents the height scale factor of the terrain.

Example

```
--Lingo Syntax
objTerrain = Member("PhysicsWorld").getTerrain("myTerrain")
put objTerrain.heightScale
```

```
//JavaScript Syntax
Var objTerrain = Member("PhysicsWorld").getTerrain("myTerrain");
Put(objTerrain.heightScale);
```

name

Usage

```
t.name
```

Access: Get

Type: string

Description

Returns the name of the terrain.

Example

```
--Lingo Syntax
put objTerrain.name
```

```
//Javascript Syntax
```

```
put(objTerrain.name);
```

orientation

Syntax

```
t.orientation
```

Access: Get

Description

List that represents the orientation of the terrain.

Example

```
--Lingo Syntax
objTerrain = Member("PhysicsWorld").getTerrain("myTerrain")
put objTerrain.orientation
```

```
//JavaScript Syntax
Var objTerrain = Member("PhysicsWorld").getTerrain("myTerrain");
Put(objTerrain.orientation);
```

position

Syntax

```
t.position
```

Access: Get

Description

Vector that represents the position of the terrain.

Example

```
--Lingo Syntax
objTerrain = Member("PhysicsWorld").getTerrain("myTerrain")
put objTerrain.position
```

```
//JavaScript Syntax
Var objTerrain = Member("PhysicsWorld").getTerrain("myTerrain");
Put(objTerrain.position);
```

rowScale

Syntax

```
t.rowScale
```

Access: Get

Description: represents the row scale factor of the terrain.

Example

```
--Lingo Syntax
objTerrain = Member("PhysicsWorld").getTerrain("myTerrain")
put objTerrain.rowScale
```

```
//JavaScript Syntax
Var objTerrain = Member("PhysicsWorld").getTerrain("myTerrain");
```

```
Put(objTerrain.rowScale);
```

terrainDesc

Syntax

```
t.terrainDesc
```

Access: Get

Description

Represents the terrain descriptor object.

The terrain descriptor has the following attributes:

- terraindesc.elevationmatrix
- terraindesc.friction
- terraindesc.restitution
- terraindesc.numrows
- terraindesc.numcolumns

Note: *numrows and numcolumns correspond to the rows and columns of the elevation matrix.*

Example

```
--Lingo Syntax
objTerrain = Member("PhysicsWorld").getTerrain("myTerrain")
put objTerrain.terrainDesc

//JavaScript Syntax
Var objTerrain = Member("PhysicsWorld").getTerrain("myTerrain");
Put(objTerrain.terrainDesc);
```

See also

createTerrainDesc,createTerrain

6 DOF joint methods

createD6Joint

The order in which rigidbodies are provided as input to a 6DOF joint is important with respect to the set joint axis and the drive values. For example, if the order is rb1,rb2 and the drive-position is vector(10,0,0), the direction in which the rigid body will move to attain the position is exactly opposite to the direction it would have moved if the order was rb2,rb1.

createD6Joint

Usage

```
<d6joint> world.createD6Joint(jointName, RigidBody rb1 ,Rigidbody rb2, vector globalanchor)
```

Description

This method creates a 6DOF joint between two rigid bodies or between a point in world and a rigid body.

Note: *D6Joints are also a type of constraint like the linear and angular joints. The constraintType for D6Joint is #d6joint.*

Use deleteConstraint() to delete the D6Joint.

The additional set of properties specific to the D6Joint are mentioned in the D6Joint properties section.

Parameters

Parameter	Description
jointName	Required. String that specifies the name of the joint.
rb1	Required. Reference to the rigid body participating in the joint. If a joint has to be created between a point and rigid body, then this parameter should be void.
rb2	Required. Reference to the rigid body participating in the joint.
globalAnchor	Required. Vector that specifies the anchor point of the joint.

Example

```
--Lingo Syntax
--Creates a joint between rigid bodies rb1 and rb2.
ObjJoint = member("PhysicsWorld").createD6Joint("myJoint",rb1,rb2,vector(1,1,1))
-- Creates a joint between the anchor point and rb2.
objJoint = member("PhysicsWorld").createD6Joint("myJoint",void,rb2,vector(1,1,1))
```

```
//JavaScript Syntax
//Creates a joint between rigid bodies rb1 and rb2.
var objJoint = member("PhysicsWorld").createD6Joint("myJoint",rb1,rb2,vector(1,1,1));
```

See also

`createAngularJoint()`, `createLinearJoint()`, `createSpring()`, `ConstraintDesc`, `globalAnchor`

6DOF properties

- axisDrive
- axisMotion
- biNormalDrive
- biNormalMotion
- constraintType
- driveAngularVelocity
- driveLinearVelocity
- driveOrientation
- drivePosition
- globalAnchor
- linearLimit
- localAnchorA
- localAnchorB
- localAxisA

- localAxisB
- localNormalA
- localNormalB
- name
- normalDrive
- normalMotion
- objectA
- objectB
- swing1Limit
- swing1Motion
- swing2Limit
- swing2Motion
- swingDrive
- twistDrive
- twistLimit
- twistMotion

axisDrive

Syntax

```
d6joint.axisDrive = [type, spring, damping, forceLimit]
```

Access: Get/Set

Description

List value that specifies the linear drive for the joint, which will drive the joint to the specified position along the joint's axis.

The list contains the following attributes:

type #position or #velocity

stiffness the spring to be applied while driving (> 0)

damping the damping of the spring (> 0)

forceLimit The force or torque with which to drive to the specified position or velocity (> 0)

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.axisDrive = [#position,100,0.1,100]

//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.axisDrive = [#position,100,0.1,100];
```

See also

biNormalDrive,normalDrive,swingDrive,twistDrive,driveAngularVelocity,driveLinearVelocity,driveOrientation,drivePosition

axisMotion

Syntax

```
D6joint.axisMotion
```

Access: Get/Set

Description

Symbol that sets the linear degree of freedom along the joint Axis.

This property takes the following values

#Locked There can be no movement along this DOF

#Limited There can be a limited movement along this DOF

#Free There is no restriction for movement along this DOF

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
put objJoint.axisMotion
```

```
//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
Put(objJoint.axisMotion);
```

See also

biNormalMotion,normalMotion,swing1Motion,swing2Motion,twistMotion

biNormalDrive

Syntax

```
d6joint.binormalDrive = [type, spring, damping, forceLimit]
```

Access: Get/Set

Description

List value that specifies the linear drive for the joint, which will drive the joint to the specified position along the joint's binormal axis.

The list contains the following attributes:

type #position or #velocity

stiffness the spring to be applied while driving (> 0)

damping the damping of the spring (> 0)

forceLimit The force or torque with which to drive to the specified position or velocity (> 0)

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
```

```
objJoint.binormalDrive = [#position,100,0.1,100]

//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.binormalDrive = [#position,100,0.1,100];
```

See also

normalDrive, swingDrive, twistDrive, driveAngularVelocity, driveLinearVelocity, driveOrientation, drivePosition, axisDrive

biNormalMotion

Syntax

```
D6joint.binormalMotion
```

Access: Get/Set

Description

Symbol that sets the linear degree of freedom along the joint's binormal Axis.

This property takes the following values

#Locked There can be no movement along this DOF

#Limited There can be a limited movement along this DOF

#Free There is no restriction for movement along this DOF

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
put objJoint.biNormalMotion

//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
Put(objJoint.biNormalMotion);
```

See also

normalMotion, swing1Motion, swing2Motion, twistMotion, axisMotion

constraintType

Syntax

```
d6joint.constraintType
```

Access: Get

Description

D6Joints are a type of constraint like the linear and angular joints. The constraintType for D6Joint is #d6joint.

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
put objJoint.constraintType

//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
put(objJoint.constraintType);
```

driveAngularVelocity

Syntax

```
d6joint.driveAngularVelocity = vector velocity
```

Access: Get/Set

Description

Vector value that specifies the intended angular velocity when the drivetype for swingDrive or twistDrive is specified as #velocity.

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.driveAngularVelocity = vector(20,10,0)

//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.driveAngularVelocity = vector(20,10,0);
```

See also

normalDrive, swingDrive, twistDrive, driveLinearVelocity, driveOrientation, drivePosition, axisDrive, biNormalDrive

driveLinearVelocity

Syntax

```
d6joint.driveLinearVelocity = vector velocity
```

Access: Get/Set

Description

Vector value that specifies the intended linear velocity when the drivetype for axisDrive, normalDrive or binormalDrive is #velocity.

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.driveLinearVelocity = vector(20,10,0)

//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.driveLinearVelocity = vector(20,10,0);
```

See also

normalDrive, swingDrive, twistDrive, driveAngularVelocity, driveOrientation, drivePosition, axisDrive, biNormalDrive

driveOrientation

Syntax

```
d6joint.driveOrientation = list orientation
```

Access: Get/Set

Description

List value that specifies the goal orientation when the drivetype for swingDrive or twistDrive is specified as #position.

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.driveOrientation = [vector(20,10,0),45]

//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.driveOrientation = [vector(20,10,0),45];
```

See also

normalDrive, swingDrive, twistDrive, driveLinearVelocity, driveAngularVelocity, drivePosition, axisDrive, biNormalDrive

drivePosition

Syntax

```
d6joint.drivePosition = vector position
```

Access: Get/Set

Description

Vector value that specifies the goal position when the drivetype for axisDrive, normalDrive and binormalDrive is specified as #position

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.drivePosition = vector(20,10,0)

//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.drivePosition = vector(20,10,0);
```

See also

normalDrive, swingDrive, twistDrive, driveLinearVelocity, driveAngularVelocity, driveOrientation, axisDrive, biNormalDrive

globalAnchor

Syntax

```
d6joint.globalAnchor = vector position
```

Access: Get/Set

Description

Vector value that specifies the anchor point for the joint.

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.globalAnchor = vector(20,10,0)
```

```
//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.globalAnchor = vector(20,10,0);
```

See also

localAnchorA, localAnchorB

linearLimit

Syntax

```
D6joint.linearlimit = [limitvalue, stiffness, damping, restitution]
```

Access: Get/Set

Description

List that specifies the joint behavior when the linear motion is limited. The rigid body will oscillate when it hits the limit value. This is applicable to all the linear motions.

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.linearLimit =  [2, 100, 0.01, 0]
```

```
//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.linearLimit = [2, 100, 0.01, 0];
```

Note: *Set the limit values first and then set the motion parameters[#free,#limited etc] for the limits to take effect.*

See also

swing1Limit, swing2Limit, twistLimit

localAnchorA

Syntax

```
D6joint.localAnchorA
```

Access: Get/Set

Description

Attachment point of joint in objectA's space.

The values for the properties localAxisA, localAxisB, localNormalA, localNormalB, localAnchorA, localAnchorB are available only after you set the values explicitly for these properties. If the value is not set, a void vector is returned.

Note: *For a stable joint, provide consistent values to localAnchorA and localAnchorB so that in the world space both of them correspond to the same point.*

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.localAnchorA = vector(0,5,0)
```

```
//JavaScript Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint");
```

```
objJoint.localAnchorA = vector(0,5,0);
```

See also

localAnchorB,globalAnchor

localAnchorB

Syntax

```
D6joint.localAnchorB
```

Access: Get/Set

Description

Attachment point of joint in objectB's space.

The values for the properties localAxisA, localAxisB, localNormalA, localNormalB, localAnchorA, localAnchorB are available only after you set the values explicitly for these properties. If the value is not set, a void vector is returned.

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.localAnchorB = vector(0,-5,0)

//JavaScript Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint");
objJoint.localAnchorB = vector(0,-5,0);
```

See also

localAnchorA,globalAnchor

localAxisA

Syntax

```
D6joint.localAxisA
```

Access: Get/Set

Description

This is the primary axis of the joint in terms of the local coordinate space of objectA of the the D6Joint. localAxisA and localNormalA should be at right angles to each other.

When a rigid body is static, the local axes set for the body are considered global because they cannot be oriented along the joint axes.

The values for the properties localAxisA, localAxisB, localNormalA, localNormalB, localAnchorA, localAnchorB are available only after you set the values explicitly for these properties. If the value is not set, a void vector is returned.

Note:

- Joint's Primary Axis is the axis around which the twist happens, along which AxisMotion will happen.
- Joint's Normal Axis is the axis around which the swing1 happens, along which NormalMotion will happen.
- Joint's Binormal Axis is the axis around which the swing2 happens, along which BinormalMotion will happen.

To change the Joint's Axes at any point in time, set the following properties appropriately

- localAxisA
- localNormalA
- localAxisB
- localNormalB

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
--If ObjectA's orientation is vector(0,0,0)
objJoint.localAxisA = vector(0,1,0)
objJoint.localNormalA = vector(1,0,0)
-- If the ObjectB's orientation is vector(90,0,0)
objJoint.localAxisB = vector(0,0,-1)
objJoint.localNormalB = vector(1,0,0)

//JavaScript Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
//If ObjectA's orientation is vector(0,0,0)
objJoint.localAxisA = vector(0,1,0);
objJoint.localNormalA = vector(1,0,0);
// If the ObjectB's orientation is vector(90,0,0)
objJoint.localAxisB = vector(0,0,-1);
objJoint.localNormalB = vector(1,0,0);
```

See also

localAxisB, localNormalA, localNormalB

localAxisB

Syntax

```
D6joint.localAxisB
```

Access: Get/Set

Description

This is the Primary axis of the joint in terms of the local coordinate space of objectB of the the D6Joint. localAxisB and localNormalB should at right angles to each other. This is axis around which twist rotation and along which AxisMotion are defined.

The values for the properties localAxisA, localAxisB, localNormalA, localNormalB, localAnchorA, localAnchorB are available only after you set the values explicitly for these properties. If the value is not set, a void vector is returned.

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
--If ObjectA's orientation is vector(0,0,0)
objJoint.localAxisA = vector(0,1,0)
objJoint.localNormalA = vector(1,0,0)
-- If the ObjectB's orientation is vector(90,0,0)
objJoint.localAxisB = vector(0,0,-1)
objJoint.localNormalB = vector(1,0,0)

//JavaScript Syntax
```

```
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
//If ObjectA's orientation is vector(0,0,0)
objJoint.localAxisA = vector(0,1,0);
objJoint.localNormalA = vector(1,0,0);
// If the ObjectB's orientation is vector(90,0,0)
objJoint.localAxisB = vector(0,0,-1);
objJoint.localNormalB = vector(1,0,0);
```

See also

localAxisA, localNormalA, localNormalB

localNormalA

Syntax

```
D6joint.localNormalA
```

Access: Get/Set

Description

This is the normal axis of the joint in terms of the local coordinate space of objectA of the the D6Joint. localAxisA and localNormalA should at right angles to each other. This is axis around which swing1 rotation and along which NormalMotion are defined.

When a rigid body is static, the local axes set for the body are considered global because they cannot be oriented along the joint axes.

The values for the properties localAxisA, localAxisB, localNormalA, localNormalB, localAnchorA, localAnchorB are available only after you set the values explicitly for these properties. If the value is not set, a void vector is returned.

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
--If ObjectA's orientation is vector(0,0,0)
objJoint.localAxisA = vector(0,1,0)
objJoint.localNormalA = vector(1,0,0)
-- If the ObjectB's orientation is vector(90,0,0)
objJoint.localAxisB = vector(0,0,-1)
objJoint.localNormalB = vector(1,0,0)
```

```
//JavaScript Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
//If ObjectA's orientation is vector(0,0,0)
objJoint.localAxisA = vector(0,1,0);
objJoint.localNormalA = vector(1,0,0);
// If the ObjectB's orientation is vector(90,0,0)
objJoint.localAxisB = vector(0,0,-1);
objJoint.localNormalB = vector(1,0,0);
```

See also

localAxisB, localAxisA, localNormalB

localNormalB

Syntax

```
D6joint.localNormalB
```

Access: Get/Set

Description

This is the normal axis of the joint in terms of the local coordinate space of objectB of the D6Joint. localAxisB and localNormalB should be at right angles to each other. This is the axis around which swing1 rotation and along which NormalMotion are defined.

The values for the properties localAxisA, localAxisB, localNormalA, localNormalB, localAnchorA, localAnchorB are available only after you set the values explicitly for these properties. If the value is not set, a void vector is returned.

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
--If ObjectA's orientation is vector(0,0,0)
objJoint.localAxisA = vector(0,1,0)
objJoint.localNormalA = vector(1,0,0)
-- If the ObjectB's orientation is vector(90,0,0)
objJoint.localAxisB = vector(0,0,-1)
objJoint.localNormalB = vector(1,0,0)
```

```
//JavaScript Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
//If ObjectA's orientation is vector(0,0,0)
objJoint.localAxisA = vector(0,1,0);
objJoint.localNormalA = vector(1,0,0);
// If the ObjectB's orientation is vector(90,0,0)
objJoint.localAxisB = vector(0,0,-1);
objJoint.localNormalB = vector(1,0,0);
```

See also

localAxisA, localAxisB, localNormalA

name

Usage

```
d6joint.name
```

Access: Get

Type: string

Description

Returns the name of the 6dof joint.

Example

```
--Lingo Syntax
objConstraint = member("PhysicsWorld").getConstraint("d6joint")
put objConstraint.name
```

```
//Javascript Syntax
var objConstraint = member("PhysicsWorld").getConstraint("d6joint");
put(objConstraint.name);
```

normalDrive

Syntax

```
d6joint.normalDrive = [type, spring, damping, forceLimit]
```

Access: Get/Set

Description

List value that specifies the linear drive for the joint, which will drive the joint to the specified position along the joint's normal axis .

The list contains the following attributes:

type #position or #velocity

Note: *If you choose position, the value for forcelimit is ignored and only the value for spring is considered. Similarly, if you choose velocity, the value for spring is ignored and only the value for forcelimit is considered.*

stiffness the spring to be applied while driving (> 0)

damping the damping of the spring (> 0)

forceLimit The force or torque with which to drive to the specified position or velocity (> 0)

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.normalDrive = [#position,100,0.1,100]
```

```
//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.normalDrive = [#position,100,0.1,100];
```

See also

swingDrive, twistDrive, driveAngularVelocity, driveLinearVelocity, driveOrientation, drivePosition, axisDrive, biNormalDrive

normalMotion

Syntax

```
D6joint.normalMotion
```

Access: Get/Set

Description

Symbol that sets the linear degree of freedom along the joint's Normal Axis.

This property takes the following values:

#Locked There can be no movement along this DOF

#Limited There can be a limited movement along this DOF

#Free There is no restriction for movement along this DOF

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
put objJoint.normalMotion
```

```
//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
Put(objJoint.normalMotion);
```

See also

swing1Motion, swing2Motion, twistMotion, axisMotion, biNormalMotion

objectA

Usage

```
d6joint.objectA
```

Access: Get

Type: RigidBody

Description

Returns the name of rigid body "A" attached to the joint.

Example

```
--Lingo Syntax
objJoint = member("PhysicsWorld").getConstraint("Joint01")
put objJoint.objectA
```

```
//JavaScript Syntax.
objJoint = member("PhysicsWorld").getConstraint("Joint01");
put(objJoint.objectA);
```

See also

objectB

objectB

Usage

```
d6joint.objectB
```

Access: Get

Type: RigidBody

Description

Returns the name of rigid body "B" attached to the spring.

Example

```
--Lingo Syntax
objJoint = member("PhysicsWorld").getConstraint("Joint01")
put objJoint.objectB
```

```
//JavaScript Syntax.
objJoint = member("PhysicsWorld").getConstraint("Joint01");
put(objJoint.objectB);
```

See also

objectA

swing1Limit

Syntax

```
D6joint.swing1limit = [limitvalue, stiffness, damping, restitution]
```

Access: Get/Set

Description

List that specifies the joint behavior when the angular motion around the joint's normal axis is limited. The rigid body will oscillate when it hits the limit value.

Limit value can take values between 3.14 to -3.14 (pi).

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.swing1limit =  [3.14*0.5, 100, 0.01, 0]

//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.swing1limit = [3.14*0.5, 100, 0.01, 0];
```

Note: *For the limits to take effect, set the limit values first and then set the motion parameters[#free,#limited etc].*

See also

swing2Limit, twistLimit, linearLimit

swing1Motion

Syntax

```
D6joint.swing1Motion
```

Access: Get/Set

Description

Symbol that sets the angular degree of freedom around the joint's Normal Axis.

This property takes the following values

#Locked There can be no movement along this DOF

#Limited There can be limited movement along this DOF

#Free There is no restriction for movement along this DOF

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
put objJoint.swing1Motion

//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
Put(objJoint.swing1Motion);
```

See also

normalMotion, swing2Motion, twistMotion, axisMotion, biNormalMotion

swing2Limit

Syntax

```
D6joint.swing2limit = [limitvalue, stiffness, damping, restitution]
```

Access: Get/Set

Description

List that specifies the joint behavior when the angular motion around the joint's binormal axis is limited. The rigid body oscillates when it hits the limit value.

Limit value can take values between 3.14 to -3.14 (pi).

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.swing2limit =  [3.14*0.5, 100, 0.01, 0]
```

```
//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.swing2limit = [3.14*0.5, 100, 0.01, 0];
```

Note: *For the limits to take effect, set the limit values first before you set the motion parameters[#free,#limited etc].*

See also

swing1Limit, twistLimit, linearLimit

swing2Motion

Syntax

```
D6joint.swing2Motion
```

Access: Get/Set

Description

Symbol that sets the angular degree of freedom around the joint's binormal Axis.

This property takes the following values

#Locked There can be no movement along this DOF

#Limited There can be limited movement along this DOF

#Free There is no restriction for movement along this DOF

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
put objJoint.swing2Motion
```

```
//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
Put(objJoint.swing2Motion);
```

See also

normalMotion, swing1Motion, twistMotion, axisMotion, biNormalMotion

swingDrive

Syntax

```
d6joint.swingDrive = [type, spring, damping, forceLimit]
```

Access: Get/Set

Description

List value that specifies the angular drive for the joint, which will drive the joint to the specified orientation around the joint's normal and biNormal axes.

The list contains the following attributes:

type #position or #velocity

stiffness the spring to be applied while driving (> 0)

damping the damping of the spring (> 0)

forceLimit The force or torque with which to drive to the specified position or velocity (> 0)

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.swingDrive = [#position,100,0.1,100]
```

```
//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.swingDrive = [#position,100,0.1,100];
```

See also

normalDrive, twistDrive, driveAngularVelocity, driveLinearVelocity, driveOrientation, drivePosition, axisDrive, biNormalDrive

twistDrive

Syntax

```
d6joint.twistDrive = [type, spring, damping, forceLimit]
```

Access: Get/Set

Description

List value that specifies the angular drive for the joint, which will drive the joint to the specified orientation around the joint's axis.

The list contains the following attributes:

type #position or #velocity

stiffness the spring to be applied while driving (> 0)

damping the damping of the spring (> 0)

forceLimit The force or torque with which to drive to the specified position or velocity (> 0)

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.twistDrive = [#position,100,0.1,100]
```

```
//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.twistDrive = [#position,100,0.1,100];
```

See also

normalDrive, swingDrive, driveAngularVelocity, driveLinearVelocity, driveOrientation, drivePosition, axisDrive, biNormalDrive

twistLimit

Syntax

```
D6joint.twistLimit = [limitvalue, stiffness, damping, restitution]
```

Access: Get/Set

Description

List that specifies the joint behavior when the angular motion along the joint's axis is limited. The rigid body will oscillate when it hits the limit value.

Limit value can take values between 3.14 to -3.14 (pi).

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
objJoint.twistLimit =  [3.14*0.5, 100, 0.01, 0]
```

```
//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
objJoint.twistLimit = [3.14*0.5, 100, 0.01, 0];
```

Note: *For the limits to take effect, set the limit values before you set the motion parameters[#free,#limited etc].*

See also

swing2Limit, swing1Limit, linearLimit

twistMotion

Syntax

```
D6joint.twistMotion
```

Access: Get/Set

Description

Symbol that sets the angular degree of freedom around the joint's Primary Axis.

This property takes the following values

#Locked There can be no movement along this DOF

#Limited There can be limited movement along this DOF

#Free There is no restriction for movement along this DOF

Example

```
--Lingo Syntax
objJoint = Member("PhysicsWorld").getConstraint("D6Joint")
put objJoint.twistMotion
```

```
//JavaScript Syntax
var objJoint = Member("PhysicsWorld").getConstraint ("D6Joint");
Put(objJoint.twistMotion);
```

See also

normalMotion, swing2Motion, swing1Motion, axisMotion, biNormalMotion

RayCasting methods

- rayCastAll
- rayCastClosest

rayCastAll

Usage

```
<list <list>> world.rayCastAll(vector origin, vector direction)
```

Description

This method will return the references of all the rigid bodies or terrains that are found along the ray from the specified origin and specified direction. The method also returns the point of contact, contact normal, and the distance from the origin of the ray.

Parameters

Parameter	Description
origin	Required. Vector that specifies the origin of the raycast.
direction	Required. Vector that specifies the direction of the raycast.

This method returns a list containing a list having the following information:

- Rigid body / Terrain reference
- Contact Point
- Contact Normal
- Distance of the rigid body or terrain, from the origin of the ray.

Example

```
--Lingo Syntax
lstraycast = member("PhysicsWorld").rayCastAll (vector(10,0,0),vector(0,0,1))
repeat with i = 1 to lstraycast.count
    raycstEntry = lstraycast[i]
    put "Name:" & raycstEntry[1].name
    put "Contact Point:" & raycstEntry[2]
    put "Contact Normal:" & raycstEntry[3]
    put "Distance:" & raycstEntry[4]
end repeat
```

```
//JavaScript Syntax
var lstraycast = member("PhysicsWorld").rayCastAll (vector(10,0,0),vector(0,0,1));

for(i = 1; i <= lstraycast.count ; i++)
{
```

```
    raycstEntry = lstraycast[i];
    put("Name:" + raycstEntry[1].name);
    put"Contact Point:" & raycstEntry[2]);
    put("Contact Normal:" & raycstEntry[3]);
    put("Distance:" & raycstEntry[4]);
}
```

See also

rayCastClosest

rayCastClosest

Usage

```
<list> world.rayCastClosest(vector origin, vector direction)
```

Description

This method returns the reference of the closest rigid body or terrain that is found along the ray from the specified origin and specified direction. The method also returns the point of contact, contact normal, and the distance from the origin of the ray.

Parameters

Parameter	Description
origin	Required. Vector that specifies the origin of the raycast.
direction	Required. Vector that specifies the direction of the raycast.

This method returns a list containing the following information:

- Rigid body / Terrain reference
- Contact Point
- Contact Normal
- Distance of the rigid body or terrain, from the origin of the ray.

Example

```
--Lingo Syntax
lstraycast = member("PhysicsWorld").rayCastClosest(vector(10,0,0),vector(0,0,1))
put "Name:"  & lstraycast[1].name
put "Contact Point:"  & lstraycast[2]
put "Contact Normal:"  & lstraycast[3]
put "Distance:"  & lstraycast[4]
```

```
//JavaScript Syntax
var lstraycast = member("PhysicsWorld").rayCastClosest(vector(10,0,0),vector(0,0,1));
put("Name:"  + lstraycast[0].name);
put"Contact Point:"  & lstraycast[1]);
put("Contact Normal:"  & lstraycast[2]);
put("Distance:"  & lstraycast[3]);
```

See also

rayCastAll

Error Codes

The following error codes along with their reasons are described in the following table.

Error code	Reason for the error
-1	Unknown failure You may get a -1 error when creating a rigid body if the camera and the model are separated by a large distance. This is because the mesh count becomes zero due to optimization.
-2	World not initialized
-3	Simulate failed
-4	Invalid parameter
-5	Duplicate constraint
-6	Invalid Constraint / The constraint does not exist.
-8	Invalid Rigid body / The rigid body does not exist.
-11	Element not found. Some physics object which has been deleted is being used.
-18	Invalid 3D model
-19	3D error
-20	Insufficient memory
-21	Mesh deform is not added for the Model.
-28	When a rigid body with concaveshape proxy is made #dynamic

Index

D

N

P

S

U

V