Isomorphic SmartClient Widgets Guide

Version 5.2

Isomorphic SOFTWARE™

Making web applications work as well as desktop applications.

Isomorphic Software, Inc.
109 Stevenson Street, Level 4
San Francisco, CA 94105-3475
U.S.A.

Web:	**www.isomorphic.com**
Email:	**info@isomorphic.com**

CONTENTS

LIST OF TABLES

PREFACE

The Isomorphic SmartClient (ISC) presentation layer is the first web application framework that spans across client *and* server, enabling live transactions and rich user interactivity *without page reloads*. ISC supports an easy, declarative authoring style in either XML or JavaScript, enabling the development of high-performance, rich web applications and portals in a fraction of the time required with other technologies. The ISC framework is also open and extensible on both client and server, allowing seamless integration with existing applications and infrastructure.

This guide contains technical details on the user interface components, called *widgets*, built into the Isomorphic SmartClient framework and how to manipulate instances of these widgets to create an application interface with superior usability. It also discusses how to customize images and styles to create a unique widget appearance.

In the Preface:

Audience

The *Isomorphic SmartClient Widgets Guide* is written for web developers who want to build user interfaces for applications within the Isomorphic SmartClient framework, and customize or extend the framework for a specific business environment. It also may be useful to web designers and producers who need to make decisions about front-end usability and site navigation based on the widgets available in the Isomorphic SmartClient system.

Prerequisites

Developers using the Isomorphic SmartClient framework should have a basic knowledge of web page development, including:

- basic HTML and familiarity with XML,
- basic JavaScript and fundamental programming concepts, and
- web site organization and deployment.

Developers wishing to integrate the ISC framework with existing server systems should also be adept at programming in Java, and have some familiarity with JavaServer Pages (JSPs), servlets, and web servers.

How to use this guide

Use this guide to learn how to incorporate user interface widgets into your web applications using the Isomorphic SmartClient framework. Widgets are implemented using extensions and enhancements to the JavaScript language developed by Isomorphic Software. These enhancements include:

- *A true class-based object system* to support both classes and instances, like the Java language.
- *A consistent cross-browser drawing system* to provide a consistent set of properties and methods for positioning, sizing, clipping, scrolling, and controlling other visual properties of on-screen elements.
- *A consistent cross-browser event-handling system* to expose all of the standard browser events, and provide extensive drag-and-drop functionality.
- *Pre-defined GUI widget classes*—from simple objects like buttons and scrollbars, to complex elements like table viewers and dynamic forms—to ensure sophisticated, interactive applications look and behave the same in any standard DHTML-enabled web browser.
- *An abstracted application framework* to enable rapid development of data-centric applications.
- *Data type objects and extensions*—including hierarchical tree objects, array list searching and sorting extensions, and more—to provide better application flow and navigation.
- *Server communication objects* to provide seamless integration of the browser-based front end with back-end application servers and data sources.

- *Utility objects and methods* to perform common tasks quickly and consistently.

This manual focuses on the object system, drawing widgets, handling events, and several of the most common classes of widgets. For full reference documentation of all widgets and all their public properties and methods, see the *SmartClient Online Reference*.

The server-side of the Isomorphic SmartClient framework is implemented in Java, and includes validation, data translation, and dataset paging capabilities in addition to pre-compiled standard data source operations for rapid development. For documentation on the server framework, consult the *SmartClient Online Reference*. For additional developer documentation covering custom widget development, data type extensions, and utilities, please contact Isomorphic Software directly.

http://www.isomorphic.com/

Summary of chapter contents

This guide contains the following content:

- Chapter 1, *"Widgets Overview,"* provides an overview of the ISC user interface components, the widget class hierarchy, and the advantages to using ISC widgets in building web applications.
- Chapter 2, *"Drawing Widgets,"* details the basic properties common to all widgets in the ISC framework including position, scrolling, sizing, background attributes, visibility, layering, styles, clipping, padding, borders, and margins. These properties are controlled by the *Canvas* widget superclass.
- Chapter 3, *"Handling Events,"* explains the event handling capabilities of the ISC system, and how to capture and respond to page events, mouse events, drag-and-drop events, and keyboard events.
- Chapter 4, *"Images and Skins,"* discusses how to place images within a ISC application, the "special directories" that can be used to specify the location of images, and how the appearance of widgets themselves can be customized through the use of application "skins".
- Chapter 5, *"Labels, Buttons, and Bars,"* details the properties and methods available to use with ISC text labels, interactive style-based buttons, scrollbars, and progressbars.
- Chapter 6, *"Forms,"* demonstrates how to build forms by generating HTML, laying out form elements, altering the styles and behavior of forms, getting and setting form values, validating form input, and handling form events.
- Chapter 7, *"ListGrids and DetailViewers,"* describes the two widget classes used to display tabular data—*ListGrid* and *DetailViewer*— and how the two classes differ. It also covers configuring layout and appearance, adding or removing records from a list, sorting a list, selecting records from a list, dragging and dropping a record from one list to another, and handling list events.
- Chapter 8, *"TreeGrids,"* describes the *TreeGrid* widget class used to display hierarchical data as folder and leaf nodes, and how to manipulate tree data by adding,

moving, and removing nodes, expanding and collapsing nodes, dragging and dropping nodes, and handling tree events.

- Chapter 9, *"Menus, Toolbars, and Menubars,"* describes the ISC classes used for navigation of an application. The *Menu* class implements interactive menus with icons, submenus, and shortcut keys. The *Toolbar* class is used to create a row or column of buttons, each of which carries out a specific action when clicked.

What this guide does not cover

This guide does not describe the general software design process or the decisions involved in executing that process. Organizing and determining application flow is covered only in reference to the commonly-used design patterns included as application components within the Isomorphic SmartClient framework.

This guide does not discuss the Java programming language or JavaScript language, nor does it address any details of HTML or XML usage. While ISC-based applications may be database-driven, this guide does not discuss database administration or development, transaction symantics, or database features.

Resources

This section maps out a variety of documentation resources available to you.

SmartClient Online Reference

The SmartClient Online Reference contains documentation for all public ISC client and server APIs, presented in a searchable, interactive HTML format. The Reference is conveniently accessible from the Developer Console (see *"JavaScript debugging" on page 21*), but is also available online at:

http://www.isomorphic.com/devcenter/docs/52/reference.html

Sample applications

Isomorphic provides complete code samples for client and server applications in the samples directory of the Isomorphic SmartClient installation, or you can access client-side samples on the Web at:

http://www.isomorphic.com/devcenter/examples/index.jsp

You must register to obtain a valid user name and password before accessing this resource at:

http://www.isomorphic.com/developers/index.jsp

Updates to documentation

Check for updates to the documentation on the Isomorphic SmartClient Developer Center at:

http://www.isomorphic.com/devcenter/documentation/index.jsp

Related readings

If you are unfamiliar with the JavaScript language, or need additional information on the Java language, Java Servlets, or JavaServer Pages, see the following Internet publications:

- JavaScript standard (also known as ECMAScript)

  http://www.ecma-international.org/publications/standards/Ecma-262.htm
- Java Programming Language

  http://java.sun.com/
- JavaServer Pages (JSPs)

  http://java.sun.com/products/jsp/index.html
- Java Servlets

  http://java.sun.com/products/servlet/index.html

Icons and conventions used in this guide

This guide uses icons and font conventions to help you recognize different types of information.

Identifier icons

ISC WIDGETS API — The block icon denotes *Isomorphic SmartClient-specific enhancements* to the JavaScript language as *Isomorphic SmartClient Widget APIs*.

ADVANCED TOPIC — The circle icon calls out the development concepts that are *advanced topics*.

Font conventions

This guide uses the typographic conventions shown in Table P.1 to indicate commands, definitions, and keywords.

Table P.1: Font conventions

Information	Type Font	Context
Commands and keywords	**Georgia Bold**	Denotes the actions a user performs within an Isomorphic SmartClient application or keywords.
Definitions and terms	*Georgia Italic*	Isomorphic SmartClient product terminology.
Java and JavaScript code	`Courier`	Java and JavaScript keywords, code samples, paths, files, and directory names.
Coding concept emphasis	**`Courier Bold`**	Emphasizes pieces of code within a sample.
Code variables	*`Courier Italics`*	Indicates a variable that may take on different values. Also used for Isomorphic SmartClient's settable properties.
Properties and keywords	*`Courier Italics`*	Isomorphic SmartClient properties and keywords.

CHAPTER 1

Widgets Overview

Isomorphic SmartClient widgets are visual interface elements, implemented as JavaScript objects, that you can use to build cross-browser graphical user interfaces. Typical uses of widgets include:

- viewers that organize and present data visually,
- controls that accept data input from the user, via keyboard or mouse,
- navigation and command tools, and
- graphical design and decorative elements.

This chapter presents the basic widget classes and how widgets are manipulated within the ISC system.

In this chapter:

Why use widgets?

To design a new user interface, you may pull together an assortment of JavaScript widgets from different sources—a drop-down menu downloaded from a JavaScript website, a list viewer adapted from a colleague's, an animated button of your own making, and so on. These interface components were probably created independently of each other, and were likely designed for specific applications, maybe even on specific browsers. Therefore, even if your widget toolbox contains all of the elements you need, each element probably has its own unique interface and scripts for performing functions that it has in common with the others.

By contrast, all of the Isomorphic SmartClient (ISC) widgets share common drawing and event models, and function consistently on all supported browsers. This sharing and consistency make ISC widgets easier to learn, applications using them faster to build, and scripts written with them smaller, faster, and easier to maintain. Isomorphic SmartClient Widgets are supported in the client browser operating system combinations listed in Table .

The ISC system hides much of the complexity of DHTML programming, and this manual further omits many features of the system that are used to build the widgets. Still, if you're familiar with DHTML it may help you to know the technical foundation of widgets:

- ISC widgets are implemented as JavaScript objects.
- ISC widgets are drawn by dynamically creating HTML elements via the Document Object Model (DOM) interface.

An ISC widget is not the same as the HTML the ISC widget has drawn. All manipulation of an ISC widget's appearance must be done through the JavaScript object that represents the widget. Directly manipulating the HTML created by an ISC widget will have unpredictable behavior.

Client web browsers and operating systems supported by ISC

Web Browser Software	Operating System
Internet Explorer 5.5+	Windows
Netscape 7.1	Windows, MacOS
Mozilla 1.4+	Windows, MacOS, Linux
Firefox 1.0+	Windows, MacOS, Linux
Safari 1.2+	MacOS

Widget classes

Widgets are built on top of the Isomorphic SmartClient class system. This system adds to JavaScript support for true classes (abstract objects that define the properties of the instances that you actually create and manipulate). The distinction between class-based and prototype-based systems is a subtle one, best treated separately.

For purposes of using ISC widgets, what you need to know is that:

- each widget type is defined by a class,

- a widget class may be defined as a subclass of another widget class (the superclass), and then inherits all the properties and methods of the superclass,
- to use a particular widget you must create an instance of that widget's class that has its own set of properties and methods, and
- a class may include static properties and methods (typically used for constants and utility functions, respectively) that are not copied to each instance of the class.

If you aren't already familiar with these concepts, read on. They will be described in terms of practical usage below.

Note An important convention in the ISC system is that class names are always capitalized (e.g. 'the *Button* class'), while instance names always begin in lowercase (e.g. 'a *button* instance'). This convention applies both to code, and to the discussion of objects in this manual.

The *Canvas* widget is the superclass of all other widgets. As such, it provides properties and methods common to all widgets, like the left and top properties that specify a widget's position on the page. The *ListGrid*, *StretchImg*, and *Toolbar* classes are also superclasses, each implementing functionality shared by one or more subclasses.

Creating widget instances

Widget classes are abstract objects. A widget class will not, for example, be drawn on the screen or manipulated by an end user. Instead, you must create *instances* of a widget class in JavaScript using the `Class.create` constructor method. When you create a new `canvas` instance, it will automatically be drawn unless you set its `autoDraw` property to `false`. By default, `autoDraw` is set to `true`. The ramifications of this are discussed in detail in Chapter 2, *"Drawing Widgets."*

For example, the following would create a new `canvas` (note the little "c") instance from the *Canvas* class (note the big "C", both here and in the script below), storing a reference to the instance in a JavaScript global variable given by the `ID` property `myWidget`. The widget's `ID` property is optional if the widget will not need to be manipulated programmatically. A unique ID will automatically be created upon instantiation for any canvas where one is not provided. If you will be programmatically manipulating a `canvas` instance, however, you should provide a unique ID. Be aware that unpredicable results will occur if two canvases are assigned the same ID.

```
<SCRIPT>
Canvas.create({ID:"myWidget"});
</SCRIPT>
```

This isn't a very exciting example, since it doesn't have any contents to display! Jumping ahead of ourselves just a bit, here's an example that you can actually see. Type in the following script, and save it as an HTML file in the SDK, in the directory which contains the `isomorphic` directory. Then launch the file within your web browser. The `SRC` attribute should reflect the relative path to the ISC libraries from the locaton of the saved file in your environment.

```
<HTML>
```

```
<HEAD>
   <SCRIPT SRC=isomorphic/system/Isomorphic_SmartClient.js></SCRIPT>
</HEAD>

<BODY>
   <SCRIPT>
   Canvas.create({
      ID:"myFirstWidget",
      contents:"Hello world!",
      backgroundColor:"blue"
   });
   </SCRIPT>
</BODY>

</HTML>
```

Note

For brevity, examples from this point onward will omit the standard HTML structure tags (`<HTML>`, `<HEAD>`, etc.), initialization `<SCRIPT>` include tag, and surrounding `<SCRIPT>` tags.

In addition to creating a new widget instance, the above code initializes two of the widget's properties. The `create` constructor method takes as a parameter an object, expressed as a list of *`property`*`:`*`value`* pairs. Note these pairs are surrounded by curly braces ('`{`' and '`}`') to define an object, not the square brackets ('`[`' and '`]`') that define an array. The properties of this passed object are added to, or override the values of, the default properties and values of the new instance.

Note

Curly braces, '`{`' and '`}`', in JavaScript indicate an object initializer, also referred to as literal notation for creating an object. For example, the following defines an object, where each property is an identifier and each value is an expression whose value is assigned to property.

```
{property1:value1, property2:value2, ..., propertyN:valueN}
```

An object initializer can be assigned to a variable…
`objectName = {property1:value1, ...}`or may be used directly (e.g. as a parameter to a function call). Properties in an object initializer may themselves be associated with object initializers as follows.

```
{property1:value1, property2:{property2a:value2a, ...}, ...}
```

JavaScript object initializers are used frequently in the ISC system, so you should familiarize yourself with this syntax.

For a more interactive example, use the following script to create a button widget:

```
Button.create({ID:"myButton"});
```

This script creates a default button widget—a 20-by-100-pixel gray area at the top-left corner of the page, with the rollover and click-highlighting behavior of a standard GUI button. But the default button has no title, and no action in response to a click. The following script repositions and resizes the button, adds a title, and adds a click action.

```
Button.create({
    ID:"myButton",
    left:100,
    top:100,
    height:50,
    title:"Say hello",
    click:"alert('Hello')"
});
```

Warning

The commas delimiting *`property`*`:`*`value`* pairs are a very common area for syntax errors, especially if you've commented out some of the properties in your script. Ensure that all *`property`*`:`*`value`* pairs are separated by commas, but that the last pair is not followed by a comma. For example, if you were to comment out the click property above, you would need to remove the comma following the `title` property to avoid an execution error, `"Expected identifier or string."` in IE.

Omitting a comma in a list of property value pairs results in an error with both browsers, `"Expected '}'."`, because without a comma, JavaScript expects a close to the object definition. Also, ensure that each property-value pair is separated by a colon.

The above example actually initializes properties inherited from two different classes. The `left`, `top`, `height`, and `click` properties are implemented in the *Canvas* class, while the `title` property is implemented in the *Button* class. Because *Button* inherits all of the properties of *Canvas*, we can initialize these properties simultaneously when creating a new button instance.

JavaScript vs. XML

Widgets can be created using either JavaScript or XML in a very similar declarative format. If you are familiar with JavaScript it is generally recommended that you use the JavaScript format, since advanced usage of SmartClient such as creating custom behaviors or creating new components currently requires JavaScript.

To use XML to declare your widgets, you must use a `.jsp` file with a `taglib` directive at the top to pick up the Isomorphic SmartClient XML tags. XML declarations can then be parsed by a servlet engine.

```
<%@ taglib uri="isomorphic" prefix="isomorphic" %>
```

JavaScript declarations can be done within a `.html` file and do not need to be parsed by a server.

In JavaScript, *`property`*:*`value`* pairs are separated by colons, and followed by a comma for all but the last pair. Instantiation is performed by surrounding the declaration with the `create()` method:

```
WidgetClass.create({property1:value1, property2:value2, ...})
```

In XML, *`property`*="*`value`*" pairs are separated by an equals sign, and followed by a space. Widget declarations in XML must be surrounded by`<isomorphic:XML>` tags, which cause the XML to be translated to JavaScript code that creates SmartClient widgets, as follows:

```
<isomorphic:XML>
```

```
    <WidgetClass property1="value1" property2="value2" .../>
</isomorphic:XML>
```

Array-valued properties are created in JavaScript using the square brackets '`[`' and '`]`'. For example, the *StretchImg* class uses an array specifying `items`. In JavaScript this appears as:

```
items:[
    {name:"start", width:"capSize", height:"capSize"},
    {name:"stretch", width:"*", height:"*"},
    {name:"end", width:"capSize", height:"capSize"}
],
```

In XML, inner tags are used to declare an array. The same items array declared above in JavaScript would appear as follows in XML:

```
<items>
    <item name="start" width="capSize" height="capSize"/>
    <item name="stretch" width="*" height="*"/>
    <item name="end" width="capSize" height="capSize"/>
</items>
```

To illustrate the main differences, Figure 1.1 compares a JavaScript and XML declaration side-by-side using the simple `myFirstWidget` example.

JavaScript Declaration
`myFirstWidget.html`

```
<HTML>

<HEAD>
  <SCRIPT SRC=../isomorphic/system/
  Isomorphic_SmartClient.js></SCRIPT>
</HEAD>

<BODY>
  <SCRIPT>
      Canvas.create({
          ID:"myFirstWidget",
          contents:"Hello world!",
          backgroundColor:"blue"
      });
  </SCRIPT>
</BODY>


</HTML>
```

XML Declaration
`myFirstWidget.jsp`

```
<%@ taglib uri="isomorphic"
         prefix="isomorphic" %>

<HEAD>
  <SCRIPT SRC=../isomorphic/system/
  Isomorphic_SmartClient.js></SCRIPT>
</HEAD>

<BODY>
  <SCRIPT>
      <isomorphic:XML>
          <Canvas
              ID="myFirstWidget"
              contents="Hello world!"
              backgroundColor="blue"
          />
      </isomorphic:XML>
  </SCRIPT>
</BODY>
```

Figure 1.1: JavaScript and XML declaration differences

Including a separate XML file

The XML declaration could also be done in a separate file and included in the `.jsp` loader by using `<isomorphicXML>` tags (without the colon) in the included file, and using the `filename` property to include the file within the `.jsp` file using the `<isomorphic:XML>` tag (with the colon).

For example, suppose the following `myFirstWidget.xml.txt` file were used to define the `myFirstWidget canvas` instance.

```
<!-- myFirstWidget.xml.txt file -->
<isomorphicXML>
   <Canvas
      ID="myFirstWidget"
      contents="Hello blue world!"
      backgroundColor="blue"
   />
   <Canvas
      ID="mySecondWidget"
      contents="Hello red world!"
      backgroundColor="red"
   />
</isomorphicXML>
```

This `myFirstWidget.xml.txt` file would then be included in the `myFirstWidget.jsp` file as follows using the `<isomorphic:XML>` tag (with the colon).

```
<!-- myFirstWidget.jsp loader file -->
<%@ taglib uri="isomorphic" prefix="isomorphic" %>

<HEAD>
   <SCRIPT SRC=../isomorphic/system/Isomorphic_SmartClient.js></SCRIPT>
</HEAD>

<BODY>
   <SCRIPT>
      <isomorphic:XML filename="myFirstWidget.xml.txt" />
   </SCRIPT>
</BODY>
```

Note that JavaScript code can be interleaved with XML in both .jsps and separate .xml files.

JavaScript and XML templates including all widget properties available for initialization, along with their default values, are provided in Appendix A, "*Widget Initialization Templates*." Each widget property and its usage will be illustrated in the appropriate functional section of this guide.

Reference

For clarity, only the JavaScript declarations are used in the source code examples throughout the remainder of this guide. Refer to Appendix A, "*Widget Initialization Templates*" for the XML equivalent structure for each widget class type.

Manipulating widget instances

Widget instances can be manipulated using JavaScript in the same way that you would manipulate any other JavaScript objects-through *properties* and *methods*. These are accessed by standard dot-notation:

`object.property`

or

`object.method``()`

A method that neither takes parameters nor returns a value is called by the simplest possible syntax:

```
widgetInstance.method();
```

Some widget methods do take parameters and/or return a value; their calling syntax is the same as that of any similar JavaScript function. For example:

```
variable = widgetInstance.method(parameter1, parameter2, ...);
```

Some methods have optional parameters, indicated in this manual by square brackets. For example, both parameters of the `moveTo` method are optional, so it is documented as:

```
moveTo([x], [y])
```

To omit an optional parameter that is not the last parameter in a method call, you must pass the value `null` for that parameter. For example, the following call moves `myWidget` to the top of the screen without changing its horizontal position:

```
myWidget.moveTo(null, 0)
```

Manipulating widget properties is a trickier propostion. All of a widget's properties that are available for initialization in the `create` constructor method are technically accessible via standard JavaScript dot notation, but only some of them *should* be accessed directly. Many widget properties should never be accessed after initialization, while others have corresponding 'setter' and/or 'getter' methods that can be called to access them indirectly. For example, the `left` property initialized in the previous example is never accessed directly; the `getLeft` and `setLeft` methods are called instead. The following script demonstrates these methods; it moves the button 10 pixels to the right by setting its left coordinate:

```
myButton.setLeft(myButton.getLeft() + 10);
```

The chapters that follow show the correct mechanism (property or method) for each operation that you can perform with a widget.

Warning

Assume that a property cannot be directly accessed or set after initialization unless the ISC documentation explicitly states otherwise.

Referring to widget instances

There are three ways to refer to widget instances, all of which are demonstrated in bold in the following script:

```
Canvas.create({
    ID:"myWidget",
    backgroundColor:"khaki",
    contents:"catch me!",
    click:"this.setLeft(0)"
});
Page.setEvent("idle", myWidget.getID()+".moveBy(5,0)");
```

This example draws a simple square widget that moves from left to right across the top of the page; clicking on the widget returns it to the left side of the page. The script refers to this widget instance via:

1. **A variable that holds a reference to the instance.**

   The `Canvas.create` constructor method returns a reference to the new widget instance, which is stored in the variable `myWidget` in this example. This variable is then used to call the widget's `getID` method.

   Most of the examples in this manual, and most of the scripts that you write, will reference widgets through variables in this manner. A variable storing a widget reference could be local or global, and its value could be passed as a function parameter or assigned to another variable.

2. **The '`this`' keyword.**

   The '`this`' keyword is a JavaScript keyword that refers to the object containing the current method. If you build or extend widget classes, you'll make frequent use of '`this`' when writing new methods. If you're using existing widgets, you'll probably use '`this`' only in event handlers, like the '`click`' event handler above. See *"Handling widget events" on page 48* for more information on widget event handlers.

3. **The instance's global ID.**

   When a widget instance is created, it is assigned a unique global identifier that can be used to access the instance by name. The `getID` method returns this ID for a particular instance. Global IDs are essential when you need to embed a widget reference in a string, usually a string that will be evaluated in the future and/or in another object, where you may not have access to a variable or parameter holding the widget's reference.

   Event handlers are usually specified as strings of script, and are the most frequent users of global ID references. The script above embeds the new widget instance's ID in a handler for the global '`idle`' event. See *"Handling Page events" on page 44* for more information on global event handlers.

Note As stated briefly in "*Creating widget instances*" above, you do not need to explicitly assign an ID for a widget unless it will be manipulated programmatically. For example, if you have a button that moves another widget, the button doesn't need an ID, but the widget that will be moved does.

Widget containment and attachment—children and peers

Widgets can be nested in a containment hierarchy, in which each widget may contain other widget, which may contain other widgets, and so on. The child widgets nested inside a parent widget inherit a number of properties from the parent, including their drawing context (i.e., positioning origin), stacking-order container, and various style attributes. Child widgets may also be clipped by the rectangle of their parent layer, depending on the parent's attribute settings.

ISC widgets also have the following behaviors with respect to their parents and/or children:

- Child widgets draw and redraw automatically when their parent is drawn or redrawn.

- Child widgets are cleared from memory automatically when their parent is cleared from memory.
- Events targeting child widgets "bubble" up through the widget containment hierarchy.
- A consistent set of methods and properties manipulates the widget containment hierarchy in all supported browsers.

Table 1.1 lists containment-related methods and properties implemented in the *Canvas* class.

Table 1.1: Canvas class containment methods and properties

Method / Property	**Action/Description**
`addChild(`*`childObject`*`, [`*`childName`*`])`	Adds *`childObject`* as a child of this widget, set up a named object reference (i.e., `this.`*`childName`*) to the new widget if *`childName`* is provided, and draw the child if this widget has been drawn already.
`parentElement`	Object reference to this widget's immediate parent, if any.
`topElement`	Object reference to the top-most (i.e., not a child of any other widget) widget, if any, in this widget's containment hierarchy.
`children`	Array of object references to all widgets that are immediate children of this widget.
`getParentElements()`	Returns an array of object references to all ancestors of this widget in the containment hierarchy, starting with `this.parentElement` and ending with `this.topElement`.
`contains(`*`childObject`*`)`	Returns `true` if *`childObject`* is a descendant of this widget (i.e., exists below this widget in the containment hierarchy); and `false` otherwise.
`autoDraw`	**Default:** `true` If set to `true`, this canvas will draw itself immediately after it is created. You should set this property to `false` for any canvases that you intend to add as children of other canvases, or they will draw twice!

For example, after executing the following script:

```
Canvas.create({ID:"A"});
Canvas.create({ID:"B", autoDraw:false});
Canvas.create({ID:"C", autoDraw:false});

A.addChild(B,"Bob");
B.addChild(C,"Chris");
```

the following statements would all evaluate to `true`:

```
A.parentElement == null
B.parentElement == A
C.parentElement == B

A.topElement == null
B.topElement == A
```

```
C.topElement == A

A.children == [B]
B.children == [C]
C.children == null

A.getParentElements() == []
B.getParentElements() == [A]
C.getParentElements() == [B,A]

A.contains(B)
A.contains(C)
B.contains(C)

A.Bob == B
B.Chris == C
```

Warning

As shown in the initialization script above, each canvas that is created to be added as child to another canvas must have its `autoDraw` property set to `false`, or the child canvas will draw twice with unpredictable results! The `autoDraw` property is set to `true` by default.

While the `addChild` method is available to all of the widget classes, it is typically used by *Canvas* widgets or other specialized widgets like the *ListGrid*, which manage their own children. Whenever a widget is redrawn, it draws its own contents only if it has no children. So, it doesn't make sense to add children to a widget that has dynamic content. A *Canvas* widget can serve as a simple shell to contain and position child widgets.

The ISC system also provides a mechanism for "attaching" widgets to other widgets without actually nesting them. Any widget can have a number of peer widgets associated with it. When this master widget is:

- moved,
- resized,
- enabled or disabled,
- shown or hidden,
- focused or blurred, or
- cleared from memory,

any peers attached to the widget will be similarly affected.

Since peer widgets are not actually nested inside their master widget, they are never clipped by the master widget (unlike children, which may be clipped by their parents). Common uses of peers, therefore, are as borders or other graphic design elements accompanying a widget. Peers are also used as interactive parts of complex widgets (e.g. the "thumb" of the scrollbar widget).

The attachment related methods and properties described in Table 1.3 are common to all widgets. These methods and properties are implemented in the *Canvas* class.

Table 1.2: Canvas class attachment methods and properties

Method / Property	Action/Description
`addPeer(peerObject, [peerName])`	Adds *`peerObject`* as a peer of this widget (also making it a child of this widget's parent, if any), set up a named object reference (i.e., `this.`*`peerName`*) to the new widget if *`peerName`* is provided, and draw the peer if this widget has been drawn already.
`masterElement`	Object reference to this widget's 'master' (the widget to which it was added as a peer), if any.
`peers`	Array of object references to all widgets that are peers of this widget.

While a peer widget is not contained by its 'master' widget as a child, it is contained by the master widget's parent. If you add a peer to a master widget that has a parent, the peer will become a child of that parent as well. Similarly, if you add a master widget to a new parent, it will carry all of its peers over to that parent as children. These effects are demonstrated visually in the *"Example: Widget containment and attachment"* which will be introduced in the following section.

Nesting children within a parent widget declaration

The previous example, repeated below for comparison purposes, has an equivalent form that is accomplished by nesting children within the parent widget declaration. Note that the widget class to use for instantiation is given by the widget class upon which the `create` method is called.

```
// Children are instantiated separately from their parents.
// The autoDraw property must be set to false.

Canvas.create({ID:"A"});
Canvas.create({ID:"B", autoDraw:false});
Canvas.create({ID:"C", autoDraw:false});

A.addChild(B,"Bob");
B.addChild(C,"Chris");
```

Nesting the children in the parent declaration is equivalent to the above declaration except, note that in this case, the `autoDraw` property does not need to be set to `false` because the children are completely controlled by the parent and will only be drawn when the parent is drawn (at instantiation time) or redrawn.

Note

Children and peers are declared in essentially the same way. Peers can be nested within their master widget's `peers` array using the `constructor` property just as children can be nested in their parent widget's `children` array.

```
// Nesting children within a parent declaration so that children will be
// instantiated with their parent class.
// The constructor property must be set for all children to declare their
// widget class types.

Canvas.create({
    ID:"A",
    children:[
        Canvas.create({
            ID:"B",
            children:[
                Canvas.create({
                    ID:"C"
                })
            ]
        })
    ]
});
```

Example: Widget containment and attachment

The example file `widget_attachment.html` (shown in Figure 1.2) provides a visual demonstration of widget containment and attachment.

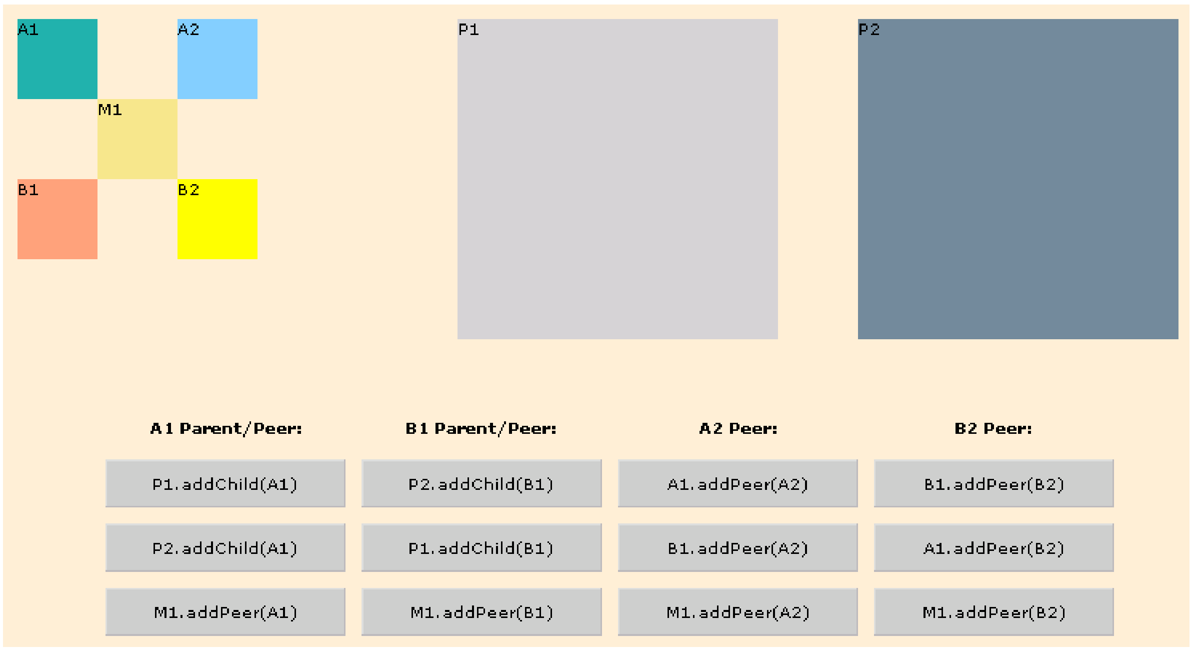


Figure 1.2: Example of widget containment and attachment

This page presents seven page-level, mouse-draggable *Canvas* widgets, and a set of buttons below them that manipulate their containment and attachment relationships via the `addChild` and `addPeer` methods. Specifically:

- Widgets A1 or B1 can be added as children to parents P1 or P2, or as peers to master M1
- Widgets A2 or B2 can be added as peers to masters A1, B1, or M1

If you click and drag any of the colored widgets (excluding the buttons), you'll see that they all move independently at first. None of them are attached to children or peers. You can click on the buttons in any order, but you should try the following sequence first:

1. Click the first button in the first row to make A1 a child of P1. Drag A1 to see that the widget is clipped by its parent. Drag P1 to see that A1 moves as well.

2. Click the last button in the first row to make B2 a peer of B1. Drag B1 to see that B2 moves as well.

3. Click the second button in the first row to make B1 a child of P2. Notice that B2, as B1's peer, becomes a child of P2 as well. Drag B1, B2, and P2 to see the effect of the parent/child and master/peer relationships on widget positioning and clipping.

4. Click the third button in the first row to make A2 a peer of A1. Notice that A2 becomes a child of P1 as well. Drag A1, A2, and P1 to see the effect of the parent/child and master/peer relationships on widget positioning and clipping.

Experiment with different combinations and sequences of `addChild` and `addPeer` method calls until you're comfortable with these concepts. To reset the example to its starting point, press the **Reload** button in your web browser. If you find that a widget disappears when added as a child of P1 or P2, click on the appropriate button in the bottom row to bring it back to the page level as a peer of M1. If the widget is more than 200 pixels from the top and/or left of the page, it will lie outside the display area of P1 or P2 (each of which are 200 pixels square) when added as a child (remember that a child gets its positioning origin from its parent!). Move the widget closer to the top-left corner of the page, then try adding it as a child of P1 or P2 again.

Hit the **Reload** button on your browser to reset this example.

Note

This example only demonstrates the effects of parent/child and master/peer relationships on widget positioning and clipping. Widget containment and attachment have a number of other effects, including the "bubbling" of events from children to their parents. These effects will be discussed in subsequent chapters.

Widget contents

In the "hello world" example earlier in this chapter, a simple canvas widget containing the text `"Hello world!"` was created. That example initialized the `contents` property of the widget. This property can be changed thereafter using the `setContents(`*`newContents`*`)` setter method. The *Canvas* and *Label* classes are the only widget classes that directly display the value of their `contents` property; other widget classes generate their contents based on other properties and/or data.

Table 1.3: Canvas and Label contents properties and methods

Property / Method	Description/Action
`contents`	The contents of a canvas or label widget. Any HTML string is acceptable.
`setContents(`*`newContents`*`)`	Changes the contents of a widget to *`newContents`*, an HTML string.

As mentioned in the earlier discussion of widget containment, a widget will redraw its own contents only if it has no children. You can initialize the contents of a parent widget before it is drawn for the first time, but you cannot change these contents afterwards. In general, you should use only *Canvas* widgets with empty contents as parents. Their children can display whatever dynamic content you want.

If you need to embed images in a canvas widget's contents, there are several methods and properties for referencing, creating, and changing images that will make your job easier. Refer to Chapter 4, *"Images and Skins,"* for details.

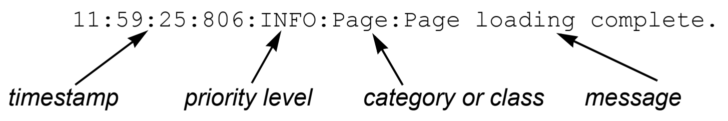
JavaScript debugging

In any page in which ISC has been loaded, you have access to the Developer Console, which can be opened by entering the following URL into your browser from the running application:

```
javascript:isc.showLog()
```

The Developer Console contains a "Results" pane that displays a list of diagnostic messages logged by the SmartClient framework. You can enable and disable diagnostic messages for specific subsystems, and you can also log your own messages. Because important diagnostic messages may be logged at any time, you should have the Developer Console open whenever you are working with SmartClient.

Log messages contain a timestamp, priority level, category or class of object, and object ID.



```
11:59:25:806:INFO:Page:Page loading complete.
```

To log messages of your own, you can call one of the logging methods listed below on any widget object instance, or on the `Log` class itself.

Table 1.4: Methods for logging messages

Method	Description
`logMessage([`*`priority`*`], `*`message`*`, [`*`category`*`])`	Writes a message to the log given by one of the following priority constants (from highest priority to lowest): • `Log.FATAL:1`—the application should probably abort • `Log.ERROR:2`—actions within the application may be unpredictable • `Log.WARN:3`—something that requires programmer attention • `Log.INFO:4`—**(default)** something that is merely additional information for the programmer • `Log.DEBUG:5`—detailed non-vital information for debugging purposes If the optional *`priority`* parameter is not provided, the message will be assigned the default .The optional *`category`* parameter allows the programmer to classify different log messages or track what class logged the message. If the *`category`* parameter is not specified, the the class of the object on which the log method was called will be recorded in the log window instead.
`logFatal(`*`message`*`, [`*`category`*`])`	Writes a `Log.FATAL` priority message to the log.
`logError(`*`message`*`, [`*`category`*`])`	Writes a `Log.ERROR` priority message to the log.
`logWarn(`*`message`*`, [`*`category`*`])`	Writes a `Log.WARN` priority message to the log.
`logInfo(`*`message`*`, [`*`category`*`])`	Writes a `Log.INFO` priority message to the log.
`logDebug(`*`message`*`, [`*`category`*`])`	Writes a `Log.DEBUG` priority message to the log.
`logIsEnabledFor(`*`priority`*`, [`*`category`*`])`	Returns `true` if logging is enabled for the specified *`priority`* and *`category`* (if supplied), and `false` otherwise.
`logIsDebugEnabled([`*`category`*`])`	Returns `true` if logging is enabled for the the `Log.DEBUG` priority and *`category`* (if supplied), and `false` otherwise.
`logIsInfoEnabled([`*`category`*`])`	Returns `true` if logging is enabled for the the `Log.INFO` priority and *`category`* (if supplied), and `false` otherwise.

Each logging category or class may have a default logging priority level that denotes the minimum level before messages are sent to the log. In other words, if the priority of a logged message is lower than the level of its encompassing category or class, the message will not appear in the log. This allows you to see only the debug messages relevant to the issue you are working on.

Warning

Never use the native `alert` function for diagnostic messages. Among other issues, `alert` can affect timing, masking or altering the behavior you are trying to debug. SmartClient's logging system doesn't suffer from these problems and provides much more control and flexibility.

You can set all priority defaults to the levels of your choosing. See the following section on *"Setting logging priorities for categories and classes"* for more information.

The Isomorphic SmartClient framework ships with a `defaultPriority` property set to `Log.INFO`. This default priority will be applied to a message if no value is given for the *`priority`* parameter, and it will also be applied to the category unless the category is explicitly set.

Warning If you log a message to a category without explicitly defining the category's priority, all `Log.DEBUG` messages logged within that category will be suppressed from the log. The ISC default assigns a priority of `Log.INFO` to all categories unless you explicitly set them otherwise.

If the priority of the message being sent to the log window is `Log.ERROR` or `Log.FATAL`, the error message will also be displayed in a JavaScript alert and a stack trace will be added to the message. This `stackTracePriority` can also be set as you choose.

Note Stack traces are currently not available in Mozilla and Netscape 7 browsers.

For example, consider the following `canvasExample` instance. To track widget instantiation and redraw within the `canvasExample` object, the `logInfo` method is called to send informational messages to the log window:

```
. . .
<SCRIPT>
Canvas.create({
   ID:"canvasExample",
   left:10,
   top:10,
   width:100,
   height:30,
   backgroundColor:"lightBlue",
   contents:"Hello World!"
});
canvasExample.logInfo("Canvas widget object instantiated!");

var width = canvasExample.getWidth();
var height = canvasExample.getHeight();
canvasExample.setContents("Width: " + width + "<BR>Height: " + height);
canvasExample.redraw();

canvasExample.logInfo("Width/Height contents set and canvas redrawn!",
   "redraw");
</SCRIPT>
. . .
```

Using the log system statically and in class instances

The log system can be used statically by calling one of the methods in Table 1.4 on the `Log` class:

```
    Log.logInfo("Log message.");
```

You can also log within the scope of an entire class of widgets, or use the log system in class instances by calling methods using the `this` keyword. For example, suppose an information message is sent to the log window whenever a user clicks on the `canvasExample` widget.

```
. . .
<SCRIPT>
Canvas.create({
    ID:"canvasExample",
    left:10,
    top:10,
    width:100,
    height:30,
    backgroundColor:"lightBlue",
    contents:"Hello World!",
    click:"this.logInfo('Ouch! Quit clicking on me!');"
});

Log.logInfo("This message is being logged in the global scope.");
Canvas.logInfo("This message to the Canvas scope may also be appropriate.");
. . .
```

Note

All the methods given in Table 1.4 are inherited by all classes and widgets. Therefore you can call these methods on any class or defined object. As a rule of thumb, call methods on the `this` keyword from within a class instance method, and on the `Log` singleton class from a global method or where the scope is unclear.

Setting logging priorities for categories and classes

The "Preferences" pane within the Developer Console can be used to dynamically change the default level of logging for a category or class. The Preferences pane contains a list of categories and widgets which have useful debugging information. Hover over any category name to see a short description of the kind of information that is logged in that category.

Keep in mind that if you set any category to a level of `Log.DEBUG`, you may be getting more messages in your log window than you intended! It is best to define your own categories for use with the `Log.DEBUG` priority to avoid getting messages that you may not want or need in debugging your own code.

If you are going to build a log message by compiling information from a bunch of objects into a string, use the `logIsEnabledFor` method or an appropriate derivative to ensure that the message won't be suppressed from the log prior to making the call, thus saving execution time. For example, suppose a function that encapsulates the mouse movement data is compiled when a user clicks on the `canvasExample` widget only when the Canvas class is set to `Log.DEBUG` priority:

```
. . .
<SCRIPT>
Canvas.create({
    ID:"canvasExample",
    left:10,
    top:10,
    width:100,
    height:30,
    backgroundColor:"lightBlue",
    contents:"Hello World!"
    click:"if (this.logIsDebugEnabled()){this.logInfo(getMouseMoveReport());}"
});
. . .
```

CHAPTER 2

Drawing Widgets

Widgets are visual interface elements, and as such have a common set of characteristics related to drawing and visual appearance. The *Canvas* superclass provides the properties and methods used by all widgets to control: position, scrolling, size, background attributes, visibility, layering, cascading style sheet (CSS) classes (styles), clipping, padding, borders, and margins.

This chapter details the process of using the *Canvas* superclass to control the drawing properties of all ISC widgets.

In this chapter:

Specifying initial position and size

When you call the `create` method to create a new instance of a widget, you can specify the widget's initial position and size with the properties described in Table 2.1.

Table 2.1: Widget positioning and sizing properties

Property	Value	Default
`position`	`absolute` or `relative`, corresponding to the "absolute" (with respect to parent) or "relative" (with respect to main page flow) values for the CSS position attribute. You will almost always use `absolute`.	`"absolute"`
`left`	Number of pixels the left side of the widget is offset to the right from its default drawing context (either its parent's top-left corner, or the document flow, depending on the value of the `position` property).	`0`
`top`	Number of pixels the top of the widget is offset down from its default drawing context (either its parent's top-left corner, or the document flow, depending on the value of the `position` property).	`0`
`width`	Number of pixels for the widget's horizontal dimension.	`100`
`height`	Number of pixels for the widget's vertical dimension.	`100`

For example:

```
Canvas.create({
    ID:"myWidget",
    left:200,
    top:100,
    width:50,
    height:50,
    backgroundColor:"khaki"
});
```

Note The `backgroundColor` property is covered *"Other visual properties" on page 39*. It is included here so that if you experiment with this code snippet, the canvas will appear in a different color than the browser page background.

In a real application, widget positioning and sizing should be managed by *Layouts*. For usage information on Layouts, consult the *SmartClient Online Reference*. If you want to embed widgets in an otherwise standard HTML page, use relative positioning. This allows you to immediately integrate ISC widgets into existing sites.

Drawing

A call to the `create` method will, by default, draw the new widget. If, however, you set the widget's `autoDraw` property to `false`, you need to explicitly call the `draw` method for it to be displayed.

```
Canvas.create({
    ID:"myWidget",
    contents:"Aloha!",
    autoDraw:false
});
myWidget.draw();
```

In most cases, you will simply allow the widget to be drawn automatically. However, the `draw` method can be called by a script at any time. For example, a user action such as pressing a button could call the `draw` method to display some new widget. In some situations, you may want to draw a widget only when the page is fully loaded. To defer drawing in this case, use the following syntax:

```
Page.setEvent("load", "myWidget.draw()");
```

Warning

Widgets that have relative positioning should be drawn before page load. Otherwise, they will be treated like widgets with absolute positioning. See "*Specifying initial position and size*" *on page 28* for information on positioning and sizing of widgets.

The `Page.setEvent` mechanism is explained in *"Handling Page events" on page 44*.

The `draw` method is also called as a side effect of several other widget methods:

- The `show` method draws the widget being shown if it is not already drawn.
- The `addChild` method draws the new child if its parent is already drawn.
- The `addPeer` method draws the new peer if its master is already drawn.

Drawing-related methods

To determine whether an existing widget has been drawn, call the `isDrawn` method. For example, the following script will draw widget2 only if widget1 has already been drawn:

```
if (widget1.isDrawn()) widget2.draw();
```

If the data behind a widget object changes, the widget may need to be redrawn. There are two methods you can use to initiate a redraw action.

- The `redraw()` method will redraw the widget immediately.
- The `markForRedraw()` method marks the widget object as "dirty" so that it will be added to a queue for redraw. After an infinitesimal time delay, all widget instances added to this queue will be redrawn in the order they were added. This allows multiple properties to be set before a widget is redrawn on the page, thus improving performance since the widget will not needlessly redraw multiple times.

The drawing-related methods for widgets are summarized in Table 2.2.

Table 2.2: Widget drawing-related methods

Method	Action
`draw()`	Draws the widget on the page, if the `autoDraw` property has been set to `false`.
`isDrawn()`	Returns the boolean `true`, if the widget has been drawn, and `false` otherwise.
`redraw()`	Redraws the widget immediately with its current property values.
`markForRedraw()`	Marks the widget as "dirty" so that it will be added to a queue for redraw. Once a small lag time has elapsed, each widget added to the queue is then redrawn with its current property values.

Controlling position and size

Once you have created and initialized a new widget with the `create` method, you should never directly set the properties controlling position and size of the widget. Instead, call the setter methods described in Table 2.3. Using these methods ensures that the widget will be redrawn correctly, along with any associated widgets (children or peers).

Note

The parameters of these methods are all expressed in pixel units, relative to the widget's drawing context (page, parent, or in-line position).

Table 2.3: Widget positioning and sizing setter methods

Method	Action
`setRect([`*`left`*`], [`*`top`*`], [`*`width`*`], [`*`height`*`])`	Moves the widget so that its top-left corner is at the specified top-left coordinates, and resize it to the specified width and height. The `setRect` method will also accept a single parameter as an object array with *`left`*, *`top`*, *`width`*, and *`height`* given as properties.
`setLeft(`*`left`*`)`	Moves the widget so that its left side is at the specified coordinate.
`setTop(`*`top`*`)`	Moves the widget so that its top side is at the specified coordinate.
`setWidth(`*`width`*`)`	Resizes the widget horizontally to the specified *`width`* (moves the right side of the widget). The *`width`* parameter can be expressed as a percentage of viewport size, or as the number of pixels.
`setHeight(`*`height`*`)`	Resizes the widget vertically to the specified *`height`* (moves the bottom side of the widget). The *`height`* parameter can be expressed as a percentage of viewport size, or as the number of pixels.
`setRight(`*`right`*`)`	Resizes the widget horizontally to position its right side at the specified coordinate.
`setBottom(`*`bottom`*`)`	Resizes the widget vertically to position its bottom side at the specified coordinate.
`moveBy([`*`deltaX`*`], [`*`deltaY`*`])`	Moves the widget *`deltaX`* pixels to the right and *`deltaY`* pixels down. Passes negative numbers to move up and/or to the left.

Method	Action
`moveTo([`*`x`*`], [`*`y`*`])`	Moves the widget so that its top-left corner is at the specified *`x`*, *`y`* coordinates. The `moveTo` method will also accept a single parameter as an object array with *`x`* and *`y`* given as properties.
`resizeBy([`*`deltaX`*`], [`*`deltaY`*`])`	Resizes the widget, adding *`deltaX`* to its width and *`deltaY`* to its height (moves the right and/or bottom sides of the widget).
`resizeTo([`*`width`*`], [`*`height`*`])`	Resizes the widget to the specified *`width`* and *`height`* (moves the right and/or bottom sides of the widget). The *`width`* and *`height`* parameters can be expressed as a percentage of viewport size, or as the number of pixels.

Tip

If you need to set both left and top coordinates simultaneously, you should use one `moveTo` or `setRect` method call rather than two separate method calls to `setLeft` and `setTop`. Similarly, separate calls to `setWidth` and `setHeight` should be combined into a single method call to `resizeTo` or `setRect`.

Setting the left or top coordinate moves a widget, while setting the bottom or right coordinate resizes the widget. To resize a widget from its left or top side, you must set the widget's new width or height in addition to its new left or top coordinate. These two operations may be performed separately, or simultaneously via the `setRect` method. Both approaches are demonstrated in the *"Example: Dynamic positioning and sizing" on page 32*.

The Isomorphic SmartClient widgets also provide a collection of 'getter' methods for accessing information about their positions and sizes (in pixel units). These methods are described in Table 2.4.

Table 2.4: Widget positioning and sizing getter methods

Method	Action
`getLeft()`	Returns the left coordinate of the widget, relative to its drawing context.
`getTop()`	Returns the top coordinate of the widget, relative to its drawing context.
`getWidth()`	Returns the width of the widget.
`getHeight()`	Returns the height of the widget.
`getVisibleWidth()`	Returns the visible width of a clipped widget (see *"Clipping and scrolling" on page 36*). If a widget is not clipped, `getVisibleWidth()` returns the full width.
`getVisibleHeight()`	Returns the visible height of a clipped widget (see *"Clipping and scrolling" on page 36*). If a widget is not clipped, `getVisibleHeight()` returns the full height.
`getRight()`	Returns the right coordinate of the widget, relative to its drawing context.
`getBottom()`	Returns the bottom coordinate of the widget, relative to its drawing context.

Method	**Action**
`getPageLeft()`	Returns the global left coordinate of the widget on the page.
`getPageTop()`	Returns the global top coordinate of the widget on the page.

The `getPageLeft` and `getPageTop` methods are useful when you need to work with global coordinates (e.g. determining the position of the mouse pointer).

Two related utility methods are provided for your convenience. These methods are described in Table 2.5.

Table 2.5: Widget positioning and sizing utility methods

Method	**Action**
`containsPoint(`*`x`*`, `*`y`*`)`	Returns `true` if this widget's rectangle, in global (page) coordinates, contains the point (*x*,*y*). Note that this method uses `getVisibleWidth()` and `getVisibleHeight()`. See *"Clipping and scrolling" on page 36*.
`intersectsRect(`*`left`*`, `*`top`*`, `*`width`*`, `*`height`*`)`	Returns `true` if the rectangle of this widget intersects with the rectangle coordinates passed in, and `false` otherwise.
`intersects(`*`otherWidget`*`)`	Returns `true` if the rectangles of this widget and *`otherWidget`* overlap. Also returns `true` if *`otherWidget`* is a child of this widget, regardless of whether the two overlap.

Example: Dynamic positioning and sizing

The example file `widget_position_size.html` (shown in Figure 2.1) uses the setter and getter methods above to demonstrate dynamic positioning and sizing of a widget.

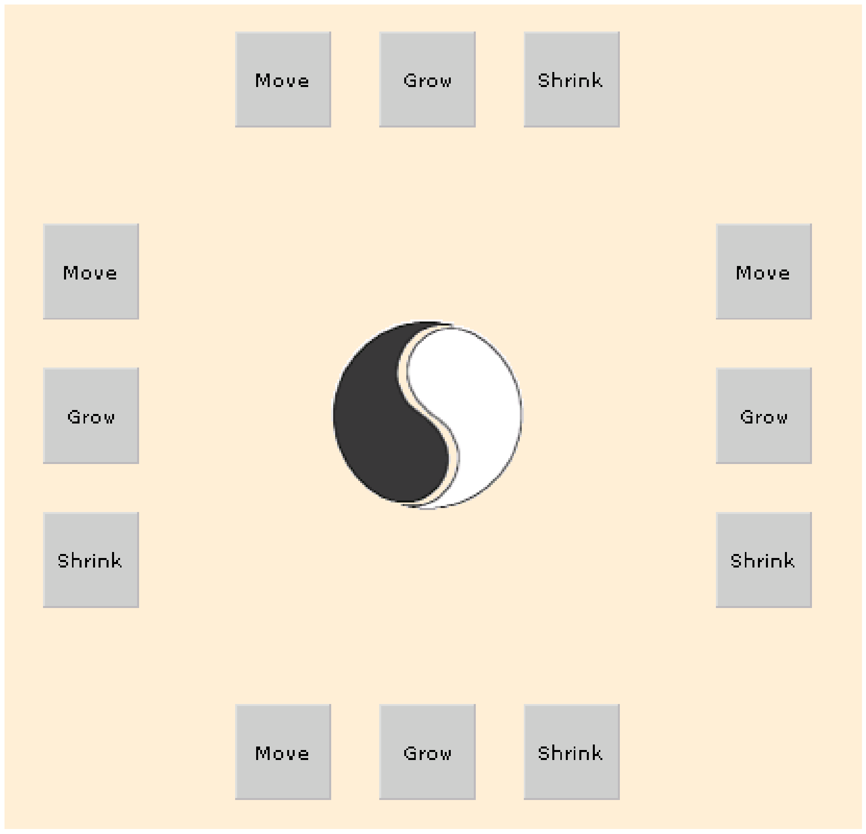


Figure 2.1: Example of dynamic postioning and sizing

Click and hold on the **Move**, **Grow**, or **Shrink** buttons on any side of the image to move, grow, or shrink the widget in/from that direction.

The methods listed in this section allow for more than one approach to any positioning or sizing operation. For each operation in this example, the file provides two different scripts that could achieve the same result. The first script (which is commented out) uses simple `get` and `set` methods only, while the second script uses the more efficient `moveBy` and `resizeBy` methods. For example, the **Move** button to the right of the widget could execute either of the following scripts (and actually executes the second one):

```
widget.setLeft(widget.getLeft()+20)
```

or

```
widget.moveBy(20,0)
```

Grow and shrink operations from the top or left sides of the widget are slightly more complicated, since the widget must be both resized and repositioned. For example, the **Grow** button above the widget could execute either of the following scripts:

```
widget.setRect(null, widget.getTop()-20, null, widget.getHeight()+20);
```

or

```
widget.moveBy(0,-20);
widget.resizeBy(0,20);
```

Shrink operations also contain some simple logic to prevent the widget from being further shrunk or repositioned when its width or height is already zero. For example, the **Shrink** button to the left of the image could execute either of the following scripts:

```
widget.setRect(widget.getWidth() > 0 ? widget.getLeft()+20 : null,
   null, Math.max(widget.getWidth()-20,0), null);
```

or

```
widget.moveBy(widget.getWidth() > 0 ? 20 : 0, 0);
widget.setWidth(Math.max(widget.getWidth()-20,0));
```

More complex logic to prevent the widget from being grown or moved outside of a defined area can be accomplished using the same methods described above.

Showing and hiding

All widgets have a `visibility` property, corresponding to the cascading style sheet (CSS) visibility attribute, that can be initialized to one of the values listed in Table 2.6.

Table 2.6: Widget visibility property values

Value	Effect
`"inherit"`	**(Default)** The widget's visibility matches that of its parent. This corresponds to the 'inherit' value of the CSS visibility attribute. This generally means the widget will be visible.
`"visible"`	The widget is always visible, even when its parent is hidden. Corresponds to the 'visible' value of the CSS visibility attribute.
`"hidden"`	The widget is always hidden, even when its parent is visible. Corresponds to the 'hidden' value of the CSS visibility attribute.

Like most widget properties, visibility should not be directly accessed after initialization. Methods to control widget visibility are listed in Table 2.7.

Table 2.7: Widget visibility methods

Method	Action
`show()`	Sets the widget's CSS visibility attribute to `"inherit"`. If the widget has not yet been drawn, this method calls the `draw` method as well.
`hide()`	Sets the widget's CSS visibility attribute to `"hidden"`.
`isVisible()`	Returns `true` if the widget's CSS visibility attribute is set to `"visible"` and `false` if the value of this attribute is `"hidden"`. If the visibility value is set to `"inherit"` and the widget's parent is visible, `isVisible()` returns `true`. Conversely, if the visibility value is set to `"inherit"` and the widget's parent is *not* visible, it returns `false`.

To show or hide a widget, use the first two methods above. To determine whether a widget is visible, call the `isVisible` method. For example, the following script will move `myWidget` only if it is visible:

```
if (myWidget.isVisible()) myWidget.moveTo(0,200);
```

Opacity

All widgets also have an `opacity` property that allows you to draw a widget so that it appears partly transparent. This property will only work with Internet Explorer web browsers on Windows operating systems and should be used with discretion. Calculating opacity is a processor-intensive task that may be time consuming for some systems to render. Be aware that even with the `opacity` property set to its maximum (fully opaque), a widget may still exhibit a slower draw response. For this reason, the `opacity` property is set to `null` by default, so that it will be ignored by the system and ensure the fastest draw times where photo-realistic transparency is not required. Possible values for the `opacity` property are summarized in Table 2.8.

Table 2.8: Widget opacity property values

Value	Effect (IE only)
`0` (min) to `100` (max)	**Default:** `null` Renders the widget to be partly transparent. A widget's `opacity` property may be set to any number between `0` (transparent) to `100` (opaque).

Like most widget properties, opacity should not be directly accessed after initialization. The method used to set widget opacity is listed in Table 2.7.

Table 2.9: Widget opacity method

Method	Action
`setOpacity(`*`newOpacity`*`)`	Sets the opacity for the widget to the *`newOpacity`* value. This *`newOpacity`* value must be within the range of `0` (transparent) to `100` (opaque).

Layering

The layering, or stacking order, of widgets is typically determined by the order in which they are drawn. Widgets drawn earlier are 'behind' widgets in the same parent that are drawn later. If sibling widgets overlap, the widget drawn later is in front. This concept is often called "z-ordering." Widgets contained in different parents cannot be interleaved.

Widget layering methods allow you to change the default layering of a widget within its parent. These methods are described in Table 2.10.

Table 2.10: Widget layering methods

Method	Action
`bringToFront()`	Puts this widget at the top of the stacking order, so it appears in front of all other widgets in the same parent.
`sendToBack()`	Puts this widget at the bottom of the stacking order, so it appears behind all other widgets in the same parent.
`moveAbove(`*`otherWidget`*`)`	Puts this widget just above *`otherWidget`* in the stacking order, so it appears in front of *`otherWidget`*.
`moveBelow(`*`otherWidget`*`)`	Puts this widget just behind *`otherWidget`* in the stacking order, so it appears behind *`otherWidget`*.

Example: Dynamic layering

The example file `widget_layering.html` (shown in Figure 2.2) uses the methods above to demonstrate dynamic layering of a widget.

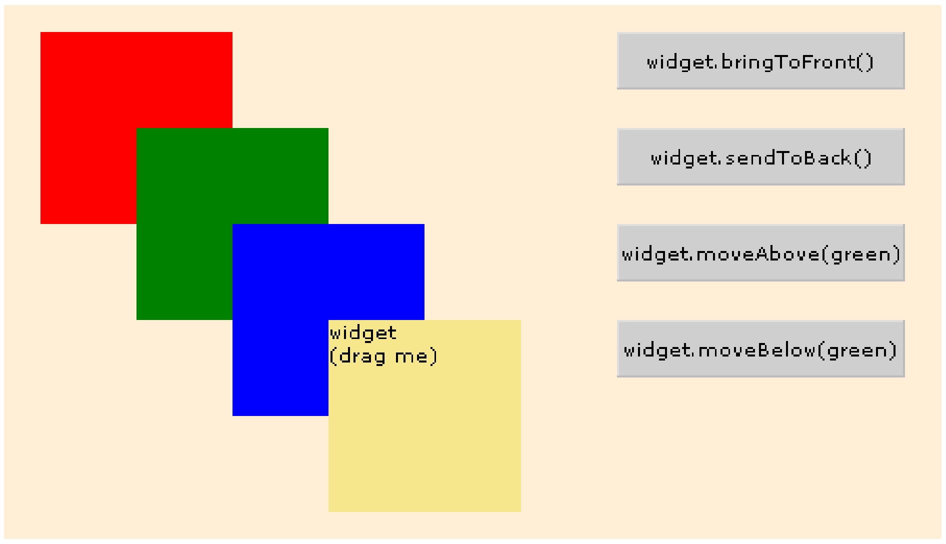


Figure 2.2: Example of dynamic layering

This example draws four overlapping *Canvas* widgets, the last of which, widget (drag me), is mouse-draggable. Dragging this or any other draggable widget automatically brings it to the top layer. This draggable widget can then be dynamically relayered by clicking on the buttons to the right. Note that the buttons are also drawn in the stacking order, so the draggable widget can be moved above or below them as well.

Clipping and scrolling

Widget clipping and scrollbars are controlled by the `overflow` property, which is initialized when a widget instance is created and set to one of the overflow values listed in Table 2.11.

Table 2.11: Widget overflow values

Value	Effect
`"visible"`	**(Default)** Content that extends beyond the widget's width or height is displayed.
`"hidden"`	Content that extends beyond the widget's width or height is clipped (hidden).
`"auto"`	Horizontal and/or vertical scrollbars are displayed only if necessary. Content that extends beyond the remaining visible area is clipped.
`"clip-h"`	Content that extends beyond the widget's width is clipped. All vertical content is displayed.
`"clip-v"`	Content that extends beyond the widget's height is clipped. All horizontal content is displayed.
`"scroll"`	Horizontal and vertical scrollbars are always drawn inside the widget. Content that extends beyond the remaining visible area is clipped, and can be accessed via scrolling.
`"ignore"`	Clipping is ignored by the ISC system. This setting may be used for improved performance, with frequently-drawn widgets whose dimensions always agree exactly with the size of their contents.

Note

Note that the `overflow` property does not correspond directly to the CSS overflow attribute. It extends the standard CSS overflow effects, and makes them consistent across browsers.

When a widget is clipped, only a portion of its content is displayed. The methods described in Table 2.12 let you determine the size of the complete contents, and scroll the contents to display a different portion (regardless of whether scrollbars are shown).

Table 2.12: Widget scrolling methods

Method	Action
`getScrollWidth()`	Returns the scrollable width of the widget's contents, including children, ignoring clipping.
`getScrollHeight()`	Returns the scrollable height of the widget's contents, including children, ignoring clipping.
`getVisibleWidth()`	Returns the clipped width of the widget's contents.
`getVisibleHeight()`	Returns the clipped height of the widget's contents.
`scrollTo([`*`left`*`], [`*`top`*`])`	Scrolls the content of the widget so that the origin (top-left corner) of the content is *`left`* pixels to the left and *`top`* pixels above the widget's top-left corner (but still clipped by the widget's dimensions).

Example: Widget overflow (clipping and scrolling)

The example file `canvas_clip_scroll.html` (shown in Figure 2.3) uses the `overflow` property and `scrollTo` method to demonstrate widget clipping and scrolling.

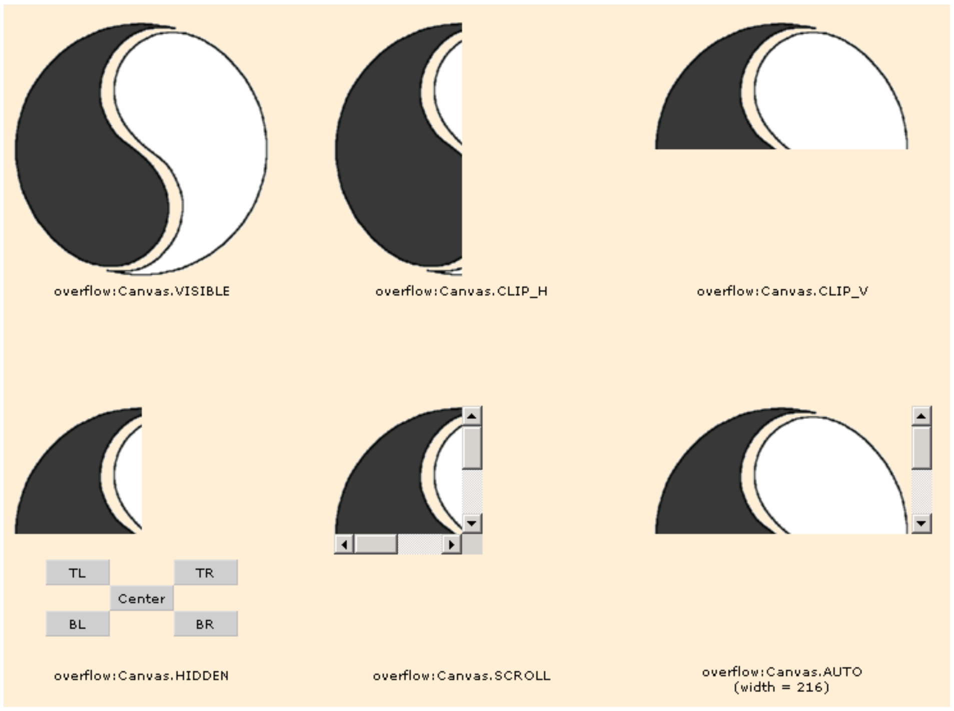


Figure 2.3: Example of widget overflow (clipping and scrolling)

This example draws six *Canvas* widgets, each containing a 200-by-200-pixel image. Except for the last widget, they are all initialized with the default dimensions of 100-by-100 pixels. Each widget demonstrates the effect of a different overflow value when the size of the contents exceed the size of the widget.

The five buttons below the widget with `overflow:hidden` (named "`hiddenCanvas`" in the code) call the `scrollTo` method to shift the contents in that widget. For example, the **BL** button shifts the contents to display the bottom-left corner by calling:

```
hiddenCanvas.scrollTo(0,100)
```

The widget with `overflow:auto` is initialized with a width of 216 pixels, to demonstrate the difference between the `scroll` and `auto` settings. A width of 216 pixels accommodates both the 200-pixel-wide image, and the 16-pixel-wide vertical scrollbar, so no horizontal scrollbar is necessary. Since scrollbars are only drawn when necessary if `overflow` is set to `auto`, only a vertical scrollbar is drawn in this case.

Note

Scrollbars are drawn inside the boundaries of a widget, so you may need to take their width of 16 pixels into account when sizing a widget. When you are using the `auto` setting for `overflow`, you should also keep in mind that the appearance of a scrollbar for one dimension will reduce the viewable area of a widget in the other dimension, possibly causing the other scrollbar to appear. This would be the case if the width of the last widget were less than 216 pixels.

If you are using Windows XP, scrollbars have a width of 17 pixels, so for the Windows XP case, the width would need to be set to 217 pixels to prevent the horizontal scrollbar from appearing. This is reflected in the source code for the example.

Other visual properties

The *Canvas* class provides a handful of other properties to control various apearance attributes of widgets. Table 2.13 lists properties you can initialize when creating a new widget instance.

Table 2.13: Widget appearance properties

Property	Value	Default
`className`	The CSS class applied to this widget as a whole.	`"normal"`
`backgroundColor`	The background color for this widget. It corresponds to the CSS background-color attribute. You can set this property to an RGB value (e.g. `#22AAFF`) or a named color (e.g. `red`) from a list of browser supported color names.	`null`
`backgroundImage`	The background image file for this widget. See *"Specifying image directories" on page 78* for details on referencing image files. It corresponds to the CSS background-image attribute. The filename is automatically prepended with `Page.imgDir` and `widget.imgDir`.	`null`
`backgroundRepeat`	Specifies how the background image should be tiled if this widget is larger than the image. It corresponds to the CSS background-repeat attribute and is given as one of the following four values. • `repeat` • `no-repeat` • `repeat-x` • `repeat-y`	`"repeat"`
`margin`	The thickness, in pixels, of the transparent space outside the borders of this widget, buffering it from other in-line content. It corresponds to the CSS margin attribute.	`0`
`padding`	The thickness, in pixels, of the space between this widget's content and its borders. It corresponds to the CSS padding attribute.	`0`

Property	Value	Default
`border`	CSS border width, border style, and/or color, to apply to all four borders of this widget. It corresponds to the CSS border attribute.	`null`
`cursor`	Specifies the cursor image to display when the mouse pointer is over this widget. It corresponds to the CSS cursor attribute and is given as one of the following five values: • `default` (arrow cursor) • `wait` • `hand` • `move` • `help` • `text` • `crosshair`	`"default"`

Warning

These properties may be overridden by different widget classes as necessary to implement class-specific functionality. Use them with caution for widget classes other than *Canvas*.

Several appearance properties from Table 2.13 have corresponding setter methods. These methods are described in Table 2.14.

Table 2.14: Widget appearance setter methods

Method	Action
`setBackgroundColor(`*`color`*`)`	Sets the background color for this widget to *`color`*.
`setBackgroundImage(`*`imageURL`*`)`	Sets the background to an image file given by the *`imageURL`*. This URL should be given as a string relative to the image directory for the page (`./images` by default). See Chapter 4, *"Images in Canvas widgets" on page 80* for more information on referencing image files
`setClassName(`*`CSSclass`*`)`	Sets the CSS class for this widget to *`CSSclass`*.
`setCursor(`*`cursor`*`)`	Sets the cursor for this widget to *`cursor`* (see the `cursor` property in Table 2.13 for possible values).

The other appearance-related properties should be set during initialization only.

CHAPTER 3

Handling Events

As the building blocks of interactive applications, Isomorphic SmartClient widgets can respond to events generated by their environment or a user. The ISC event model enables consistent handling of:

- page events (e.g., page loaded),
- mouse events (e.g., mouse button clicked),
- drag-and-drop events (e.g., dragging started), and
- keyboard events (e.g., key pressed).

This chapter describes the Isomorphic SmartClient event-handling architecture and presents how to handle events within ISC applications.

In this chapter:

The ISC event model

The ISC system provides a predictable cross-browser event-handling mechanism for ISC widgets. Events can be handled both at the page level (i.e., globally), and at the level of individual widgets.

With the exception of a few page-specific events ('`load`', '`unload`', '`idle`' and '`resize`'), events are processed in the following sequence:

1. The event is sent to any global (page-level) event handlers. These handlers can cancel further propagation of the event by returning `false`.

2. If the event occurred on a form element or a link, it is passed on to the browser so that the element will perform its default action. No widget receives the event.

3. If the event occurred on an enabled widget (but not on a form element or link inside the widget), it is sent to that widget's event handler, if any. This handler can cancel further propagation of the event by returning `false`.

4. The event is "bubbled" up to the widget's parent in the containment hierarchy, if any. Again, the parent's handler for the event can cancel further propagation by returning `false`. This step is repeated, with the event "bubbling" up through the containment hierarchy, until a top-level widget is reached or the event is explicitly canceled.

Note

Canceling propagation of an event may cancel its side effects as well, including the generation of other events. For example, if a global `mouseDown` handler returns `false`, drag-and-drop events will not be generated. Specific effects are discussed in the descriptions of the various events in the following sections.

In brief, the ISC event model offers the best features of browser event models:

- *Page-first event handling* allows you to reliably process or cancel any event before it affects the objects on the page.
- *Event "bubbling"* ensures that parent widgets receive events sent to their children, and allows you to create generalized parent-level handlers rather than duplicating code in each child.

Note

Isomorphic SmartClient libraries will not interfere with native event handling when events occur outside of a target widget. You can therefore have HTML that is not ISC-based on the same page as widget objects that will react to native events as you would expect.

Example: Event propagation

The example file `event_propagation.html` (shown in Figure 3.1) demonstrates this flow for the '`click`' event.

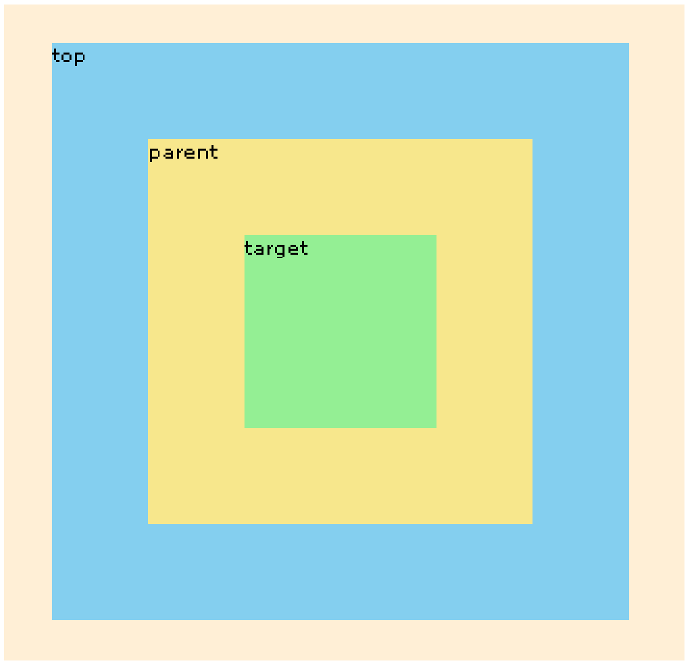


Figure 3.1: Example of event propagation

In this example, three nested widgets, **top**, **parent**, and **target**, all have click handlers assigned to them. Each widget's handler displays a confirmation dialog, and returns the value from that dialog (`true` for **OK**, `false` for **Cancel**). For example, the script for target's `click` handler is:

```
return confirm('target received click event. Continue?')
```

The page also has a global `click` handler that reports the event in the window status bar at the bottom left of the browser window:

```
return confirm('Page received click event. Continue?')
```

Click on **target**, and click **OK** in each of the confirmation dialogs to send a '`click`' event through the system. Figure 3.2 depicts the propagation of this '`click`' event through the containment hierarchy. Dashed arrows represent the path of the '`click`' event, while dashed boxes represent the event handlers that execute in response.

If you click **Cancel** in any of the confirmation dialogs, the handler that called it will return `false`, thereby canceling event propagation. Note that the event can be canceled by any of its handlers, at the page level as well as in the widgets.

Note

The source code for this example packaged with the Isomorphic SmartClient SDK shows both children declaration methods. See *"Nesting children within a parent widget declaration" on page 18* for more information.

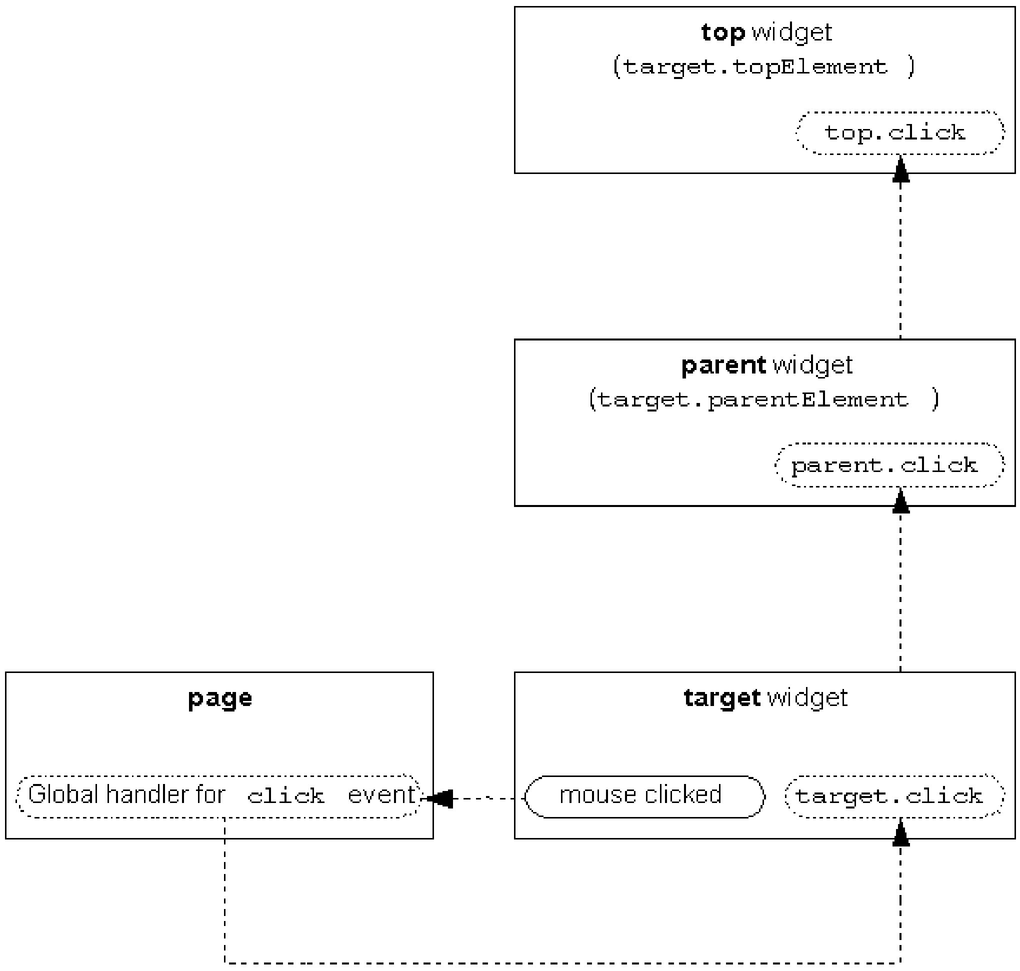


Figure 3.2: Click event propagation

Handling *Page* events

Global (a.k.a. "page-level") event handlers can be set and cleared using two methods of the *Page* object. The *Page* object is a special non-widget ISC class that provides various static properties and methods related to the current page as a whole. There are two methods for manipulating global event handlers, described in Table 3.1.

Table 3.1: Global event handler methods

Method	Action
`Page.setEvent(`*`eventName`*`, `*`handler`*`, [`*`Page.FIRE_ONCE`*`])`	Sets *`handler`* (either a string to evaluate or a reference to a function) as a global handler for the *`eventName`* event. If the *`Page.FIRE_ONCE`* constant is provided as an optional third parameter, this handler will be automatically removed after executing once. Returns a unique ID and eventID for this handler, which can be used later in a call to the `Page.clearEvent` method.
`Page.clearEvent(`*`eventName`*`, [`*`eventID`*`])`	Clears the global handler with ID *`eventID`*, for the *`eventName`* event. If no *`eventID`* is specified, the method will clear all global handlers for the *`eventName`* event.

The handler parameter of the `Page.setEvent` method can be either a function, or a string of script to evaluate. Both approaches are shown later in this section. Note that multiple global handlers may be assigned to a single event. The `Page.setEvent` method adds a new handler without removing any existing handlers for that event. If you want to clear a global handler, you must do so explicitly by calling the `Page.clearEvent` method.

Any event in the ISC system may have global handlers assigned to it. For the mouse events and drag-and-drop events discussed below, global handlers provide a mechanism for processing and/or canceling an event before it is sent to its target. For page events, global handlers are the only handlers that can respond to the event.
Table 3.2 lists the page events currently supported in the ISC system.

Table 3.2: ISC *Page* events

Event	Description
`load`	Executed when the page has finished loading. It corresponds to the browser '`load`' event normally handled by `window.onload`.
`unload`	Executed when the page is exited or unloaded. It corresponds to the browser '`unload`' event normally handled by `window.onunload`.
`idle`	Executed repeatedly (every 10 ms by default) when the system is idle (i.e., not busy running other scripts) after the page is loaded.
`resize`	Executed when the browser window is resized by the user. It corresponds to the browser '`resize`' event normally handled by `window.onresize`.
`showContextMenu`	Executed when the right mouse button is clicked. Canceling this event globally (by returning `false`) suppresses the default widget behavior of showing a context menu. See *"Mouse events" on page 53* for details.

For example, the following script (which appeared in *"Specifying initial position and size" on page 28* earlier) draws a widget after the page has been fully loaded:

```
Page.setEvent("load", "myWidget.draw()");
```

If you intend to remove a global event handler at some point, you must keep track of the event's ID. For example:

```
var myEventID = Page.setEvent("idle", myAction);
```

The event's ID can then be passed to the `Page.clearEvent` method later:

```
Page.clearEvent("idle", myEventID);
```

Alternatively, if you want to remove the handler after it executes just once, you can use the optional `Page.FIRE_ONCE` parameter in the call to the `Page.setEvent` method. For example, to execute `myAction` just once, on the next '`idle`' event:

```
Page.setEvent("idle", myAction, Page.FIRE_ONCE);
```

As mentioned above, global handlers can also be set for any of the mouse or drag-and-drop events. *"Example: Event propagation" on page 43* sets a global handler for the '`click`' event as follows:

```
Page.setEvent("click", "return confirm('Page received click event.
    Continue?')");
```

The *"Example: Getting event details" on page 55* demonstrates global handling of all possible events.

Warning

Always use the `Page.setEvent` method to assign document-level event handlers. When the ISC event-handling system is initialized, handlers previously assigned to many document-level events are replaced by the system's handlers. Specifically, the following handlers are overwritten if they have been set previously via scripting or `BODY` tag attributes:

- `window.onload`
- `window.onunload`
- `document.onresize`
- `window.onresize`
- `document.onscroll`
- `document.onmouseover`
- `document.onmouseout`
- `document.onclick`
- `document.ondblclick`
- `document.oncontextmenu`
- `document.onmousedown`
- `document.onmousemove`
- `document.onmouseup`
- `document.ondragstart`
- `window.ondragstart`
- `document.onkeydown`
- `document.onkeypress`
- `document.onkeyup`
- `document.onselectstart`
- `window.onselectstart`

Never set these event handlers directly, as doing so will disable the ISC system's processing of these events.

Registering keypress events

If you want a widget on the page to respond to a keypress event, the *Page* object can be used to register the key and designate an action to perform when the key is pressed. The *Page* object has two methods for registering keypress events, summarized in Table 3.3.

Table 3.3: Page methods for registering keypress events

Method	**Action**
`Page.registerKey(`*`key`*`, `*`action`*`, `*`target`*`)`	Registers when an alphanumeric or "special" key is pressed. An alphanumeric *`key`* parameter can be provided as a character or ASCII number. The *`key`* parameter for special keys (e.g. alt, meta, function, and arrow keys) should be provided as one of the strings given in Table 3.4. The *`target`* parameter is given as the object(s) to notify when the key is pressed. (More than one object can be notified.) The *`action`* parameter provides the script to be evaluated when the key is pressed. This *`action`* is typically a method called on the target object: `"target.`*`methodName`*`(`*`params`*`)"`. **Note:** To register a double quote, pass it contained in single quotes as the *`key`* parameter or escape the double quote with a backslash.
`Page.unregisterKey(`*`key`*`, `*`target`*`)`	Unregisters a registered *`key`* and stops notifying the specified *`target`*.

Example: Keypress Handling

The example file `keypress_handling.html` (shown in Figure 3.4) demonstrates how keys can be registered to invoke actions on a target widget using the `registerKey` method.

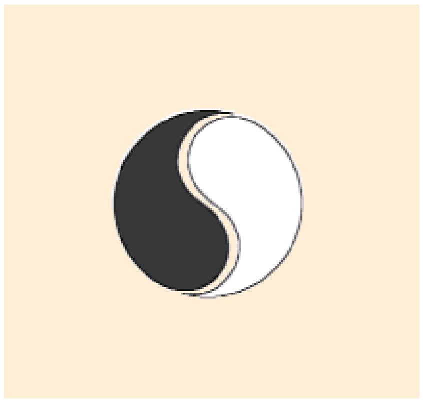


Figure 3.3: Example of registering and handling keypress events

In this example, four keys are registered to move the image 20 pixels at a time. Press on the "**i**" key to move the image up, the "**j**" key to move the image left, the **"k**" key to move the image down, and the "**l**" key to move the image to the right.

This is accomplished by registering these four keys and calling the `moveTo` method on the target object as follows.

```
Canvas.create({
   ID:"widget",
   left:200,
   top:200,
   width:100,
   height:100,
   backgroundImage:"yinyang_small.gif"
});
```

```
Page.registerKey("i", widget, "target.moveBy(0,-20)");
Page.registerKey("j", widget, "target.moveBy(-20,0)");
Page.registerKey("k", widget, "target.moveBy(0,20)");
Page.registerKey("l", widget, "target.moveBy(20,0)");
```

Special keys

Special keys refer to the non-alphanumeric keys such as the **Alt**, **Ctrl**, function, and arrow keys on a keyboard. To register these keys, you must call the `Page.registerKey` method and pass the appropriate name for the key given in Table 3.4.

Table 3.4: String mappings for registering special keys

Special Key Names (case-sensitive)		
"Tab"	"Arrow_Left"	"f1"
"Enter"	"Arrow_Up"	"f2"
"Shift"	"Arrow_Right"	"f3"
"Ctrl"	"Arrow_Down"	"f4"
"Alt"	"Insert"	"f5"
"Pause_Break"	"Delete"	"f6"
"Caps_Lock"	"Meta_Left"	"f7"
"Escape"	"Meta_Right"	"f8"
"Page_Up"	"Num_Lock"	"f9"
"Page_Down"	"Scroll_Lock"	"f10"
"End"		"f11"
"Home"		"f12"

Suppose the example above used the arrow keys on the keyboard to move the widget target object around instead. To do this, the `registerKey` method would be called as follows:

```
Page.registerKey("Arrow_Up", widget, "target.moveBy(0,-20)");
Page.registerKey("Arrow_Left", widget, "target.moveBy(-20,0)");
Page.registerKey("Arrow_Down", widget, "target.moveBy(0,20)");
Page.registerKey("Arrow_Right", widget, "target.moveBy(20,0)");
```

Handling widget events

An individual widget can respond to mouse or drag-and-drop events that target the widget directly, or that bubble up from a contained child widget, with its own event handlers. Widget event handlers can be set or cleared like other widget methods, either in the `create` method initialization block, or via standard dot notation.

For example, in this script from *"Creating widget instances" on page 9*, a button's click handler was set during initialization:

```
Button.create({
   ID:"myButton",
   left:100,
   top:100,
   height:20,
   title:"Say hello",
   click:"alert('Hello')"
});
```

Note

The name for a widget event handler is the same as the name of the actual event. The ISC event system does not use an "`on`"-prefixed event name (as in native JavaScript, e.g. '`onClick`') for event handlers.

A widget event handler can also be set after initialization. For example:

```
myButton.click = "alert('Goodbye')";
```

To clear a widget event handler, delete it:

```
delete myButton.click;
```

Like a global event handler, a widget event handler can be set either to a function, or to a string of script that will be automatically converted to a function. The best approach for a specific handler depends on a balance of coding flexibility, readability, and maintainability. Strings allow for deferred evaluation and dynamic construction of handlers (via concatenation), but may cause you some grief when your code contains quoted items (hence the alternation of double and single quotes in the examples above). Functions may be easier to read if your script editor highlights script syntax, but when set as handlers they only take parameters in a fixed order. The following four handlers differ in readability, flexibility, and maintainability, but all achieve the same result:

```
widget1.click = "alert('Hello')";
widget2.click = function(){alert('Hello')};
widget3.click = "sayHello()";
widget4.click = sayHello;

function sayHello() {
   alert('Hello');
}
```

The third handler above uses the best general approach for non-trivial handlers, setting the handler to a string of code that simply makes a function call. This allows reuse of the same function for multiple events, without restricting your ability to pass parameters to the function.

Keep in mind, however, that when you have a string to be evaluated that contains an internal string, any escape characters will be interpreted as part of the string itself—even if the escape character is contained within the internal string. For example, suppose you want a JavaScript alert to break to a new line.

```
myWidget.click = "alert('Are you sure?\nClick OK to continue.')";
```

The above statement will be interpreted as:

```
alert('Are you sure?
Click OK to continue.')
```

This results in an 'unterminated string constant' error. In cases like these, you need to escape the backslash for the interpreter to do what you intended.

```
myWidget.click = "alert('Are you sure?\\nClick OK to continue.')";
```

Warning

When you use a JavaScript statement to evaluate a string of script, the script must return a value as the result. Otherwise the ISC system will detect that there is no specified return value given in the script and will attempt to return the evaluated expression itself. In other words, suppose you use something like the following in your script:

```
myWidget.click = "if (testVariable == true)
  alert('true');"
```

This will fail because the ISC system will attempt to turn this string into a function by adding the '`return`' keyword at the beginning. The statement then becomes: "`return if (... `", which ultimately generates a syntax error. To avoid this pitfall, you should specify a return value wherever you will be using a JavaScript statement like the `if` statement given above. The following statement would succeed:

```
myWidget.click = "if (testVariable == true)
  alert('true'); return true"
```

You may also use the code below to achieve a valid return value depending upon what your function statement itself returns.

```
myWidget.click = "if (testVariable == true)
  return myFunction()"
```

Since handlers are actually methods of a widget instance, you can refer to a widget in any of its handlers with the `this` keyword. To refer to a different widget in a handler script string, you should call that widget's `getID` method to get its global identifier. See *"Referring to widget instances" on page 14* for details. For example:

```
hideMe.click = "this.hide()";
hideOther.click = widgetName.getID()+".hide()";
```

A widget handler that returns a `false` value will cancel further propagation of an event (i.e., bubbling of the event to the widget's parent). For example, this script sets a handler that stops the '`click`' event from bubbling beyond `myWidget`:

```
myWidget.click = "alert('The click stops here.'); return false"
```

Tip

It's a good habit to always cancel event propagation in your widget event handlers, unless you specifically want an event to continue bubbling after the handler executes. Most of the examples in this manual omit this step for brevity; however, canceling event propagation improves performance and avoids potential conflicts in parent elements that expect to receive the same event directly. Return `false` from your widget event handlers (except `mouseStillDown` and `dragStart`, for reasons described later) to stop a handled event.

You should also stop propagation of related events. For example, if a widget handles `dragStart` but not `dragStop`, it usually makes sense to stop `dragStop` here too. If you're implementing one handler on the widget, chances are it will be receiving related events, so just implement and stop the whole block of them. An event can be stopped even earlier, before any widget event handlers are executed, if the target widget is disabled. All widgets have an `enabled` property, with a default value of `true`, that can be initialized to `false` in the `create` method call. For example:

```
myButton = Button.create({ID:"myButton", title:"Say
  hello", click:"alert('Hello')", enabled:false});
```

Enabling and disabling widgets

The `enabled` property enables or disables a widget when it is initially instantiated. You can, however, also enable or disable widgets based on events. To do this, use the `enabled` and `disabled` methods described in Table 3.5. The `enabled` property should not be directly accessed after initialization.

Table 3.5: Widget enable and disable methods

Method	Action
`enable()`	Sets the widget's `enabled` property, and the `enabled` properties of any peers of the widget, to `true`.
`disable()`	Sets the widget's `enabled` property, and the `enabled` properties of any peers of the widget, to `false`.
`isEnabled()`	Returns `true` if the widget and all widgets above it in the containment hierarchy are enabled (i.e., `enabled==true`); returns `false` otherwise.

For example, the following button would be disabled after receiving a 'click' event:

```
Button.create({
   ID:"myButton",
   title:"Disable me",
   click:"this.disable()"
});
```

Note

A widget is only considered enabled if it is individually enabled and all parents above it in the containment hierarchy are enabled. This allows you to enable or disable all components of a complex nested widget by enabling or disabling the top-level parent only.

The visual appearance of some widgets (like the button in the example above) is affected by their enabled/disabled state. Refer to the chapter for each respective widget class for details on its behavior.

Default widget event handlers

Many of the predefined widget classes in the ISC system implement event handlers to provide their default functionality. Table 3.6 shows which classes implement which handlers by default.

Table 3.6: Predefined widget event handlers

Class(es)	Implements
Button	• `mouseDown` • `mouseUp` • `mouseOver` • `mouseOut`
Scrollbar	• `mouseDown` • `mouseStillDown` • `mouseUp` • `mouseOver` • `mouseOut`
ListGrid *Menu* *TreeGrid*	• `mouseDown` • `mouseUp` • `mouseMove` • `mouseOut` • `click` • `doubleClick` • `dragStart` • `dragMove` • `dragOut` • `dragUp`

Warning

You should never set these handlers for instances of these classes, as doing so will disable the predefined behavior of the affected widget.

Mouse events

The mouse events listed in Table 3.7 are generated by user interaction via the mouse.

Table 3.7: Mouse events

Event	**Description**
`mouseOver`	Executed when the mouse enters this object.
`mouseMove`	Executed when the mouse moves within this object.
`mouseOut`	Executed when the mouse leaves this object.
`mouseDown`	Executed when the left mouse button is pressed in this object.
`mouseStillDown`	Executed repeatedly (every 100 ms by default) when the system is idle—i.e. not busy running other scripts—and the left mouse button is held down after having been pressed in the object. This event is not native to JavaScript, but is provided by the ISC system.
`mouseUp`	Executed when the left mouse button is released in this object.
`showContextMenu`	Executed when the right mouse button is released in this object.If a context menu is defined for the widget, it will automatically show. See also *"Handling Page events" on page 44*.
`click`	Executed when the left mouse button is clicked (both pressed and released) in this object.
`doubleClick`	Executed when the left mouse button is clicked twice in rapid succession (within 250 ms, by default) in this object.

Following a `mouseDown` event, the `mouseOver`, `mouseMove`, `mouseOut`, `mouseUp`, `click` and `doubleClick` events are only sent to the `mouseDown` target and only if the mouse is over the `mouseDown` target.

The *"Example: Getting event details" on page 55* demonstrates both global and widget-level handling of all the mouse events listed above.

For performance reasons, the `mouseStillDown` event is sent *only* to widgets. All of the other mouse events, however, may be handled by both widget event handlers and global event handlers. Several of these events have cancellation side effects:

- If the `mouseDown` event is canceled by a native handler, widget handler, or page event handler, dragging (see *"Drag-and-drop operations" on page 56*) will not be initiated and `mouseStillDown` events will not be sent.
- If the `mouseStillDown` event is canceled by a widget event handler returning `false`, the repeated sending of this event will be terminated; `mouseStillDown` will not be sent again until triggered by another mouse-button press. To cancel bubbling of a single `mouseStillDown` event without canceling the repeated sending of the event, return `EventHandler.STOP_BUBBLING` or execute `EventHandler.stopBubbling()` from a widget's `mouseStillDown` handler.

The sequence of events generated by a double-click is `mouseDown, mouseUp, click, mouseUp, doubleClick`. If you need to handle both the `doubleClick` event and one or more of the `mouseDown`, `mouseUp`, `click`, and `mouseMove` events in the same widget, you should script your handlers with this event sequence in mind.

Note

Repeated double-clicking results in the following event sequence:
`click -> doubleClick -> click -> doubleClick -> click etc.`

In some situations, you may want to temporarily disable mouse events for all widgets, or for all widgets except a few that should be the focus of user interaction. For example, you may have a widget that displays a custom dialog box (with child widgets for buttons, etc.), and you want to accept mouse events *only in this widget* whenever it is displayed. Disabling (and later re-enabling) every other widget on the page could be a large, error-prone task; however, there's an easier way. SmartClient provides a "click mask" to capture all mouse events on the page. The click mask blocks all mouse events for everything on the page except a list of "unmasked" targets. When a `mouseDown` event occurs anywhere on the page outside of these unmasked targets, the user-specified "clickAction" fires. This action can, if necessary, cancel the `mouseDown` event.

Methods to show and hide the click mask are available on every Canvas instance, and are detailed in Table 3.8.

Table 3.8: Click-mask methods

Method	Action
`canvas.showClickMask ([`*`clickAction`*`], [`*`autoHide`*`], [unmaskedTargets])`	Shows the click mask. The *`clickAction`* parameter is an optional function or string of code that will be executed when a click occurs on the click mask. If the optional *`autoHide`* parameter is `true`, a click on the click mask will hide it (after executing any *`clickAction`*). Unmasked targets is a widget or Array of widgets that should not be masked
`canvas.hideClickMask()`	Hides the click mask.

Getting event details

When an event such as `mouseDown` occurs, the browser creates an event object that encapsulates properties of this event, such as the mouse location, which mouse button was pressed, etc. Unfortunately, browsers don't agree on the names for these event properties, making cross-browser event coding difficult.

In the ISC event system, these differences have been abstracted for you. When a ISC event handler is fired, it automatically extracts relevant properties of the event, and allows you to access these properties uniformly on both browsers. Instead of passing an event to your event handler, you call static methods on the *EventHandler* class that return properties of the last event seen by the system. These methods are described in Table 3.9.

Table 3.9: Event information methods

Method	**Action**
`EventHandler.getX()`	Returns the global X (horizontal) coordinate of the last event. This uses the same coordinate system as `widget.getPageLeft()`.
`EventHandler.getY()`	Returns the global Y (vertical) coordinate of the last event. This uses the same coordinate system as `widget.getPageTop()`.
`EventHandler.leftButtonDown()`	Returns `true` if the left mouse button (or only mouse button, on Macintosh) was pressed during the last event.
`EventHandler.rightButtonDown()`	Returns `true` if the right mouse button (or mouse button plus **Ctrl** key, on Macintosh) was pressed during the last event.
`EventHandler.shiftKeyDown()`	Returns `true` if the **Shift** key was pressed during the last event.
`EventHandler.ctrlKeyDown()`	Returns `true` if the **Ctrl** key was pressed during the last event.
`EventHandler.altKeyDown()`	Returns `true` if the **Alt** key was pressed during the last event.
`widget``.containsEvent()`	Returns `true` if the last event's coordinates were within the rectangle of widget (regardless of the widget's position in the stacking order).
`widget``.getOffsetX()`	Returns the value of the X coordinate where the event occurred relative to the widget's (0,0) drawn coordinate (top-left corner).
`widget``.getOffsetY()`	Returns the value of the Y coordinate where the event occurred relative to the widget's (0,0) drawn coordinate (top-left corner).

Example: Getting event details

The example file `event_details.html` (shown in Figure 3.4) demonstrates all of the event info methods above.

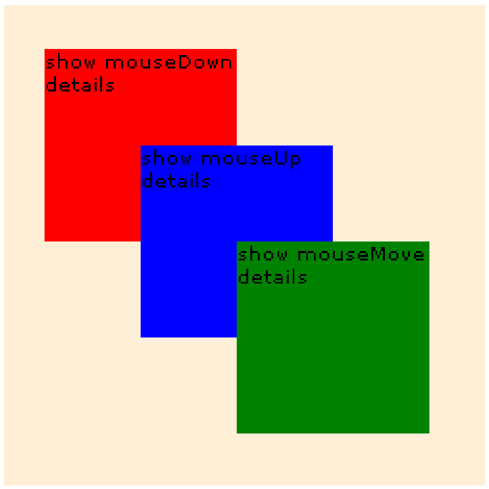


Figure 3.4: Example of getting event details

This example draws three simple widgets, containing handlers for `mouseDown` (red widget), `mouseUp` (blue widget), and `mouseMove` (green widget). Each of these handlers executes the following script:

```
   showEventInfo(this)
```

The `showEventInfo` function constructs a string describing the event, based on the return values of the event info methods, and displays this string in the window status area:

```
function showEventInfo(obj) {
   var result = "Global: " + EventHandler.getX() + "," + EventHandler.getY();
   result += "  Local: " + obj.getOffsetX() + "," + obj.getOffsetY();
   if (EventHandler.rightButtonDown()) result += "(Right Button)";
   if (EventHandler.leftButtonDown()) result += "(Left Button)";
   if (EventHandler.shiftKeyDown()) result += "(Shift)";
   if (EventHandler.ctrlKeyDown()) result += "(Ctrl)";
   if (EventHandler.altKeyDown()) result += "(Alt)";
   if (EventHandler.metaKeyDown()) result += "(Meta)";
   if (redWidget.containsEvent()) result += "(red widget contains event");
   if (blueWidget.containsEvent()) result += "(blue widget contains event");
   if (greenWidget.containsEvent()) result += "(green widget contains event");
   window.status = result;
}
```

As you click on the red or blue widgets, or move the mouse over the green widget, the window status area will update to indicate the details of the relevant event. Keep in mind that more than one widget can "contain" the event. The `containsEvent` method does not consider layering, nor whether the widget was the target of the event.

Drag-and-drop operations

The ISC event system provides an easy mechanism for implementation of drag-and-drop operations. All widgets have drag-and-drop properties that can be set during initialization. Table 3.10 lists these properties.

Table 3.10: Widget drag-and-drop properties

Property	Description	Default
`canDragReposition`	Indicates whether this widget can be moved by a user of your application by simply dragging with the mouse.	`false`
`canDragResize`	Indicates whether this widget can be resized by dragging on the edges and/or corners of the widget with the mouse.	`false`
`canDrag`	Indicates whether this widget can initiate custom drag-and-drop operations (other than reposition or resize).	`false`
`canDrop`	Indicates that this object can be dropped on top of other widgets. Only valid if `canDrag` or `canDragReposition` is `true`.	`false`
`canAcceptDrop`	Indicates that this object can receive dropped widgets (i.e other widgets can be dropped on top of it.)	`false`

Property	**Description**	**Default**
`dragAppearance`	Visual appearance to show when the object is being dragged, as described in Table 3.11.	`"outline"`
`dragTarget`	A different widget that should be actually dragged when dragging initiates on this widget. One example of this is to have a child widget that drags its parent, as with a drag box. Because the parent automatically repositions its children, setting the drag target of the child to the parent and then dragging the child will result in both widgets being moved.	`null`

Dragging and events

For all three dragging styles, dragging is initiated when the mouse goes down in a widget and then, before the mouse button is released, is moved five pixels away in any direction from its original location. If the mouse button goes down and up on a draggable widget without moving at least five pixels, no dragging operations will be generated. This means that you can script normal mouse events, for example, `mouseDown` or `click`, on a draggable widget, and the system will fire the appropriate messages based on whether the user actually starts dragging.

One thing to note about the ISC event model is that dragging events take precedence over normal mouse events. If we are in a dragging situation, normal mouse events such as `mouseMove` or `mouseUp` will be suppressed, in favor of specific messages such as `dragMove` or `dragStop`. This simplifies your event handling code, as you can code for normal and dragging mouse events completely separately. See *"Sequence of events in drag-and-drop operations" on page 69* for more information.

During drag operations, you can use the standard `EventHandler` event details methods (see *"Getting event details" on page 54*) to get the location of the mouse, whether the shift key is down, etc. You can use the property `EventHandler.dragTarget` (referred to as "the drag target" below) to get a pointer to the widget that is being dragged. The droppable object under the mouse (if there is one) can be accessed as `EventHandler.dropTarget` (referred to as "the drop target", below).

See the following sections *"Drag repositioning," "Drag resizing,"* and *"Custom drag-and-drop operations"* for details on the events actually generated in each case.

Drag appearance

The ISC system provides default behavior for showing where the mouse is during a drag-and-drop operation through an object's `dragAppearance` value. Values for `dragAppearance` are listed in Table 3.11.

Table 3.11: The dragAppearance property values

Value	Description
`"tracker"`	A "drag tracker" object is automatically shown and moved around with the mouse. This is generally set to an icon that represents what is being dragged. The default tracker is a 10-pixel black square, but you can customize this icon; see *"Setting the drag tracker" on page 58* for details. This `dragAppearance` is not recommended for use with drag resizing or drag moving.
`"outline"`	An outline the size of the target object is moved, resized, etc. with the mouse. This is recommended for drag resizing, especially for objects that take a significant amount of time to draw.
`"target"`	The target object is actually moved, resized, etc. in real time. This is recommended for drag repositioning, but not for drag resizing of complex objects.
`"none"`	No default drag appearance is indicated. Your custom dragging routines should implement some behavior that indicates that the user is in a dragging situation, and where the mouse is.

Event sequences for various drag-and-drop operations with different `dragAppearance` settings are provided in *"Sequence of events in drag-and-drop operations" on page 69.*

Setting the drag tracker

If your drag operation has a `dragAppearance` of `tracker`, your widget can implement a `setDragTracker` method to customize the appearance of the drag tracker. The drag tracker will generally be an image, but you can use any legal HTML string for your tracker. The tracker will expand to show the entire HTML you define for your tracker image.

Call the `EventHandler.setDragTracker` method to set the tracker:

```
EventHandler.setDragTracker(contents, [width], [height], [offsetX],
   [offsetY]);
```

For example, to set the drag tracker to a 20-by-20 pixel image, do the following:

```
myWidget.setDragTracker = function () {
   EventHandler.setDragTracker(this.imgHTML("yinyang.gif",20,20));
}
```

You do not have to explicitly set the width and height of the tracker in the `EventHandler.setDragTracker` method. It will default to a size of 20-by-20 pixels because this is the size you set for the image. If you're setting the tracker to a block of text or other HTML, you may want to set an explicit size, like so:

```
myWidget.setDragTracker = function () {
   var contents = "Some text for the <B>drag tracker</B>";
   EventHandler.setDragTracker(contents, 50, 10));
}
```

This will create the text in a 50-pixel wide block; the height will be at least 10 pixels, but may be taller if required to show all of the specified text.

Both `offsetX` and `offsetY` are optional parameters that can be set to specify how far to offset the drag tracker from the mouse pointer. If not specified, a default of -10 is used for each, so the tracker appears 10 pixels below and 10 pixels to the right of the mouse pointer.

Drag repositioning

One of the most common dragging operations performed in applications is simply allowing the user to move a widget around with the mouse. In the ISC event system, to set this interaction up you only have to set `widget.canDragReposition` to `true`. A widget with this property set will automatically display a move cursor when the mouse passes over it and, when clicked and dragged, will be repositioned with the mouse until the mouse button is released. The appearance as the object is moved with the mouse is controlled by the `dragAppearance` property, detailed previously.

Drag repositioning is compatible with drag resizing, allowing you to have a widget that can be both moved and resized. Drag repositioning and custom drag-and-drop operations are not compatible, though. If an object has both `canDragReposition == true` and `canDrag == true`, drag repositioning will take precedence and the custom dragging code will be ignored.

Note

As the cursor is passed over a widget that is drag-repositionable, the cursor will automatically change to the move cursor shown below.

✥

To turn this default behavior off, set the `canvas.dragRepositionCursor` to `null`, or set it to another cursor as desired. See `cursor` in Table 2.13 on page 39 for a list of available cursors.

As the widget is dragged around with the mouse, the drag events listed in Table 3.12 fire.

Table 3.12: Widget drag events

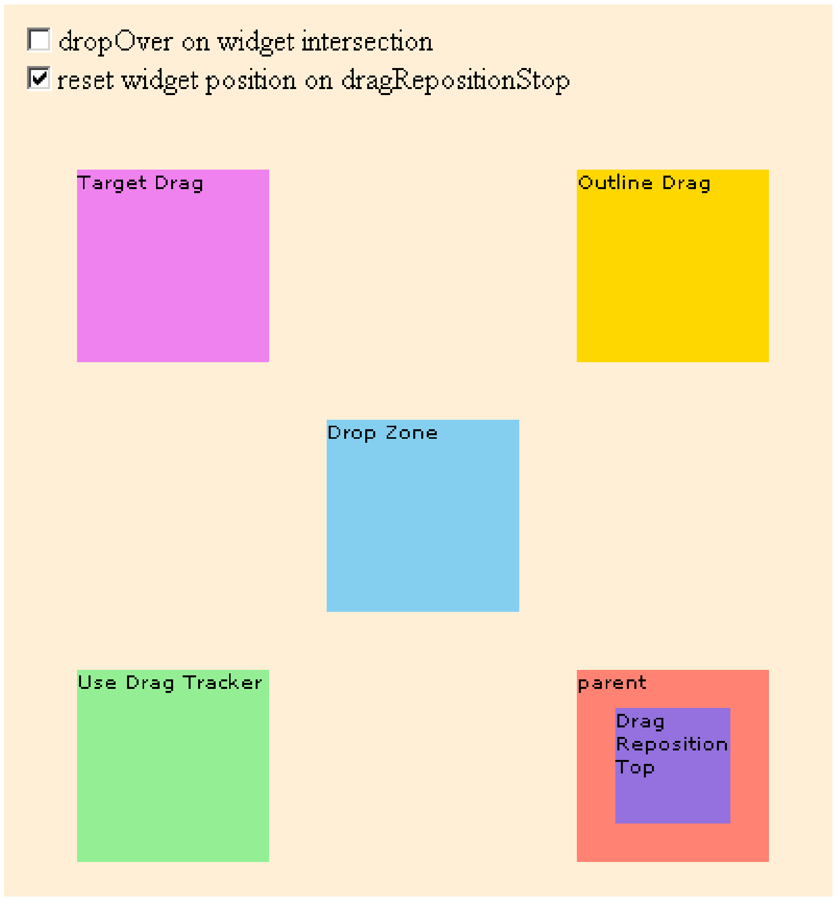
Event	**Description**
`dragRepositionStart`	Executed when dragging first starts. You can create a handler for this event to set things up for the drag reposition. Returning `false` from this event handler will cancel the drag reposition action.
`dragRepositionMove`	Executed every time the mouse moves while drag-repositioning. If your `dragRepositionMove` handler does not return `false`, the widget or outline will automatically be moved as appropriate whenever the mouse moves.
`dragRepositionStop`	Executed when the mouse button is released at the end of the drag. Your widget can use this opportunity to fire custom code based upon where the mouse button was released, etc. Returning `true` from this handler will cause the drag target to be left in its (or the outline's) current location. Returning `false` from this handler will cause the object to snap back to its original location.

Note

You do not need to implement any special behavior in any of these events for a normal drag reposition operation.

Example: Drag appearance

The example file `widget_drag_appearance.html` (shown in Figure 3.5) shows the different appearances that can be set for draggable widgets when dragged.



☐ dropOver on widget intersection

☑ reset widget position on dragRepositionStop

Figure 3.5: Example of drag appearance

Each draggable widget demonstrates a different `dragAppearance` setting.

All four draggable widgets can be dropped on the drop zone (blue square). Selecting the **dropOver on widget intersection** checkbox causes the drop event behavior—the display of a dialog box—to commence upon intersection of the dragged widget with the drop zone. Otherwise, the drop event only occurs once the mouse is over the drop zone while a widget is being dragged. See *"Drop operations" on page 66* for more information. All widgets have `canDragReposition` set to `true`.

- The top-left (purple) widget has `dragAppearance` set to `"target"`. This causes the widget itself to move when dragged.
- The top-right (gold) widget has `dragAppearance` set to `"outline"`. When dragged, an outline of the widget moves with the mouse.
- The bottom-left (green) widget has `dragAppearance` set to `"tracker"`. When dragged, a tracker appears that moves with the mouse to indicate dragging. The tracker is set with the following line of code:

```
setDragTracker:"EventHandler.setDragTracker(this.imgHTML('yinyang_icon.gif',
    20, 20))"
```

- The bottom-right widgets are parent (orange) and child (purple). The parent widget has a `dragAppearance` set to `"outline"`, and `canDrop` set to `true`. However, no `canDrag`, `canDragReposition`, or `canDragResize` properties are set. The child widget, on the other hand, has `canDragReposition` set to `true`, and sets its parent as a `dragTarget`. Attempting to drag the parent widget alone has no effect. Dragging the child widget, however, drags the parent widget. The parent widget's `dragAppearance`, drag event handlers, and other drag-and-drop properties are used in the drag operation. When the drag sequence ends, the parent is redrawn at the new location, and the child is moved with the parent.

Drag resizing

Another common drag-and-drop operation is allowing the user to resize a widget with the mouse. The ISC event system allows you to accomplish this by setting a single widget property, `canDragResize` to `true`. When the user presses and holds the left mouse button in the side and/or corner of a drag resizable widget, the widget will automatically grow or shrink as they move their mouse, until the mouse button is released. Sides and edges are by default defined as the outer five pixels of a widget.

The default implementation is that a drag-resizable widget can be dragged from any corner or edge, and that the edge or corner clicked upon will dictate how the object resizes. For example, a drag resize beginning in the top edge of the widget will resize the widget from the top, increasing its height. A drag-resize from the bottom right corner will resize both the bottom and right sides of the widget simultaneously.

If you want to only allow resizing in certain edges or corners, you can do this by setting the `widget.resizeFrom` property. The default value of `null` indicates that the widget is resizable from any corner or edge. To restrict resizing to only certain corners, set `resizeFrom` to an array of any of the values listed in Table 3.13.

Table 3.13: The `resizeFrom` property values

Value	Description
`T`	Top edge
`B`	Bottom edge
`L`	Left edge
`R`	Right edge
`TL`	Top-left corner
`TR`	Top-right corner
`BL`	Bottom-left corner
`BR`	Bottom-right corner

Note Values in the `resizeFrom` array may be set in any order.

So, for example, to make a widget resizable from only the bottom and right sides and intersecting corner, set the following:

```
Canvas.create({
    ID:"myWidget",
    . . .
    canDragResize:true,
    resizeFrom:["B","BR","R"],
    . . .
});
```

As the user passes the mouse over a widget that is drag-resizable, the cursor will change to an arrow that indicates the direction that the resize will happen. To turn this feature off, set the `canvas.edgeCursorMap` property to `null`. This cursor change will *only* appear in edges or corners of the object that the widget can be resized from.

You can also constrain the width and/or height of a drag-resizable object by setting the properties described in Table 3.14.

Table 3.14: Minimum and maximum height and width properties

Property	Description	Default
`minWidth`	Minimum width, in pixels, for drag resizing.	`10`
`maxWidth`	Maximum width, in pixels, for drag resizing.	`10000`
`minHeight`	Minimum height, in pixels, for drag resizing.	`10`
`maxHeight`	Maximum height, in pixels, for drag resizing.	`10000`

If the current mouse position during a drag resize would make the widget larger than its stated maximums, or smaller than its stated minimums, the size of the widget will be constrained as appropriate.

As the widget is dragged around with the mouse, the `dragResize` events detailed in Table 3.15 are generated.

Table 3.15: the `dragResize` property events

Event Generated	Description
`dragResizeStart`	Executed when drag-resizing first starts. You can set an event handler to set things up for the drag reposition. Returning `false` from this event handler will cancel the drag resize action entirely.
`dragResizeMove`	Executed every time the mouse moves while drag-resizing. If your `dragRepositionMove` handler does not return `false`, the widget or outline will automatically be resized as appropriate whenever the mouse moves.
`dragResizeStop`	Executed when the mouse button is released at the end of the drag. Your widget can use this opportunity to fire custom code based upon where the mouse button was released, etc. Returning `true` from this handler will cause the drag target to remain at its (or the outline's) current size. Returning `false` from this handler will cause the object to "snap back" to its original size.

Note

A widget that is currently being drag-resized will not participate in '`drop`' events, even if its `canDrop` property is `true`. To drop a widget that is drag-resizable and either drag-repositionable or has custom drag behavior, the user must drag from the center of the widget, not from the edge.

Drag resizing from a sizer button

A widget can be resized from a child widget. The child widget must have `canDragResize` set to `true`, and must specify the parent as the `dragTarget`. The child also should set a `getEventEdge` property to a function that returns which corner or side the child should be considered as resizing. For example, a child widget in the bottom right corner of the parent it resizes should have the following property:

```
getEventEdge:function(){return "BR"}
```

This causes the entire child widget to act as a bottom-right corner of a resizable widget normally would, dictating the cursor appearance and resizing directions allowed.

The parent widget should include an implementation of `layoutChildren()` that places the resizer widget at the parent's lower right corner. Otherwise, the resizer widget will not be moved or resized with the parent.

Example: Drag resizing

The example file `widget_drag_resize.html` (shown in Figure 3.6) demonstrates different drag resizing options.

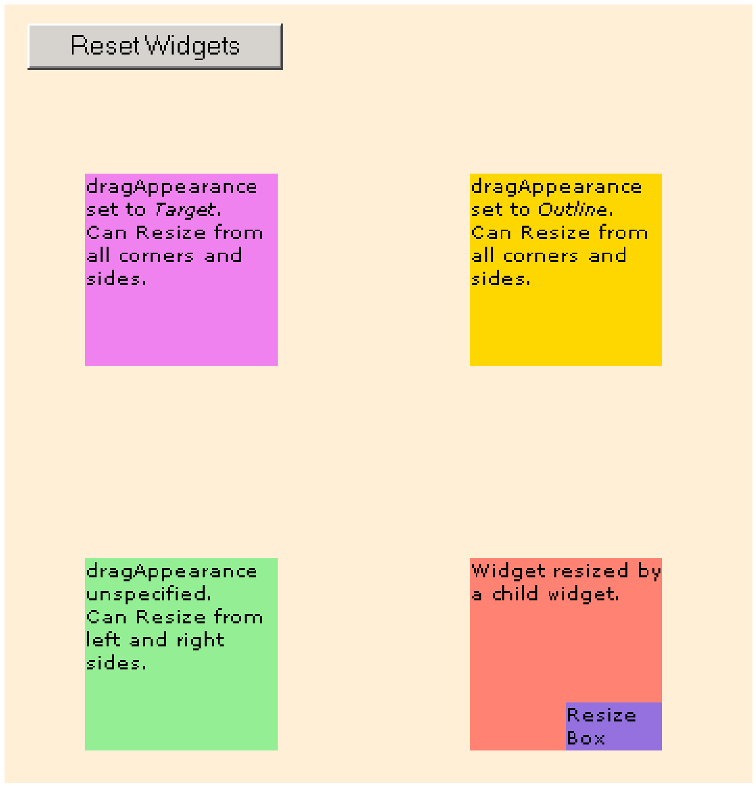

Reset Widgets
dragAppearance set to *Target*. Can Resize from all corners and sides.
dragAppearance set to *Outline*. Can Resize from all corners and sides.
dragAppearance unspecified. Can Resize from left and right sides.
Widget resized by a child widget.
Resize Box

Figure 3.6: Example of drag resizing

- The top-left widget sets `dragAppearance` to `"target"`. Therefore, as the widget is resized, the widget itself is shown changing size. This widget does not specify corners and sides for its `resizeFrom` property. Therefore, the widget is resizable from all corners and sides.
- The top-right widget also does not specify corners and sides for `resizeFrom`. The `dragAppearance` of this widget is set to `outline`. As this widget is resized, an outline of the widget shows the size change. When the mouse is released, the widget is drawn in the space occupied by the outline.
- The bottom-left widget does not specify a `dragAppearance`. By default, `outline` is used. The `resizeFrom` property of this widget is set to `["L","R"]`, allowing the widget to only be resized from the left and right sides.

- The bottom-right widget is resized by a child widget. The parent sets the `dragAppearance`, which in this case is `outline`. The child sets `canDragResize` to `true` and `dragTarget` to be the parent widget. The parent must also set a property `resizeChildrenBy` to a function. This function is evaluated to determine the location and size of child widgets upon a resize event. This function usually determines the size and position of the child widgets relative to the parent widget's new size. In this case, the child widget is moved to the same corner and kept the same size. Therefore, it uses the `moveTo` method to set the left and top coordinates of the child widget relative to the width and height of the parent widget, as follows:

```
resizeChildrenBy:function() {
   var resizeBox = this.children[0];
   this.children[0].moveTo(this.getWidth()-resizeBox.getVisibleWidth(),
      this.getHeight()-resizeBox.getVisibleHeight())
}
```

See *"Controlling position and size" on page 30* for positioning and sizing properties and methods.

The child widget uses a function to specify a value for `getEventEdge`:

```
getEventEdge:function(){return "BR"}
```

This sets the entire child widget to have the `resize` properties associated with a bottom-right corner. The cursor over the entire child widget changes, and the resizing of the parent occurs as if the bottom right corner of the parent were resizable.

Custom drag-and-drop operations

If you want to perform a drag-and-drop operation other than simply moving or resizing a widget, you must code your own custom drag-and-drop logic. This is fairly easy, and mainly involves writing handlers for the drag events generated, described in Table 3.16.

Table 3.16: Drag events

Event Generated	Description
`dragStart`	Executed when dragging first starts. Your widget can use this opportunity to set things up for the drag, such as setting the drag tracker (see *"Setting the drag tracker" on page 58*), or showing an outline around your widget to indicate that it is being dragged (see *"Drag appearance" on page 57*). Returning `false` from this event handler will cancel the drag action entirely.
`dragMove`	Executed every time the mouse moves while dragging. Returning `false` from this event handler will cancel the drag action entirely.
`dragStop`	Executed when the mouse button is released at the end of the drag. Your widget can use this opportunity to fire code based on the last location of the drag or reset any visual state that was sent. Returning `false` from this event handler will cancel the drag action entirely.

Using the drag tracker with a custom drag

A common pattern is to use the drag tracker with custom drag-and-drop interactions, to indicate what is being dragged. To do this, set the following widget properties:

```
Canvas.create({
    ID:"myWidget",
    . . .
    canDrag:true,
    dragAppearance:"tracker",
    . . .
});
```

Then set the drag tracker in your `dragStart` handler, as detailed in *"Setting the drag tracker" on page 58*.

Note

You do not need to write logic on `dragMove` to move the drag tracker; it will be done for you automatically as a result of choosing the `"tracker" dragAppearance`.

Drop operations

Drop operations are what your operating system does when you move files onto folders in the **Explorer** (Windows) or **Finder** (Macintosh) window. We define "dropping" to be the movement and subsequent release of a 'draggable' widget over a drop target. What your application does as the result of a drop requires custom code, but the mechanics of figuring out where the drop occurred are handled for you by the ISC event system.

There are two parts to every successful drop operation, the draggable widget (where the event begins) and the droppable widget (the widget being dropped on to). Use the drop properties detailed in Table 3.17 to set up these widgets.

Table 3.17: Drop properties

Property	Set to...	Description
`canDrop`	`true`	**Set on:** draggable widget Indicates that this widget can be dropped on top of other widgets. You must either have `canDrag:true` or `canDragReposition:true` to make this widget start the drag-and-drop sequence. You can have a draggable widget that is not droppable, such as the thumb of a scrollbar. You can also have widgets that are `canDragReposition:true` and `canDrop:true`, such as a file on the desktop of your OS file management interface. These files can be repositioned around the screen, and can also be dropped on a folder to move them into that folder. **Note:** The same object may be `canDrag:true` and `canDrop:true`. An example of this usage is a list of folders and files where you can drag from one part of the list into another part.
`dragIntersectStyle`	`"mouse"` or `"rect"`	**Set on:** draggable widget Indicates how the system will test for droppable targets. The `mouse` value indicates to look for drop targets that are under the current mouse cursor position.The `rect` value indicates to look for drop targets by intersection of the entire rect of the drag target with the droppable target.
`canAcceptDrop`	`true`	**Set on:** droppable widget Indicates that this widget can accept other objects dropped upon them. Only widgets with `canAcceptDrop:true` will be considered by the ISC event system as legal drop candidates.

In addition to the events sent to the draggable (or drag-repositionable) object mentioned above, the drop events described in Table 3.18 will be executed to any compatible droppable object under the mouse, if one exists.

Table 3.18: Drop events

Event	Description
`dropOver`	Executed when the compatible dragged object is first moved over this drop target. Your implementation can use this to show a custom visual indication that the object can be dropped here.
`dropMove`	Executed whenever the compatible dragged object is moved over this drop target. You can use this to show a custom visual indication of where the drop would occur within the widget.

Event	Description
`dropOut`	Executed when the dragged object is moved out of the rectangle of this drop target. If you have set a visual indication in `dropOver` or `dropMove`, you should reset it to its normal state in `dropOut`.
`drop`	Executed when the mouse button is released over a compatible drop target at the end of a drag sequence. Your widget should implement whatever it wants to do when receiving a drop here. For example, in a file moving interface, a drop might mean that you should move or copy the dragged file into the folder it was dropped on, or dropping something in a trash can might mean to clear it from the screen.

Checking for drop compatibility

The default behavior of the system is that anything draggable can be dropped on anything else that is droppable. However, in your application, you may have multiple draggable things on your page of different, incompatible types, or in a particular widget you may have some parts of the widget which can be dropped on and some others which cannot accept drops. For example, in a `treeGrid`, you can drop a file on a folder, but not on another file.

There are two methods for restricting what can be dropped within your application:

- `dragType`/`dropTypes`, and
- the `willAcceptDrop` method.

DragType and DropTypes

You may have different containers that will be dragged from, and other containers that will be dropped onto. Some of the drop containers may be able to accept anything that is dropped on them (think of a trash can), while other drop containers may only accept certain things to be dropped on them. This can be easily accomplished with the `dragType` and `dropTypes` properties described in Table 3.19.

Table 3.19: The dragType and dropTypes properties

Property	Set to...	Set on...	Description
`dragType`	`string`	draggable widget	The "type" of thing that can be dragged from this widget. If specified, this will be matched up with the `dropTypes` of droppable widgets as detailed below.
`dropTypes`	`string` or `string[]`	droppable widget	The "type" of thing(s) that can be dropped on this widget. Can be a string or an array of strings (indicating multiple types). Leave this with the value `null` to indicate that this widget can accept anything dropped on it from the page.

Set the `dragType` of your draggable widget to a string that represents what kind of object can be dragged from the widget. For example, in an e-mail application, the list of messages may have a `dragType` of "message". The choice of "message" is completely arbitrary, and should be something that makes sense in the context of your application.

To restrict what can be dropped in a widget, set its `dropTypes` to the string or array of strings that represent the `dragType` that you want the widget to accept. So a list of e-mail folders in our e-mail application might be set as follows:

```
. . .
dropTypes : ["message", "folder"],
. . .
```

This indicates that both "messages" and "folders" can be dropped onto this widget.

When the ISC event system is determining which element should receive a drop event during a drag operation, it will automatically compare the `dragType` of the drag target and the `dropTypes` of potential drop targets to see if they match. If they are not compatible (i.e. the `dropTypes` does not contain the `dragType`), the dragged object would not be a candidate for a drop, regardless of its `canDrop` setting.

The willAcceptDrop method

Sometimes you need finer-grained control of whether something can be dropped in a particular part of your droppable object. For example, think of two lists of folders and files, where items can be moved back and forth between the lists through dragging and dropping. You can drag a file from one list onto a folder in the other list, but it cannot be dropped on a file on the other list.

For finer-grained control of droppability, implement a `willAcceptDrop` method in your droppable canvas. This can perform any custom check you like, and should return `true` if a drop at this location is acceptable, or `false` if not. In the file movement example above, as the mouse is moved a method would determine which item the mouse is over, returning `true` if over a folder and `false` if over a file.

The default `dragType` and `dropTypes` implementation is done via a `willAcceptDrop` handler implemented in the base *Canvas* class. If you override the `willAcceptDrop` handler in your custom class or object, you can use the default `dragType` and `dropTypes` behavior as an initial by calling the superclass implementation:

```
myCustomCanvas.addMethods({
    . . .
    willAcceptDrop : function () {
        if (!this.Super("willAcceptDrop", arguments)) return false;
        // custom drop test code here
        . . .
    },
    . . .
});
```

Sequence of events in drag-and-drop operations

Table 3.20, Table 3.21, and Table 3.22 show the sequence of various drag-and-drop events for widgets with differing `dragAppearance` settings.

These sequences omit the `dragOut` event that would be sent to a drop target if the user moves the mouse off that widget while the mouse is still down. A `dropOut` message is sent to the drop target if the mouse is released over it. This is so the logic that sets the visual appearance of the drop target on `dropOver` will get a `dropOut` message to reset the appearance.

The usual mouse events (`mouseOver`, `mouseMove`, `mouseOut`, `mouseUp`, `click`, `doubleClick`) are not sent during a drag-and-drop interaction. The `mouseStillDown` event will be sent if a widget has both `canDrag` or `canDragReposition` set to `true` and a `mouseStillDown` handler. Also, keep in mind that the `mouseOver` and `mouseOut` events at the end of a drag-and-drop interaction are not necessarily sent to the drop target. They could be sent to another widget that did not receive any drag events because its `canAcceptDrop` property is `false`.

Drag repositioning with target dragAppearance

The events in Table 3.20 are both repositionable and droppable with `dragAppearance` set to `"target"`.

Table 3.20: Event sequence—Drag repositioning of a widget

Action	Event --> Sent to
1: User presses left mouse button in `canDragReposition` target.	`mouseDown --> dragTarget`
2: User moves mouse within the drag target.	`mouseMove --> dragTarget`

3: User keeps moving mouse more than 5 pixels since `mouseDown`.	`mouseOut --> dragTarget`
	`dragRepositionStart --> dragTarget`

4: User moves mouse within the browser window.	`dragRepositionMove --> dragTarget` Drag target is automatically moved to follow the mouse.

5: User moves mouse onto a compatible drop target.	`dragRepositionMove -->` `dragTarget`
	Drag target is automatically moved.
	`dropOver --> dropTarget`

6: User moves mouse within the drop target.	`dragRepositionMove --> dragTarget`
	Drag target is automatically moved.
	`dropMove --> dropTarget`

7: User releases mouse button.	`dropOut --> dropTarget`
	`drop --> dropTarget`
	`dragRepositionStop --> dragTarget`
	`mouseOver -->` `Object` under the mouse

Drag resizing with outline dragAppearance

Table 3.21: Event sequence—Drag resizing of a widget

Action	**Event --> Sent to**
1: User presses left mouse button in corner or side (set in `resizeFrom` array) of `canDragResize` target.	`mouseDown --> dragTarget`
2: User moves mouse within the drag target.	`mouseMove --> dragTarget`

3: User keeps moving mouse more than 5 pixels since `mouseDown`.	`mouseOut --> dragTarget`
	`dragResizeStart --> dragTarget`

4: User moves mouse while button is still down.	`dragResizeMove --> dragTarget` Outline is automatically resized.
5: User releases mouse button.	`dragResizeStop --> dragTarget` `mouseOver --> Object` under the mouse

Custom dragging with tracker dragAppearance

Table 3.22: Event sequence—Custom dragging of a `canDrag` and `canDrop` widget

Action	**Event --> Sent to**
1: User presses left mouse button in `canDragReposition` target.	`mouseDown --> dragTarget`
2: User moves mouse within the drag target.	`mouseMove --> dragTarget`

3: User moves mouse more than 5 pixels since `mouseDown`.	`mouseOut --> dragTarget`
	`dragStart --> dragTarget`
	`setDragTracker --> dragTarget`

4: User moves mouse off drag target or elsewhere on the screen.	`dragMove --> dragTarget` Tracker is automatically moved to follow the mouse.

5: User moves mouse onto a compatible drop target.	`dragMove --> dragTarget`
	Tracker is automatically moved.
	`dropOver --> dropTarget`

6: User moves mouse within the drop target.	`dragMove --> dragTarget`
	Tracker is automatically moved.
	`dropMove --> dropTarget`

7: User releases mouse button.	`dropOut --> dropTarget`
	`drop --> dropTarget`
	`dragStop --> dragTarget`
	`mouseOver --> Object` under the mouse

CHAPTER 4

Images and Skins

The Isomorphic SmartClient framework provides a variety of ways to place images within an application's widgets and customize the appearance of the widgets themselves within an application. For convenience and portability, the ISC system also provides some "special directories" that can be used in specifying relative paths to images you use.

This chapter details how to work with images within the ISC framework to achieve the desired application appearance and explains how files should be organized within your applications.

In this chapter:

Placing images in an application

The ISC system provides several ways to place your own images in an application to achieve the desired application operation and appearance.

- *Specifying images on a page*: You can specify an image directory to load your images and styles for an entire page.
- *Specifying images within a Canvas widget*: You can embed images in the contents of a widget by calling a method to generate the HTML for the image. This method can be called for any widget because it is inherited from the *Canvas* widget superclass. For more information, see*"Widget contents" on page 20*. If you want to customize your application by setting a `backgroundImage` for a *Canvas* widget, see *"Other visual properties" on page 39* or peruse the `widget_position_size.html` example for details.
- *Using the Img and StretchImg widgets*: The *Img* and *StretchImg* widgets can be used to display a single image or a list of multiple images. The image to display on a page is set using the properties of these simple widgets. See *"Img widgets,"* and *"StretchImg widgets"* later in this chapter for more information. Widgets that use images in their display will typically allow you to set properties to pick up custom images for them.

  The ISC system also allows you to customize the appearance of widgets themselves.
- *Using skins to specify widget images and styles*: The Isomorphic SmartClient framework includes a set of standard "skins" to choose a familiar "look and feel" or use as a basis for creating your own custom appearance. Each skin includes both images and a cascading style sheet (CSS) to achieve a familiar and consistent view.

ISC "special directories"

The ISC framework is organized with respect to the `isomorphic` directory where you installed ISC, and your applications directory where your ISC applications will be stored. You should keep your application code and customizations separate from the ISC files to make upgrading easy without running the risk of clobbering any of your own work.

The ISC system will assume images and styles are located in specific places relative to your application files unless you specify alternate locations for them. Four "special directories" are available for your use in specifying relative paths to images in your environment.

- **[ISOMORPHIC] Special Directory:** The `[ISOMORPHIC]` "special directory" refers to the `isomorphic` directory at the top-level of the SDK package.

- **[ISOMORPHIC_CLIENT] Special Directory:** The `[ISOMORPHIC_CLIENT]` "special directory" specifies the location of the ISC libraries, in other words, the directory where the `Isomorphic_SmartClient.js` file lives. The `[ISOMORPHIC_CLIENT]` "special directory" variable is resolved to `[ISOMORPHIC]/system/`.

  The `[ISOMORPHIC_CLIENT]` "special directory" is useful for specifying relative paths to the ISC libraries and system files used by your pages. The files in this directory may be updated as new versions of Isomorphic SmartClient become available. You should not put any of your own files or images in this directory.

- **[APP] Special Directory:** The location of your application files is given by the `[APP]` "special directory". This is assumed to be one level above the bootstrap file that launches the application, typically in a directory under webroot named after your application. Putting your application files outside of the `isomorphic` directory makes upgrades easier because it can then be replaced when a newer version becomes available.

  By default, the page will assume that the application's image files are located in `[APP]/images/`. To change the page's application image directory property, use the `setAppImgDir` method. For example, if you want to specify that all your images are in the `myImages directory` for a page in your application, call `Page.setAppImgDir("[APP]/myImages/")`. See *"Specifying image directories,"* Table 4.1 on page 79 for details.

- **[SKIN] Special Directory:** The location of images and styles for use with widgets is given by the `[SKIN]` "special directory". This directory is used to specify which images should be loaded for use with standard widgets. See *"Using and customizing ISC skins" on page 75* for more information.

Warning

If your `[APP]` "special directory" is located somewhere other than `../` relative to the `isomorphic` directory, you must specify this relative location to ensure that the ISC libraries and styles can be located. This is done using the `Page.setIsomorphicDir(`*`path`*`)` method near the top of the page in your application bootstrap file. See "*Specifying image directories*" *on page 78* for more information.

Using and customizing ISC skins

A ISC "skin" comprises a set of images and styles to change the appearance of widgets without changing their underlying functionality. The ISC framework ships with standard skins you can use to create a familiar user interface style or you can create your own custom skin with your own images and styles.

Skin directory structure

Each ISC skin has the directory structure given in Figure 4.1.

`[SKIN]`	The skin directory to use, (relative to the `[ISOMORPHIC]` or `[APP]` directory).
`skinName/`	Directory for a particular skin
`load_skin.js`	JavaScript file used for loading the skin within a page
`skin_styles.css`	Cascading style sheet to define styles for this skin
`unsupported_browser.html`	File to load if this browser is not supported
`images/`	Directory containing images used by this skin
`blank.gif`	1-pixel square white image used for spacing
`black.gif`	1-pixel square black image used for separators
`grid.gif`	An image that can be tiled for creating a grid background
`common/`	Directory containing images used by various widgets
`Dialog/`	Directory containing images used by the *Dialog* widget
`ListGrid/`	Directory containing images used by the *ListGrid* widget
`Menu/`	Directory containing images used by the *Menu* widget
`Progressbar/`	Directory containing images used by the *Progressbar* widget
`Scrollbar/`	Directory containing images used by the *Scrollbar* widget
`TreeGrid/`	Directory containing images used by the *TreeGrid* widget

Figure 4.1: ISC skin directory structure

This structure allows all system-defined widget images to be cleanly separated out so that you can easily load them within an application as necessary, thereby changing the "look and feel" of an application's widgets from a single location.

Using alternate skins included with the ISC framework

To change Isomorphic SmartClient skins place the skin to load between `<SCRIPT>` tags at the top of your ISC page. You must load the `Isomorphic_SmartClient.js` file first, and then load the skin to use with the page. The `standard` skin is loaded in all the widget samples to give them a Microsoft Windows appearance.

```
<HTML>
<HEAD>
<SCRIPT SRC=../../isomorphic/system/Isomorphic_SmartClient.js></SCRIPT>
<SCRIPT SRC=../../isomorphic/skins/standard/load_skin.js></SCRIPT>
```

Note

In all the widget samples, the Isomorphic SmartClient libraries and `standard` skin are loaded relative to the `examples` directory. These paths will vary based on the location of your own applications.

If, instead, you wanted to use the ISC skin with your widgets, you'd simply load it by referencing the `load_skin.js` script within the `SmartClient` skin directory instead of the `standard` skin directory at the top of the page. In this case, you would load the Isomorphic SmartClient libraries and skins relative to your applications directory.

```
<HTML>
<HEAD>
<SCRIPT SRC=../isomorphic/system/Isomorphic_SmartClient.js></SCRIPT>
<SCRIPT SRC=../isomorphic/skins/SmartClient/load_skin.js></SCRIPT>
```

Example: ISC skin

The example file `skin_custom.html` (shown in Figure 4.2) uses the `SmartClient` skin instead of the `standard` skin.

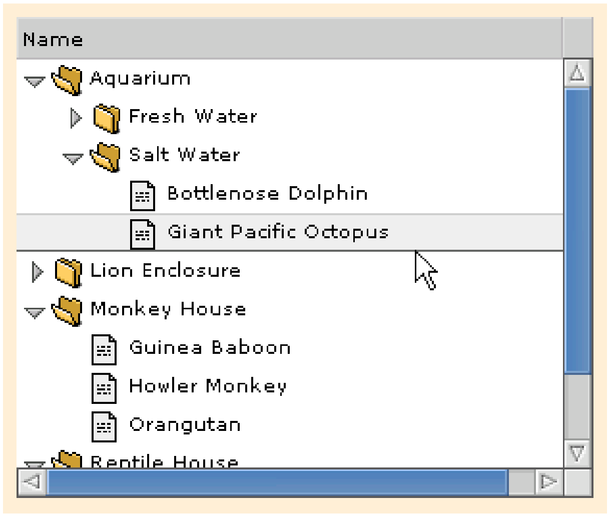


Figure 4.2: Example of the SmartClient skin

Creating your own custom skins

If none of the skins that shipped with Isomorphic SmartClient adequately matches the desired "look and feel" for your application, you can create a custom skin. This custom skin can be applied to all widgets within your ISC application so that they retain a unified appearance exactly matching the color scheme, font styles, and widget attributes you'd like to see. You should use one of the standard styles shipped with the SmartClient framework as a basis for your custom skin, so that you can achieve the desired appearance with the bare minimum of image changes. To create your own custom skin:

1. Copy one of the standard skins from the `[ISOMORPHIC_CLIENT]/skins` directory and paste it within your `[APP]/skins` directory. Then rename it.

2. Open the `load_skin.js` file, and change the `Page.setSkinDir` to your application directory (where you moved the skin) and the new name of your skin. The path provided here must be relative to your application file or the `[ISOMORPHIC_CLIENT]` directory. If you move a skin, you must change the value passed to the `setSkinDir` method accordingly.

   ```
   Page.setSkinDir("[APP]/skins/mySkin/");
   ```

3. Customize the cascading style sheet (CSS) classes for your new skin as you see fit. This allows you to change fonts, sizes, and colors for the widgets in your application and may be all you need to achieve the look you want.

Reference

See Appendix B, "*Isomorphic SmartClient Styles*," for a description of each of the CSS classes (styles) in the `skin_styles.css` file.

4. Each skin should contain an `unsupported_browser.html` file. This file automatically loads when users try to access your application using an unsupported browser and directs them to where they can obtain a supported browser. You may want to customize this page to match the rest of your site from within your custom skin. The `unsupported_browser.html` file is loaded with the following method call in `load_skin.js`.

   ```
   Page.checkBrowserAndRedirect("[SKIN]/unsupported_browser.html");
   ```

5. Replace existing icons and images for your new skin with custom ones. Since you are loading the skin from the page level, you should retain the existing names for the images in your new skin. This will ensure that all the widgets will pick up the images from your new skin using their default property settings.

6. Load your custom skin within any application files you create relative to your applications directory as appropriate.

   ```
   <HTML>
   <HEAD>
   <SCRIPT SRC=../isomorphic/system/Isomorphic_SmartClient.js></SCRIPT>
   <SCRIPT SRC=./skins/mySkin/load_skin.js></SCRIPT>
   ```

Specifying image directories

Specifying the skin to use, if the default is not acceptable, will only tell the ISC system what images and styles to use with widgets in your application. You must still specify where the images specific to your application reside using the methods given in Table 4.1. The widget image properties in Table 4.2 can alternatively be used to place a specific image into a widget as its contents. Whenever possible, you should avoid embedding directory paths directly in the image `src` filenames in order to make it easier to change your application to use different image paths.

Table 4.1: Image directory methods for a page

Method	**Description**
`Page.setAppDir(`*`path`*`)`	**Default:** The same directory as the invoked page. Sets the root directory for application-specific files. This defines a new value for the `[APP]` special directory. Other URLs you define using this special directory keyword will be resolved to their absolute paths. **Note:** The value for the *`application image directory`* property will be changed automatically based on the value you set for the *`application directory`* using this method. Therefore, if you have images stored in an `images` directory within your *`application directory`*, the *`application image directory`* will be set correctly, and you do not need to call the `setAppImgDir` method.
`Page.setAppImgDir(`*`path`*`)`	**Default:** `"[APP]/images/"` Sets the image directory for *all* application-specific images. You only need to call this method if the images are not in the expected location.
`Page.setIsomorphicDir(`*`path`*`)`	**Default:** `"../isomorphic/"` (relative to the application directory property) Sets the root directory for isomorphic-supplied files. This defines a new value for the `[ISOMORPHIC]` special directory. Other URLs you define using this special directory keyword will be resolved to their absolute paths.
`Page.setIsomorphicClientDir(`*`path`*`)`	**Default:** `"[ISOMORPHIC]/system/"` Sets the root directory for Isomorphic-supplied files. This defines a new value for the `[ISOMORPHIC_CLIENT]` special directory. Other URLs you define using this special directory keyword will be resolved to their absolute paths.
`Page.setSkinDir(`*`path`*`)`	**Default:** `"[ISOMORPHIC]/skins/standard/"` Sets the root directory for skin files. This defines a new value for the `[SKIN]` special directory. Other URLs you define using this special directory keyword will be resolved to their absolute paths.

If your application directory is not in the default location relative to your `isomorphic` directory, you must specify the relative location to ensure that the ISC libraries and styles can be located. This is done using the `Page.setIsomorphicDir(`*`path`*`)` method near the top of the page in your application file as follows.

```
<HTML>
<HEAD>
<!-- Load the ISC libraries relative to the application file -->
<SCRIPT SRC=../../isomorphic/system/Isomorphic_SmartClient.js></SCRIPT>

<!-- Specify the relative location of "isomorphic" as the IsomorphicDir -->
<SCRIPT>Page.setIsomorphicDir("../../isomorphic");</SCRIPT>

<!-- Specify the relative location of the skin to load for the application -->
<SCRIPT SRC=../../isomorphic/skins/mySkin/load_skin.js></SCRIPT>
```

Note

You can specify an image directory as a relative or absolute path. If the path parameter you specify begins with a `/`, `http://`, `https://`, or `file://`, the path will be assumed to be absolute. If you use a "special directory" variable like `[APP]`, `[ISOMORPHIC]`, `[ISOMORPHIC_CLIENT]`, or `[SKIN]`, or begin the path with a directory name, it will be assumed to be relative.

You can use the `Page` methods to set image directories in conjunction with widget properties to specify a path to custom images specific to that widget.

Table 4.2: Image directory properties/methods for a widget

Property	Description
widget.appImgDir	**Default:** `""` The default subdirectory used for application-specific images that are contents of this specific widget. This value is local to `Page.appImgDir`.
widget.skinImgDir	**Default:** `"images/"` The default subdirectory used for all skin images defined by this particular widget class. This value is local to `Page.skinImgDir`.

For example, the following script:

```
Page.setAppImgDir("[APP]/myImages");
Canvas.create({
    ID:"myWidget",
    appImgDir:"myWidget/",
    backgroundImage:"bg.gif"
});
```

tells `myWidget` to use `myImages/myWidget/bg.gif` as its background image.

Warning

Calling the `Page.setAppImgDir` method (or setting `widget.appImgDir`) will not automatically update images that have already been embedded in the page's HTML. You may redraw individual widgets to regenerate their HTML using the new image directory, or you could simply reload the page and set the appropriate image directory before any widgets containing images are drawn.

Images in Canvas widgets

If you are writing a custom *Canvas* subclass that incorporates images that you need to swap dynamically, you can use the following methods to generate the HTML for the image and swap its media.

Table 4.3: Image-related *Canvas* widget instance methods

Method	**Action**
`widget.imgHTML(src, [width], [height], [name], [options], [imgDir])`	Generates the HTML for an image unique to this Canvas. The full URL for the image will be formed according to the rules documented for canvas.getImgURL() The created image will have an identifier unique to this Canvas, and subsequent calls to getImage() and setImage() with the name passed to this function will act on the image object produced by the HTML returned from this call.
`widget.getImgURL(src, [imgDir])`	Returns the relative URL for an image to be drawn in this canvas. If the passed URL begins with the special prefix "[SKIN]", it will have the widget.skinImgDir and Page.skinImgDir prepended. Otherwise the image is assumed to be application-specific, and will have the widget.appImgDir and Page.appImgDir automatically prepended.
`widget.getImage(identifier)`	Returns the DHTML image object specified by `identifier`. The image element must have been created from HTML generated by calling widget.imgHTML() on this particular Canvas.
`widget.setImage(identifier, URL, [imgDir])`	Sets the URL of the DHTML image object specified by `identifier` to a new `URL`. The `URL` will automatically be prepended with `Page.appImgDir` and `widget.appImgDir`. The image element must have been created from HTML generated by calling widget.imgHTML() on this particular Canvas.

Example: Specifying HTML as a widget's contents

The `canvas_clip_scroll.html` example (shown in Figure 2.3 on page 38) set the contents property of several widgets to:

```
<IMG SRC=yinyang.gif WIDTH=200 HEIGHT=200>
```

An alternate approach is to defer setting contents until after initialization, and then do so by calling the `setContents` method with the `getImgHTML` method as follows:

```
widget.setContents(widget.imgHTML('yinyang.gif', 200, 200, 'yinyang'));
```

This approach ensures that the application and widget image directories are used, and allows you to subsequently change any widget's 'yinyang' image as follows:

```
widget.setImage('yinyang', newURL)
```

Example: Using a custom skin on a single class of widgets

If you want to use a particular skin for a single widget, regardless of what skins are used by all other widgets, you can specify the special skin directory to use with the `setSkinImgDir` method. For example, the following code specifies that a Macintosh style skin be used with the scrollbar widget. See the `scrollbar_custom.html` example code for more details.

```
Scrollbar.setSkinImgDir("[APP]/images/MacScrollbars/");
```

Note

The `setSkinImgDir` method is being called on the *Scrollbar* widget class. Therefore, this skin will be applied to all scrollbars.

Img widgets

The *Img* class implements a simple widget that displays a single image. Settable *Img* widget properties are listed in Table 4.4 (not including those inherited from the *Canvas* class).

Table 4.4: Img widget properties

Property	Value	Default
`src`	The base filename for the image. `Page.imgDir` and `this.imgDir` are prepended to this name to form the full URL.	`"blank.gif"`
`state`	A string representing a distinct state of this widget. If state is provided, the base filename is modified by inserting "`_state`" before the file extension.	`""`
`imageType`	Indicates whether the image should be tiled/cropped, stretched, or centered when the size of this widget does not match the size of the image. The following values can be set for this property: • `tile`—Tile the image. • `stretch`—Stretch the image to fit. • `center`—Center the image on the widget.	`"stretch"`

Img widgets also provide two setter methods for the above properties. These setter methods are described in Table 4.5.

Table 4.5: Img widget setter methods

Method	Action
`setSrc(src)`	Set src to *src*, and update the displayed image.
`setState(state)`	Set state to *state*, and update the displayed image.

Example: Img rollover

The `state` property provides an easy mechanism for displaying different variants of an image. The example file `img_rollover.html` (shown in Figure 4.3) uses the `setState` method to implement a simple rollover interaction.

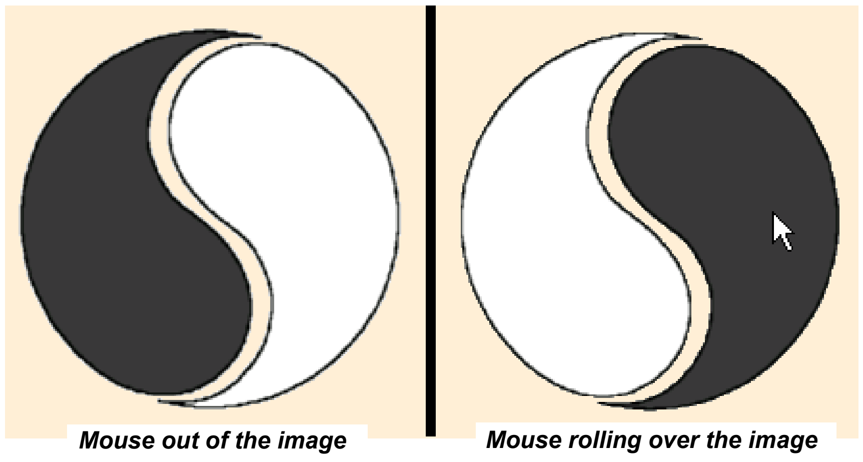


Figure 4.3: Example of Img rollover

The script for this rollover interaction follows:

```
var img = Img.create({
    left:100,
    top:100,
    width:200,
    height:200,
    src:"yinyang.gif",
    mouseOver:"this.setState('inverted')",
    mouseOut:"this.setState('')"
});
img.draw();
```

When the mouse moves over the widget in this example, the state name '`inverted`' is inserted into the base filename, and the image in '`yinyang_inverted.gif`' is displayed (as shown above in Figure 4.3). When the mouse moves out of the widget, state is set to an empty string and the image in the unmodified base filename ('`yinyang.gif`') is displayed.

StretchImg widgets

The *StretchImg* class implements a widget type that displays a list of multiple images. Settable *StretchImg* widget properties (not including those inherited from the *Canvas* class) are listed in Table 4.6.

Table 4.6: StretchImg widget properties

Property	**Value**	**Default**
`vertical`	Indicates whether the list of images is drawn vertically from top to bottom (`true`), or horizontally from left to right (`false`).	`true`
`src`	The base filename for the images. `Page.imgDir` and `this.imgDir` are prepended to this name to form the full URL.	`"blank.gif"`

Property	Value	Default
`items`	The list of images to display; an array of objects specifying the image names, sizes, and states. See below for details.	varies (see below)
`capSize`	If the default items are used, `capSize` is the size in pixels of the first and last images in this `stretchImg`.	`2`
`state`	A string representing a distinct state of this widget. If state is provided, the image filenames are modified by inserting "`_state`" between the base filename and the image name.	`""`
`imageType`	Indicates whether the image should be tiled/cropped, stretched, or centered when the size of this widget does not match the size of the image. The following values can be set for this property: • `tile`—Tile the image. • `stretch`—Stretch the image to fit. • `center`—Center the image on the widget.	`"stretch"`

StretchImg widgets also provide a single setter method, described in Table 4.7.

Table 4.7: StretchImg widget setter method

Method	Action
`setState(statename, [itemName])`	Set the specified image's state to `stateName` and update the displayed image, or set the state for all images to `stateName` and update the displayed images if `itemName` is not provided.

The items property of a StretchImg widget

The `items` property of a `stretchImg` should be set to a list of objects of the format:

```
items:[
    {height:"capSize", name:"start", width:"capSize"},
    {height:"*", name:"stretch", width:"*"},
    {height:"capSize", name:"end", width:"capSize"}
]
```

Where:

- The `name` property is the name of the image; the image's filename is generated by inserting '`_name`' before the file extension in the base filename.
- The `height` and `width` properties specify the size of the image as either: an absolute number of pixels, a named property of this widget that specifies an absolute number of pixels, a percentage of the remaining space (e.g. '`60%`'), or '`*`' (the default value) to allocate an equal portion of the remaining space. If the images are drawn vertically, the `height` property is used, and if the images are drawn horizontally, the `width` property is used.
- The optional `state` property (not shown above) overrides any state specified for the widget as a whole.

StretchImg widget image file names

The image files for a `stretchImg` should therefore be named in the following format:

```
srcBase_name.srcExt
   or
srcBase_state_name.srcExt
```

Where:

- The `srcBase` is given as the part of src before the dot.
- The `state` property is the state name, specified either for the entire `stretchImg`, or for an individual image in the `items` array.
- The `name` property is the image name, specified in the `items` array.
- The `srcExt` is given as the file extension that follows the dot in src.

Example: Image names using the default value for items

The default value for items is:

```
items:[
   {height:"capSize", name:"start", width:"capSize"},
   {height:"*", name:"stretch", width:"*"},
   {height:"capSize", name:"end", width:"capSize"}
]
```

Consider the following example widget:

```
StretchImg.create({
   ID:"stretchImg",
   src:"myStretch.gif",
   state:"on"
});
```

This example would use the image files named:

```
myStretch_on_start.gif
myStretch_on_stretch.gif
myStretch_on_end.gif
```

Note If you need to use the same image more than once in a `stretchImg`, you can specify more than one item with the same name in the `items` array. However, you should be aware that the `setState` method will only set the state of the first item in the array with the specified *`itemName`* (if not empty). If this poses a problem, you'll need to create duplicate image files with different names.

CHAPTER 5

Labels, Buttons, and Bars

The four most basic user interface widget types supported by the Isomorphic SmartClient framework are: labels, buttons, scrollbars, and progressbars.

- The *Label* widget class is used to display a text label.
- The *Button* widget class is used to display interactive, style-based buttons.
- The *Scrollbar* widget class is used to display cross-platform image-based scrollbars that control the scrolling of content in other widgets.
- The *Progressbar* widget class is used to display graphical bars where the length represents percentages, typically, of task completion.

This chapter discusses each of these basic widgets with their settable properties and methods, and presents how to specify them within your own ISC applications.

In this chapter:

Label widgets

The *Label* class implements a simple widget type that displays a text label. *Label* widgets have the properties given in Table 5.1, in addition to those inherited from the *Canvas* class.

Table 5.1: Label widget properties

Property	Value	Default
`align`	Horizontal alignment of label text. The possible values are: • `left` • `center` • `right`	`"left"`
`valign`	Vertical alignment of label text. The possible values are: • `top` • `center` • `bottom`	`"center"`

Use the `contents` property to set the HTML-formatted text for a label as shown in the following code:

```
Label.create({
    ID:"myLabel",
    left:50,
    top:75,
    height:50,
    contents:"<B>This label is centered</B>",
    align:"center",
    border:"groove blue 4px",
    backgroundColor:"lightgrey"
});
```

Button widgets

The *Button* class implements interactive, style-based button widgets. *Button* widgets provide the properties listed in Table 5.2 in addition to those inherited from the *Canvas* class.

Table 5.2: Button widget properties

Property	Value	Default
`title`	The text title to display in this button.	`"Untitled Button"`
`wrap`	A boolean indicating whether the button's title should word-wrap, if necessary.	`false`
`selected`	A boolean indicating the selection state of this button; affects button appearance.	`false`

Property	**Value**	**Default**
`actionType`	Specifies this button's selection behavior when clicked. Possible values are: • `button` • `checkbox` • `radio`	`"button"`
`showRollOver`	A boolean indicating whether this button should change its appearance when the mouse rolls over the button while up.	`true`
`showDown`	A boolean indicating whether this button should change its appearance when the mouse is pressed while over the button.	`true`
`align`	Horizontal alignment of this button's title. See the *Label* widget properties above for possible values.	`"center"`
`valign`	Vertical alignment of this button's title. See the *Label* widget properties above for possible values.	`"center"`
`baseStyle`	The base name for the CSS class applied to this button when deselected. Appended with "`Over`", "`Down`", "`Disabled`", "`Selected`", "`SelectedOver`", "`SelectedDown`" or "`SelectedDisabled`" when the button is in one of those states. See below for details.	`"button"`

In addition to these new properties, the *Button* class overrides the following inherited properties to set different default values:

```
height:25
cursor:"hand"
overflow:"hidden"
```

The visual style of a button is defined by the CSS classes whose base names are set in `baseStyle`. For the default settings of these properties, the following eight CSS classes are defined:

- `button`
- `buttonOver`
- `buttonDown`
- `buttonDisabled`
- `buttonSelected`
- `buttonSelectedOver`
- `buttonSelectedDown`
- `buttonSelectedDisabled`

A button's `actionType` property specifies its selection behavior as follows:

- `button`

  The button ignores selection state. It is drawn in the default deselected styles after any click.

- `checkbox`

  The button toggles its selection state. It is drawn in the selected style after being clicked when deselected, and in the deselected styles after being clicked when selected.

- `radio`

  The button is always selected when clicked. It is drawn in the selected styles after any click.

Selection state is reflected in the visual style applied to the button. A button's selection state can also be set or checked via the methods listed in Table 5.3.

Table 5.3: Button widget methods

Method	Action
`select()`	Sets selected to `true`, and displays the button in the appropriate selected style.
`deselect()`	Sets selected to `false`, and displays the button in the appropriate selected style.
`isSelected()`	Returns the value of the `selected` property.

Example: Button selection

The example file `button_selection.html` (shown in Figure 5.1) demonstrates the three button `actionType` settings, and the effects of selecting and disabling buttons.

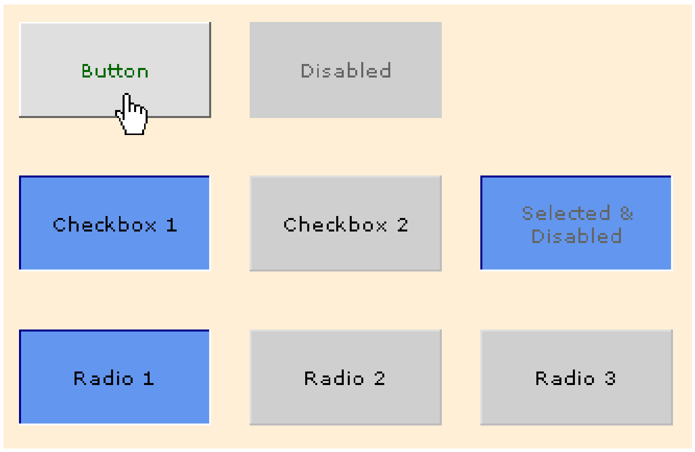


Figure 5.1: Example of button selection

This example draws three rows of *Button* widgets:

- The buttons in the first row have `actionType:"button"`, and the second button is disabled (`enabled:false`).
- The buttons in the second row have `actionType:"checkbox"`, the first and third buttons are selected (`selected:true`), and the third button is disabled (`enabled:false`).
- The buttons in the third row have `actionType:"radio"`, and the first button is selected (`selected:true`).

All of the buttons have click handlers that call `showClicked(this.title)` to display their titles in the window status bar when clicked. The buttons in the third row also execute the following script when clicked:

```
if (selectedButton != this) {
        selectedButton.deselect();
        selectedButton=this
    };
```

This script implements the mutual exclusivity expected of radio buttons in a group. This behavior is not built into the *Button* class, as it requires a higher-level container to keep track of which button is currently selected. That task falls instead to the `selectedButton` variable defined at the end of this example:

```
    var selectedButton = rb1;
```

Mutual exclusivity of radio buttons is implemented in the *Toolbar* class, and is supported by native (non-widget) radio buttons in forms. Refer to Chapter 6 for more information on forms, and Chapter 9 for details on the *Toolbar* class.

Scrollbar widgets

The *Scrollbar* class extends the *StretchImg* class to implement cross-platform, image-based scrollbars that control the scrolling of content in other widgets. Widgets with an overflow setting of `scroll` or `auto` will create *Scrollbar* widgets automatically when required. See *"Clipping and scrolling" on page 36* for details. Some browsers provide scriptable native scrollbars, so *Scrollbar* widgets are not created by default on those platforms. However, you may want to manually create *Scrollbar* widgets in order to customize and control:

- Size— Making the scrollbars narrower or wider than 16 pixels, or longer or shorter than the scrolled widget,
- Positioning— Placing the scrollbars outside of the scrolled widget's rectangle, and/or on different sides of the scrolled widget,
- Images— Specifying custom graphics for the arrows, tracks, and thumbs of different scrollbars, and
- Enabled state— Enabling or disabling scrollbars as you require.

Scrollbar widgets provide the properties listed in Table 5.4 in addition to those inherited from the *Canvas* and *StretchImg* classes:

Table 5.4: Scrollbar widget properties

Property	Value	Default
`btnsize`	The size of the square buttons (arrows) at the ends of this scrollbar. This overrides the width of a vertical scrollbar or the height of a horizontal scrollbar to set the scrollbar's thickness.	`16`
`showCorner`	If `true`, displays a corner piece at the bottom end of a vertical scrollbar, or the right end of a horizontal scrollbar. This is typically set only when both horizontal and vertical scrollbars are displayed and abut the same corner.	`false`
`scrollTarget`	The widget whose contents should be scrolled by this scrollbar. The scrollbar thumb is sized according to the amount of visible vs. scrollable content in this widget.	`null`

Property	Value	Default
autoEnable	If true, this scrollbar will automatically enable when the scrollTarget is scrollable (i.e., when the contents of the scrollTarget exceed its clip size in the direction relevant to this scrollbar), and automatically disable when the scrollTarget is not scrollable. Set this property to false for full manual control over a scrollbar's enabled state.	true
thumbMinSize	The minimum pixel size of the draggable thumb regardless of how large the scrolling region becomes.	12
allowThumbStateDown	If true, the thumb's appearance changes when it's clicked on.	false
thumbResizeThreshold	The percentage that the thumb's size has to change before it is resized.	0.1 (10%)

The `vertical` property inherited from *StretchImg* determines whether a scrollbar is vertical or horizontal.

The standard images for all widgets, by default, reside within the Isomorphic skin directory loaded with the application, `[ISOMPORPHIC]/skins/standard/`. The *Scrollbar* widget's `skinImgDir` property specifies the subdirectory—`images/Scrollbar/`—within the skin root for images specific to *Scrollbar* widgets. These directories can be changed to pick up custom images by using the `Scrollbar.setSkinImgDir` method. See *"Using and customizing ISC skins" on page 75* for more information.

Note

If you change the size of the images to be used by *Scrollbar* widgets, you will need to change `btnSize` (default: `16`) and `capsize` (default: `2`) properties for them to match your new image sizes. The `btnSize` specifies the width of the scrollbar and the `capsize` property, inherited from the *StretchImg* widget class, specifies the size in pixels of the edges of the scroll thumb.

Example: Custom scrollbars

The example file `scrollbar_custom.html` (shown in Figure 5.2) demonstrates the manual creation and control of customized *Scrollbar* widgets.

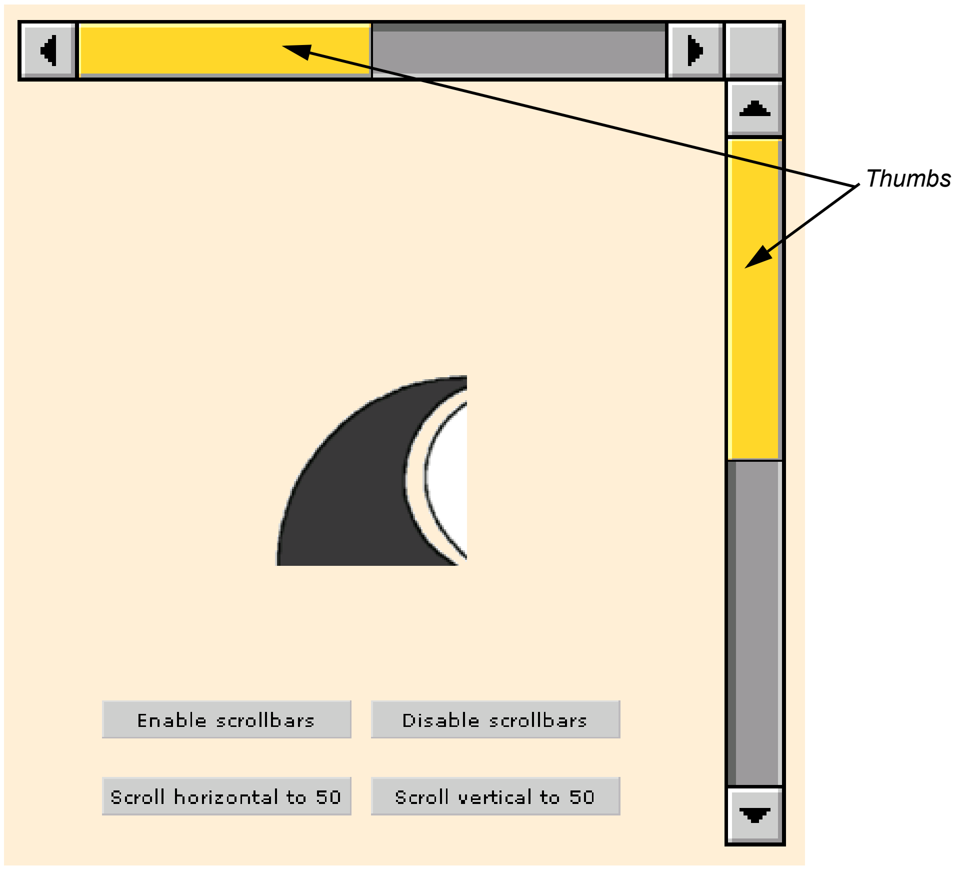


Figure 5.2: Example of custom scrollbars

This example creates a 100-by-100-pixel *Canvas* widget containing a 200-by-200-pixel image, with `overflow:"hidden"` so the image will be clipped to the widget's area. The example also creates two scrollbars, both with `scrollTarget` set to the scrollable *Canvas* widget, as follows:

```
Scrollbar.setSkinImgDir("[APP]images/MacScrollBars/");

Canvas.create({
    ID:"scrolledWidget",
    left:150,
    top:225,
    width:100,
    height:100,
    contents:Canvas.imgHTML("yinyang.gif", 200, 200),
    overflow:"hidden"
});

var scrollbarWidth = 32;

Scrollbar.create({
    ID:"hscrollbar",
    left:16,
    top:41,
    width:400,
    vertical:false,
    btnSize:scrollbarWidth,
    showCorner:true,
    imageType:"stretch",
    scrollTarget:scrolledWidget,
    autoEnable:false
});

Scrollbar.create({
    ID:"vscrollbar",
    left:hscrollbar.left + hscrollbar.width - scrollbarWidth,
    top:hscrollbar.top + scrollbarWidth - 2,
    height:400,
    btnSize:scrollbarWidth,
    imageType:"stretch",
    scrollTarget:scrolledWidget,
    autoEnable:false
});
```

These scrollbars demonstrate the following visible customizations:

- The horizontal scrollbar is positioned roughly 200 pixels above the scrollable widget, and displays a corner piece.
- The vertical scrollbar is positioned roughly 150 pixels to the right of the scrollable widget.
- Both scrollbars are double the normal thickness (32 pixels, instead of 16).
- The scrollbars use images from a different skin directory—`[APP]images/MacScrollBars/`.

Below the scrollable widget, two buttons implement click handlers that enable and disable the scrollbars:

```
click:"hscrollbar.enable();vscrollbar.enable()"

click:"hscrollbar.disable();vscrollbar.disable()"
```

Progressbar widgets

The *Progressbar* class extends the *StretchImg* class to implement image-based progress bars (graphical bars whose lengths represent percentages, typically of task completion). *Progressbar* widgets provide the properties given in Table 5.5 in addition to those inherited from the *Canvas* and *StretchImg* classes.

Table 5.5: Progressbar widget properties

Property	Value	Default
`percentDone`	Number from `0` to `100`, inclusive, for the percentage to be displayed graphically in this `progressbar`.	`0`

Property	Value	Default
`breadth`	Thickness of the progressbar in pixels. This is effectively width for a vertical progressbar, or height for a horizontal progressbar.	`20`
`length`	Length of the progressbar in pixels. This is effectively height for a vertical progressbar, or width for a horizontal progressbar.	`100`

Progressbar widget image file names

The default image file names for *Progressbar* images are the following:

- `progressbar_h_start.gif`
- `progressbar_h_stretch.gif`
- `progressbar_h_end.gif`
- `progressbar_h_empty_start.gif`
- `progressbar_h_empty_stretch.gif`
- `progressbar_h_empty_end.gif`

Vertical *Progressbar* widgets use `_v_` in place of `_h_` in the file names above.

The `items` array must contain exactly six objects, in the order specified above. You can change the `size` property of these objects to accommodate your custom images, but any other changes to this array are not recommended. Sizes for the second and fourth objects in the array (the two "stretch" images) are not actually used. These images are sized based on the `percentDone` property of the `progressbar`.

Progressbar widget setter method

Progressbar widgets also provide a single setter method, described in Table 5.6.

Table 5.6: Progressbar widget method

Method	Action
`setPercentDone(`*`percentage`*`)`	Sets `percentDone` to *`percentage`*, and redraws this `progressbar` to show the new percentage graphically.

Example: A progressbar widget

The example file `progressbar.html` (shown in Figure 5.3) demonstrates the creation and updating of a *Progressbar* widget.

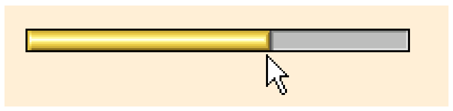


Figure 5.3: Example of a progressbar widget

This example creates a single `progressbar` as follows:

```
Progressbar.create({
   ID:"pb",
   left:20,
   top:20,
   length:200,
   percentDone:50,
});
```

A global handler for the '`mouseMove`' event sets the percentage of the `progressbar` to the x-coordinate of the mouse on the page:

```
Page.setEvent("mouseMove", "pb.setPercentDone(Math.round(100*(pb.getOffsetX()/
   pb.getWidth())))");
```

When the mouse is moved to the left side of the browser window with respect to the `progressbar`, `percentDone` will be `0` and the `progressbar` will be empty. When the mouse is then moved to the right with respect to the `progressbar`, `percentDone` will be 100 and the `progressbar` will be full.

Note

The `percentDone` property will not exceed `100`, even if a higher value is passed to the `setPercentDone` method.

C H A P T E R   6

Forms

The Isomorphic SmartClient system includes a class of widgets for generating and manipulating forms on web pages. The *DynamicForm* widget class provides a logical representation of an HTML form, using a collection of form item objects to handle:

- generating form HTML,
- laying out form elements,
- getting and setting form element values, and
- validating user input before submitting a form.

This allows the developer to focus on the logic and structure of forms and data rather than the appearance and functionality of the forms themselves. This chapter covers how to create forms within the ISC system.

In this chapter:

Specifying a form

DynamicForm widgets provide fundamental properties for specifying a form, as described in Table 6.1.

Note

The terms *item* and *element* in this chapter have precise meanings, referring to two different types of objects. A form ***item*** is a JavaScript object that defines and manipulates an HTML form ***element*** (field, button, etc.). Items provide the properties and methods that you use to create and work with form elements. You do not typically access form elements directly when using *DynamicForm* widgets.

The term ***form*** is used in this chapter to refer both to *DynamicForm* widgets and to the HTML forms they generate, depending on context.

Table 6.1: DynamicForm fundamental widget properties

Property	Value	Default
`items`	An array of form-item objects, defining the elements of the form. See *"Specifying form items" on page 100* for details.	`[]`
`values`	A property list of `itemName:value` pairs, specifying the current set of values for the form elements. See *"Working with form item values" on page 110* for more information.	`{}`
`errors`	A property list of `itemName:errorMessage` pairs, specifying the set of error messages displayed with the corresponding form elements. Each `errorMessage` may be either a single string or an array of strings. See *"Validating form input" on page 112* for more information.	`{}`
`action`	The URL to which the form will submit its values.	`"#"`
`target`	The name of a window or frame that will receive the results returned by the form's action. The default `null` indicates to use the current frame.	`null`
`method`	The mechanism by which form data is sent to the action URL: • `post` for HTTP POST • `get` for HTTP GET	`"post"`

Reference

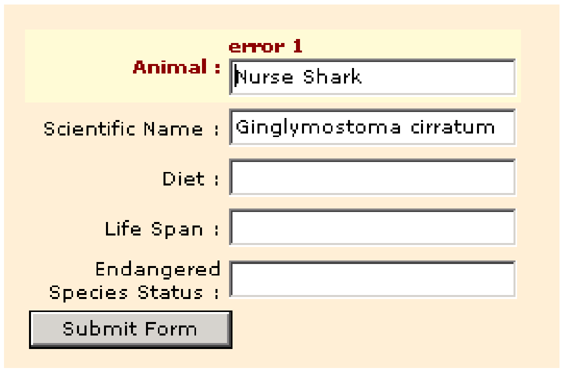
Refer to Appendix A, "*Widget Initialization Templates*," for a *DynamicForm* initialization template that includes all *DynamicForm* properties available for initialization. The template includes properties documented in this chapter and relevant properties inherited from the *Canvas* class.

Tip

Since the `items`, `values`, and `errors` properties often contain many values, you may want to set them to variables whose values are defined outside of the form initialization block. This will improve the readability of your code, especially if you generate these properties with server-side programming. *"Example: Form initialization"* below takes this external-variable approach.

Example: Form initialization

The example file `dynamicForm_init.html` (shown in Figure 6.1) creates and draws a simple form.



Animal :	error 1 Nurse Shark
Scientific Name :	Ginglymostoma cirratum
Diet :	
Life Span :	
Endangered Species Status :	

Figure 6.1: Example of form initialization

The following script creates this form:

```
var formItems = [
        {name:"commonName", title:"Animal", type:"text"},
        {name:"scientificName", title:"Scientific Name", type:"text"},
        {name:"diet", title:"Diet", type:"text"},
        {name:"lifeSpan", title:"Life Span", type:"text"},
        {name:"status", title:"Endangered Species Status", type:"text"},
        {title:"Submit Form", type:"submit"}
    ],

    formValues = {
        commonName:"Nurse Shark",
        scientificName:"Ginglymostoma cirratum"
    },

    formErrors = {
        commonName:"error 1"
    }
;
DynamicForm.create({
    ID:"simpleForm"
    left:20,
    top:45,
    items:formItems,
    values:formValues,
    errors:formErrors
});
```

This form contains six elements, five TEXT elements and one SUBMIT element, that are defined by the object initializers in the `formItems` array (and therefore in `simpleForm.items`, which is initialized to this array). The following section on *"Specifying form items"* discusses the various types of items that may be specified in a *DynamicForm* widget.

The `formValues` and `formErrors` property lists initialize `simpleForm.values` and `simpleForm.errors`, respectively. The property names in these lists correspond to the values of the name property (i.e., '`field1`' and '`field2`') for the corresponding items in the `formItems` array.

Specifying form items

DynamicForm widgets provide a default set of items that fall into three general categories: data items, button items, and display items. Table 6.2, Table 6.3, and Table 6.4 list all of the supported form-item types, grouped by these categories.

Table 6.2: DynamicForm data item types

Form Item	Description	HTML Form Element(s)
`text`	A single-line text input field.	`<INPUT TYPE="TEXT">`
`password`	A single-line text input field that displays input characters as asterisks or dots to conceal the input value.	`<INPUT TYPE="PASSWORD">`
`upload`	A single-line text input field for entering the path to a file, accompanied by a **Browse** button for selecting a file via standard file dialog boxes. **Note:** You can not set a `defaultValue` or `defaultDynamicValue` for an `upload` field. If set, these properties are ignored.	`<INPUT TYPE="FILE">`
`textArea`	A multi-line, scrollable text input field. Using the all lower case `textarea` is also accepted.	`<TEXTAREA>`
`checkbox`	A toggle switch input, with a "hot" label that selects/ deselects the checkbox when clicked. **Note:** If a checkbox is used with a valueMap, checked will equate to `"true"`, and unchecked to `"false"`.	`<INPUT TYPE="CHECKBOX">`
`radioGroup`	A set of mutually exclusive radio buttons, each with a "hot" label that selects the button when clicked.	`<INPUT TYPE="RADIO">` (multiple elements)
`select`	A selection list input.	`<SELECT>`
`selectOther`	A selection list input with an **Other...** option, which brings up a prompt that can be used to enter a different value.	`<SELECT>`
`date`	A group of selection lists for specifying a day, month, and year.	`<SELECT>` (three elements) and `<INPUT TYPE="HIDDEN">`

Form Item	Description	HTML Form Element(s)
`time`	An input field that accepts and displays a value formatted as a time.	`<INPUT TYPE="TEXT">`
`hidden`	An invisible field.	`<INPUT TYPE="HIDDEN">`

Table 6.3: DynamicForm button item types

Form Item	Description	HTML Form Element(s)
`button`	An arbitrary button.	`<INPUT TYPE="BUTTON">`
`submit`	A button that attempts to submit the form when clicked.	`<INPUT TYPE="SUBMIT">`
`reset`	A button that resets the form values when clicked. See the `resetValues` method in Table 6.11 on page 111.	`<INPUT TYPE="RESET">`
`toolbar`	A set of button items displayed in a single row, with configurable spacing.	Any combination of the above elements.

Table 6.4: DynamicForm display item types

Form Item	Description
`staticText`	A static piece of text with a title. Used for showing an unchangeable value, such as a key.
`blurb`	A static piece of text with no title. Used for descriptive text.
`header`	A static piece of text displayed in a header style.
`rowSpacer`	A vertical spacer to separate form elements.
`spacer`	A blank cell of specified height and width.

Each item is defined by an object in the `dynamicForm.items` array. In the *"Example: Form initialization"* above, a textfield item was defined with the following object initializer:

```
{name:"field1", title:"field 1", type:"text"}
```

The properties of each form item control its behavior, layout, appearance, validation, and other characteristices. The fundamental properties of a form item are described in Table 6.5.

Table 6.5: Form item fundamental properties

Property	Description	Default
`name`	The name of the item, used both in scripts (e.g. `form.values.itemName`) and as the field name when form data is submitted.	`null`
`title`	The text of the item's title, if it has one. See *"Form item annotations and styles" on page 108* for more information.	`null`

Property	Description	Default
`defaultValue`	The default value for the item, if none is provided in the *`dynamicForm`*`.values` property list. See *"Working with form item values" on page 110* for more information. **Note:** You can not set a `defaultValue` or `defaultDynamicValue` for an `upload` field. If set, these properties are ignored.	`null`
`defaultDynamicValue`	A default value for the item, determined dynamically.Set `defaultDynamicValue` to a string of script to evaluate for the item's value, if *`form`*`.values[`*`itemName`*`]` is not defined. In this script, you can use: • '`form`' as a reference to the *DynamicForm* widget, and • '`item`' as a reference to the form item object.	`null`
`type`	The type of the item; one of the types listed in the first columns of the tables above.	`text`
`prompt`	Displays an instructional prompt in the status bar (all browsers) and a tool tip when a mouse moves over a form item.	Varies based on the form item.
`valueMap`	**(For `select`, `selectOther`, and `radioGroup` items only)** An array or property list (or an expression that evaluates to an array or property list) that specifies the possible values for the item. If an array, the value for each option is used both internally and for display. If a property list, the property names are the actual internal values while the property values are displayed in the form. See *"Working with form item values" on page 110* for more information.	`null`
`redrawOnChange`	Items with `redrawOnChange` set to `true` will cause the entire form to be redrawn when their contents are changed. This is useful if any items have a `showIf` property. See *"Controlling form layout" on page 105*.	`false`
`validateOnChange`	Items with `validateOnChange` set to `true` will undergo any validation checks set for them once a change is made to their form element contents and the focus is removed. See *"Validating form input" on page 112* for details on validation.	`false`

DynamicForm items support many more properties, which are discussed in subsequent sections of this chapter.

Reference

Appendix A, "*Widget Initialization Templates*," includes basic initialization templates for all item types, covering the most commonly-used properties for each type. Appendix A also includes a generic item-initialization template, covering all item properties except for a handful of type-specific properties.

Warning

Since item names are used as data field names when a form is submitted, you must specify a name for every data item in a form you wish to submit. Item names must also be unique within a form, to avoid conflicts when referring to items or data fields by name. Names for button and display items are optional.

Form-item properties are typically set during initialization only. However, since items are JavaScript objects, you can access their properties and methods as you would those of any other object. The `dynamicForm.getItem` method returns a reference to an item that you can use in your scripts:

```
form.getItem(name)
```

For example, in the *"Example: Form initialization,"* you could access the title of the first textfield item (named '`field1`') with the following script:

```
simpleForm.getItem('field1').title
```

If you're working in the opposite direction (i.e. if you already have a reference to an item object and need to access the dynamicForm in which it is used), you can use the item's form property:

```
item.form
```

For example:

```
item.form.getValues(property)
```

Tip

The `getValues` method simply returns the current value in the form, but there is no guarantee that what has been entered is valid. To ensure valid data, you should call the `form.validate()` method prior to the call to `form.getValues()`, and then only proceed if `form.validate()` returns `true`.

Example: Form items

The example file `dynamicForm_item_types.html` (shown in Figure 6.2) creates and draws a form containing one of each data, button, and display item types listed above in Table 6.2, Table 6.3, and Table 6.4.

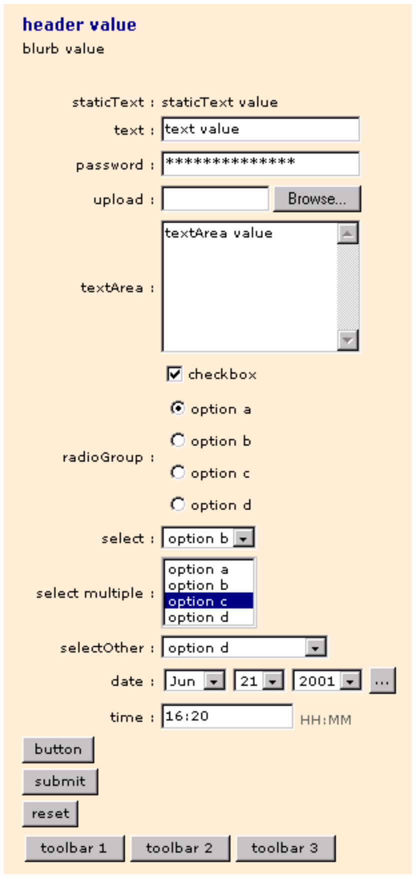


Figure 6.2: Example of form items

Every item in this form that takes a value (i.e., every data item and every display item except the separator) is given a unique name. The default value for each of these items is set in its object initializer with the `defaultValue` property.

Tip

Since all of these items are named, their values (see exception below) could instead be initialized by setting the form's values property to the following object literal:

```
{item1:"header value",
item2:"text value",
item3:"label value",
item4:"textfield value",
item5:"password value",
item7:"textarea value",
item8:true,
item9:"a",
item10:"b",
item11:"c",
item12:"d",
item13:new Date(1971, 9, 25),
item14:"hidden value"}
```

You cannot initialize a value for `upload` fields. For this reason, there is no value set for `item6`, the `upload` field, above.

This example also demonstrates the use of arrays and property lists to specify the options of `radioGroup`, `select`, and `selectOther` items. The options of the `radioGroup` item and the first select item are initialized to an array of values that are used both internally and for display:

```
["a", "b", "c", "d"]
```

The options of the second `select` item and the `selectOther` item are initialized to a property list with different internal/display values (`valueMap`):

```
{a:"option a", b:"option b", c:"option c", d:"option d"}
```

Controlling form layout

A *DynamicForm* widget generates its associated HTML form in an invisible grid of rows and columns. Each visible form item is positioned and sized in this grid. You can control the layout of form items with the properties described in Table 6.6.

Table 6.6: Form layout properties

Property	Description	Default
form.width	Width in pixels of the entire form.	100
form.numCols	The number of columns of titles and items in this form's layout grid. A title and corresponding item each have their own column, so to display two form elements (each having a title and item), you would set this property to `4`.	2
form.colWidths	An array of widths for the columns of items in this form's layout grid. If specified, these widths should sum to the total width of the form (*form*.width). If not specified, the width of this form will be divided equally between its columns. Acceptable values for *form*.colWidths include: • a number (e.g. `100`)—The number of pixel widths to allocate to a column. • a percent (e.g. `20%`)—The percentage of the total *form*.width to allocate to a column. • `"*"` (all)—Allocate remaining width (*form*.width minus all specified column widths). Multiple columns can use `"*"`, in which case remaining width is divided between all columns marked `"*"`.	null
form.cellPadding	The amount of empty space, in pixels, surrounding each form item within its cell in the layout grid.	2
form.cellBorder	Width of border for the table that form is drawn in. This is primarily used for debugging form layout.	0
item.visible	Indicates whether this item should be included in the form. If `false`, the item will not be displayed and will not be included when the form is submitted. To include an invisible data field, use the `hidden` item type instead.	true

Property	Description	Default
item.showIf	A string of script that, if provided, is evaluated to conditionally determine the value of this item's `visible` property when the form is drawn or redrawn. In this script, you can use: • '`form`' as a reference to the dynamicForm widget, and • '`item`' as a reference to the form item object.	`null`
item.width[1]	The width of this item. Acceptable values for *item*.`width` include: • a number (e.g. `100`)—The number of pixel widths to allocate to a column, • a percent (e.g. `"20%"`)—The percentage of the total *item*.`width` to allocate to a column, and • `"*"` (all)—Allocate remaining width (*item*.`width` minus all specified column widths). Multiple columns can use `"*"`, in which case remaining width is divided between all columns marked `"*"`.	`"*"`
item.height[1]	The height of this item in pixels.	`20`
item.align	The horizontal alignment of this item within its cell in the form layout grid.	`"left"`
item.colSpan	The number of columns this item should occupy in the form layout grid, or '`*`' if the item should occupy all remaining columns in its row. Keep in mind that each title has a column as well as each item.	`2`
item.rowSpan	The number of rows this item should occupy in the form layout grid.`"*"` is not currently supported for *item*.`rowSpan`.	`1`
item.startRow	Indicates whether this item should always start a new row in the form layout grid.	`false`
item.endRow	Indicates whether this item should always be the last in its row in the form layout grid.	`false`

Note

If the actual width of a form element is greater than the width of its column(s) in the form layout, other columns may be compressed and/or the overall width of the form may be expanded to accommodate the element.

Example: Form layout

The example file `dynamicForm_layout.html` (shown in Figure 6.3) uses many of the properties listed above to generate the form.

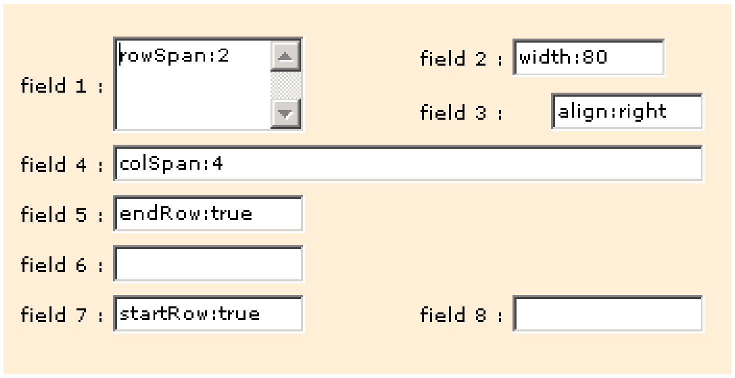


Figure 6.3: Example of form layout

The form items in this example are defined by the following object list:

```
var formItems = [
   {name:"item1", title:"field 1", width:100, height:50, rowSpan:2,
      type:"textarea"},
   {name:"item2", title:"field 2", width:80},
   {name:"item3", title:"field 3", width:80, align:"right"},
   {name:"item4", title:"field 4", width:308, colSpan:4},
   {name:"item5", title:"field 5", width:100, endRow:true},
   {name:"item6", title:"field 6", width:100},
   {name:"item7", title:"field 7", width:100, startRow:true},
   {name:"item8", title:"field 8", width:100}
]
```

The form itself is initialized as follows:

```
DynamicForm.create({
   ID:"layoutForm",
   width:400,
   numCols:4,
   items:formItems,
   values:formValues
});
```

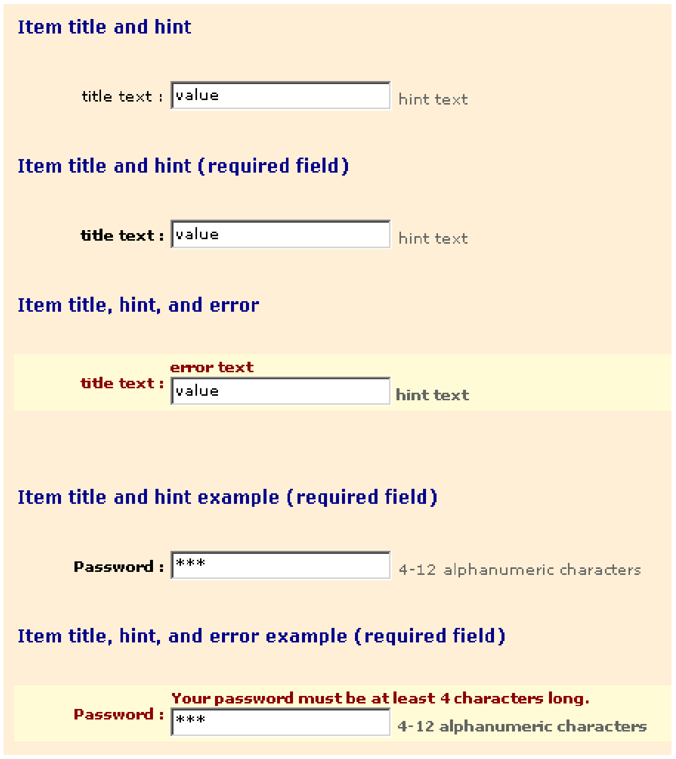
Form item annotations and styles

Form items support three different kinds of annotations (pieces of text accompanying a form element):

- a *title*, specified by the `title` property discussed earlier in this chapter,
- a *hint*, specified by the `hint` property, that can provide instructions or explanation of a form element to the user, and
- an *error*, either assigned by the *DynamicForm* widget's built-in validation mechanisms (see *"Validating form input" on page 112*), or explicitly set in the `errors` array by client-side scripting or server-side logic.

Example: Form item annotations

The example file `dynamicForm_annotations.html` (shown in Figure 6.4) presents the default layout and style of these annotations.


Item title and hint

title text : value hint text

Item title and hint (required field)

title text : value hint text

Item title, hint, and error

error text
title text : value **hint text**

Item title and hint example (required field)

Password : *** 4-12 alphanumeric characters

Item title, hint, and error example (required field)

Your password must be at least 4 characters long.
Password : *** **4-12 alphanumeric characters**

Figure 6.4: Example of form item annotations

Item errors are typically set in the course of validating form data, so most of the properties and methods affecting errors are discussed in *"Validating form input" on page*

112. Item titles and hints, however, are configured explicitly with the properties listed in Table 6.7.

Table 6.7: Form title/hint properties

Property	Description	Default
`item.title`	The text of this item's title (as listed in Table 6.5).	`null`
`item.hint`	The text of this item's hint.	`null`
`item.showTitle`	Indicates whether this item's title should be displayed.	`true`
`item.showHint`	Indicates whether this item's hint should be displayed.	`true`
`item.titleAlign`	The alignment of this item's title within its cell in the form layout. Acceptable values for `item.titleAlign` include: • `left` • `center` • `right`	`"right"`
`item.titleOrientation`	The positioning of this item's title relative to its form element. Acceptable values for `item.titleOrientation` include: • `left` • `right` • `top`	`"left"`
`form.titleWidth`	The width in pixels allocated to the title of every item in this form.	`100`
`form.titlePrefix`	The string prepended to the title of every item in this form.	`""`
`form.rightTitlePrefix`	The string prepended to the title of every item in this form if the `titleOrientation` property is set to `"right"`.	`":&nbsp;"`
`form.titleSuffix`	The string appended to the title of every item in this form.	`"&nbsp;:"`
`form.rightTitleSuffix`	The string appended to the title of every item in this form if the `titleOrientation` property is set to `"right"`.	`""`
`form.highlightRequiredFields`	Indicates whether the titles of required items in this form should use the special prefix and suffix specified by the next two properties, instead of the standard prefix and suffix. See *"Validating form input" on page 112* for details.	`true`
`form.requiredTitlePrefix`	The string prepended to the title of every required item in this form if `highlightRequiredFields` is `true`.	`"<B>"`
`form.requiredRightTitlePrefix`	The string prepended to the title of every required item in this form if `highlightRequiredFields` is `true` and the `titleOrientation` property is set to `"right"`.	`"<B>:&nbsp;"`

Property	Description	Default
form.requiredTitleSuffix	The string appended to the title of every required item in this form if highlightRequiredFields is true.	"&nbsp;:</B>"
form.requiredRightTitleSuffix	The string appended to the title of every required item in this form if highlightRequiredFields is true and the titleOrientation property is set to "right".	"</B>"

DynamicForm styles

The properties listed in Table 6.8 set the cascading style sheet (CSS) class names used to specify styles of *DynamicForm* widgets. These properties name the CSS class in the skin_styles.css style sheet to use for a part of the form. Generally, you will not need to change these properties. To change a style, modify the CSS class description in the skin_styles.css file for the skin you're loading with your application.

Sometimes, however, you may want to create multiple forms with their own styles. In that case, you can set new CSS classes using the properties described in Table 6.8.

Warning You must create a new CSS class in skin_styles.css if you change the value of any of these properties. The new CSS class name must match the value of the corresponding *DynamicForm* style property.

Table 6.8: Form item style properties

Property	Description	Value
cellClassName	The CSS class applied to the item's form element if there are no errors for the item.	"formCell"
errorCellClassName	The CSS class applied to the item's form element if there is an error for the item.	"formError"
titleClassName	The CSS class applied to the item's title if there are no errors for the item.	"formTitle"
titleErrorClassName	The CSS class applied to the item's title if there is an error for the item.	"formTitleError"
hintClassName	The CSS class applied to the item's hint text.	"formHint"

Working with form item values

When you create a *DynamicForm* widget, you can specify an initial set of values for its items by setting the form's values property (as demonstrated by *"Example: Form items" on page 103*). You can also set default or dynamic values for each item with the item properties described in Table 6.9.

Table 6.9: Form item value properties

Property	Description
`defaultValue`	The default value for the item, if neither `form.values[itemName]` nor `item.defaultDynamicValue` is defined.
`defaultDynamicValue`	A string of script to evaluate for the item's value, if `form.values[itemName]` is not defined. In this script, you can use: • '`form`' as a reference to the dynamicForm widget, and • '`item`' as a reference to the form item object.

The value for each form item is therefore determined in the following order:

- `form.values`, if a value for the item is listed; otherwise,
- `item.defaultDynamicValue`, if defined; otherwise,
- `item.defaultValue`.

Warning

You cannot set a `defaultValue` or `defaultDynamicValue` for an `upload` field. If set, these properties are ignored. Redrawing dynamicForms with `upload` field items will cause the form to lose the value entered for the `upload` field.

The resulting value may furthermore be mapped to a different display value. The `radioGroup`, `select`, and `selectOther` items support display-value mapping via their `valueMap` property, as demonstrated in *"Example: Form items" on page 103*. Other form items (except for date items) support display-value mapping using the `valueMap` property detailed in Table 6.10.

Table 6.10: Form item display-value mapping property

Property	Description
`valueMap`	A property list specifying a mapping of internal values (the property names) to display or input values (the property values) for the item.

Form items automatically apply their `valueMap`, if defined, to:

- translate internal values into display values, and
- translate user-input values into internal values.

To access and manipulate the current values in the displayed form elements, use the getter and setter methods described in Table 6.11.

Table 6.11: Form value getter and setter methods

Property	Description
`item.getValue()`	Returns the current value of the form element associated with this item.
`form.getValue(itemName)`	Shorthand for `form.getItem(itemName).getValue()`.

Property	Description
`*item*.setValue(*value*)`	Sets the value of the form element associated with this item to *`value`*.
`*form*.setValue(*itemName*, *value*)`	Shorthand for `*form*.getItem(*itemName*).setValue(*value*)`.
`*form*.getValues()`	Returns `*form*.values`. **Note:** The `*form*.values` list is a direct pointer to the values. Manipulating these values and redrawing the form will update element values seen by the user.
`*form*.setValues([*values*])`	Sets `*form*.values` to *`values`*, if provided, then set the values of all form elements to the values of their items determined as discussed above from `values`, or from `defaultDynamicValue`, or from `defaultValue` in that order.This method also calls `*form*.rememberValues()` to store `*form*.values` before they are changed to *`values`*.
`*form*.rememberValues()`	Stores current `*form*.values`. A later call to `*form*.resetValues()` will reset `*form*.values` to the remembered values.
`*form*.resetValues()`	Resets `*form*.values` to last time `*form*.setValues()` or `*form*.rememberValues()` was called. If neither of those methods has been called, `*form*.resetValues()` will reset `*form*.values` using any `defaultDynamicValue` or `defaultValue` set, or to `null`, if no value is set for a field initially.
`*form*.valuesHaveChanged()`	Compares the current set of form values with the values stored by the call to the `*form*.rememberValues()` method. Returns `true` if the values have changed and `false` otherwise.

Validating form input

Before a form is submitted, the *DynamicForm* widget automatically validates the form's values according to criteria that you can specify for each form item. This client-side validation can catch many common input errors before data is sent to the server, thereby improving the responsiveness of your application, and simplifying your server-side programming.

In a typical form, the user clicks on a *submit* item (button) to attempt form submission. The *DynamicForm* widget automatically validates the form's values and either:

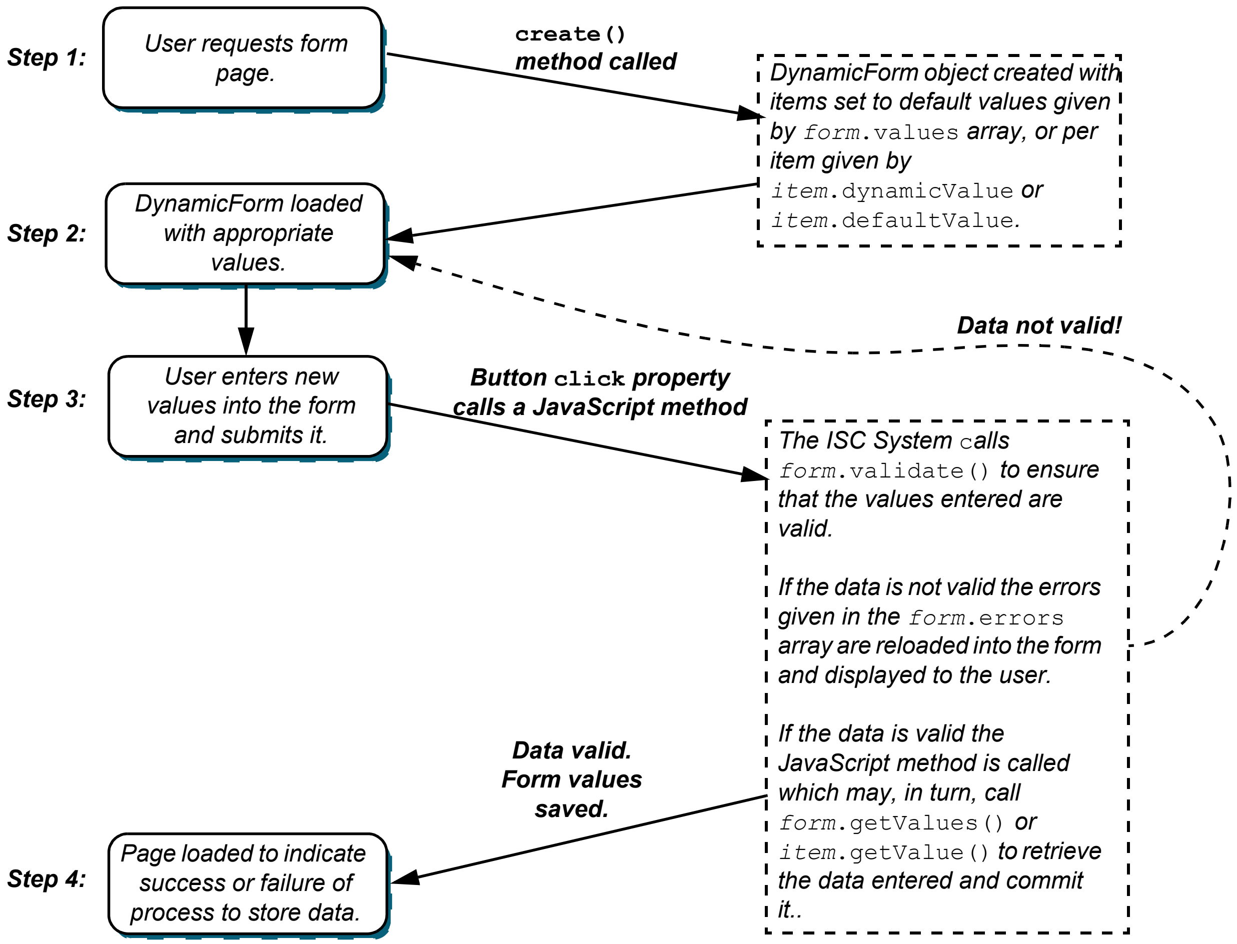
- submits the form to the specified action URL, using the specified method, if there are no validation errors, or
- redraws the form with error messages displayed for any items whose values caused validation errors.

You can also explicitly trigger form submission and/or validation in your scripts with the two form methods described in Table 6.12.

Table 6.12: Form submission and validation methods

Property	Description
`submitForm()`	Calls *`form`*`.validate()` to validate the form's values, and submits the form if there are no validation errors. Returns `true` if validation succeeds, or `false` if validation fails.
`validate()`	Validates the form without submitting it, and redraws the form to display error messages if there are any validation errors. Returns `true` if validation succeeds, or `false` if validation fails.

Figure 6.5 illustrates a common user interaction flow with a dynamic form.



Step 1:
User requests form page.
create() method called
DynamicForm object created with items set to default values given by form.values array, or per item given by item.dynamicValue or item.defaultValue.
Step 2:
DynamicForm loaded with appropriate values.
Data not valid!
Step 3:
User enters new values into the form and submits it.
Button click property calls a JavaScript method
The ISC System calls form.validate() to ensure that the values entered are valid.
If the data is not valid the errors given in the form.errors array are reloaded into the form and displayed to the user.
If the data is valid the JavaScript method is called which may, in turn, call form.getValues() or item.getValue() to retrieve the data entered and commit it..
Data valid. Form values saved.
Step 4:
Page loaded to indicate success or failure of process to store data.

Figure 6.5: Form input user interaction

The most common validation of form input is to ensure that a required field is not empty when the form is submitted. You can specify whether a form item is required using the `item` property listed in Table 6.13.

Table 6.13: Form item required property

Property	Description	Default
`required`	Indicates whether the item requires a non-empty value.	`false`

The title of a required item is drawn in bold text by default, to provide a visual cue to users filling out the form. See *"Form item annotations and styles" on page 108* for the properties that affect this behavior and style.

If a required field is empty when the form is validated, the following error message will be displayed for that item:

```
Required field itemTitle not entered.
```

All other types of input validation are specified by validators (objects that provide the criteria for specialized validation methods). You can specify one or more validators for each item in its `validators` property. For example, the following item has an '`isInteger`' validator, ensuring that the value of the '`Age`' field will be a whole number:

```
{name:"age", title:"Age",
    validators: {type:"isInteger", errorMessage:"Age must be a number."}
}
```

The `validators` property may be set to a single validator object (as above), or to an array of validator objects. Every validator object supports the properties described in Table 6.14.

Table 6.14: Form item validator properties

Property	Description
`type`	The type of the validator, determining which validation method will be executed. See Table 6.15 for a complete list.
`errorMessage`	The error message to display with an item if this validator fails (i.e., its validation method returns `false`). If `errorMessage` is not specified and this validator fails, the default error message specified by *`form`*`.unknownErrorMessage` will be displayed.
`stopIfFalse`	Indicates whether validation of the current item should terminate if this validator fails. If `stopIfFalse` is `true` and this validator fails, any validators following it in the validators array will be skipped. By default, all validators for an item are tested.
`suggestedValue`	A suggested value to return if the field fails this validator. Some validators automatically return suggested values (see Table 6.15). For example, the `integerRange` validator can be evaluated with respect to minimum and/or maximum values, set as `min` and/or `max` parameters. If a user submits a value greater than the `max` parameter, validation fails, and the `max` parameter value is returned as a suggested value. The specified error message is displayed, and the field value is set to the suggested value. Suggested values can also be set manually by including the `suggestedValue` parameter. This value will be returned if the field fails validation, regardless of any automatically generated suggested value.
`clientOnly`	Set this property to `true` to perform the validation check on the client only.

Each validator may also support other properties specific to its type. The *DynamicForm* class provides a default set of validators with the types and properties detailed in Table 6.15.

Table 6.15: DynamicForm validators

Type	Properties	Description
`requiredIf`	`expression`	Fails if `expression` is `true` and the item's value is empty.
`isBoolean`	None	Succeeds if the item's value is `true`, `false`, "`true`", or "`false`".
`isInteger`	None	Succeeds if the item's value is an integer.
`integerRange`	`min, max`	Succeeds if the item's value is an integer between `min` and `max`, inclusive. If the item's value is less than `min` or greater than `max`, it is set to a suggested value of `min` or `max` (but still fails and displays an error). Returns `true` for an empty value.
`lengthRange`	`min, max`	Succeeds if the string length of the item's value is between `min` and `max`, inclusive.
`matchesField`	`fieldName`	Succeeds if the item's value exactly matches the value of the item named `fieldName`. This validator is useful for matching password fields, etc.
`isOneOf`	`list`	Succeeds if the item's value exactly matches any item in the `list` array.
`contains`	`substring`	Succeeds if the item's value contains `substring`.
`doesntContain`	`substring`	Succeeds if the item's value does not contain `substring`.
`substringCount`	`substring, operator, count`	Succeeds if comparing the number of occurrences of `substring` in the item's value to the specified count, using the specified comparison operator (==, !=, <, <=, >, or >=), returns `true`. For example, the following validator succeeds if the item's value contains three or fewer commas. `{type:"substringCount", substring:",", operator:"<=", count:"3"}`
`regexp`	`expression`	Succeeds if the regular expression in `expression` matches a pattern in the item's value. Refer to a JavaScript language reference guide for more information on regular expressions.
`mask`	`mask, transformTo`	Succeeds if the regular expression in `mask` matches a pattern in the item's value; and replaces this pattern using the string or lambda expression in `transformTo`, if specified. Refer to documentation for the JavaScript `string.replace` method for more information.

Note

With the exception of '`requiredIf`', all of these validators will succeed if an item's value is empty. This allows you to specify the allowable type and/or format for the data if entered, without actually requiring data to be entered. To make a field mandatory, you must also set the item's `required` property or include a '`requiredIf`' validator for the item.

Custom validators

To create a custom validator, specify a `condition` property rather than the validator type. Set the value of the `condition` property to the JavaScript function or string that needs to be evaluated to determine if the data entered is valid. Strings are automatically evaluated by the system when the validator runs. If the string evaluates to `true`, the field value passes the custom validator.

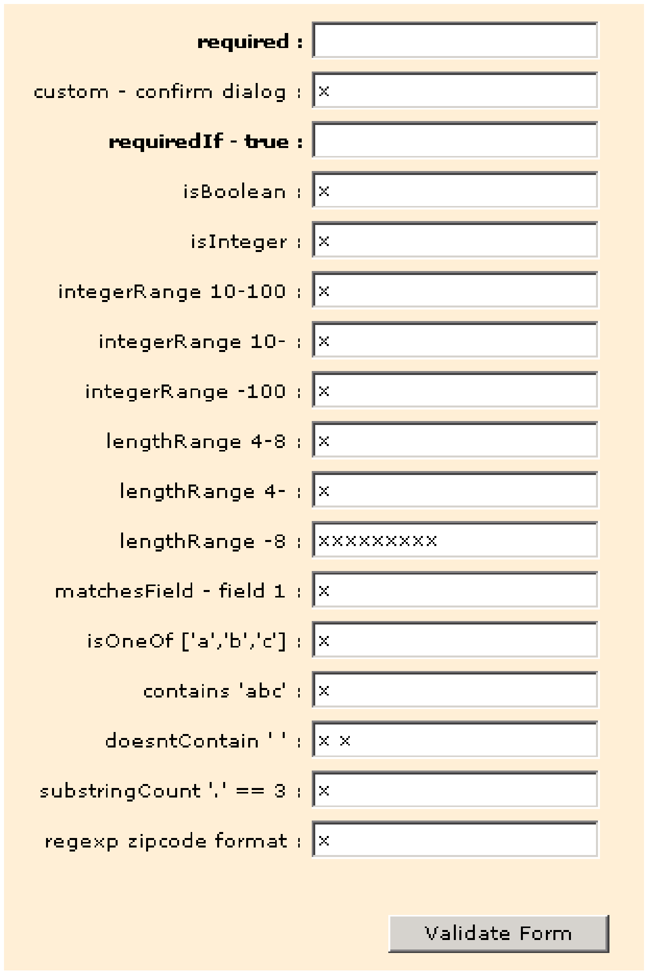
The following custom validator brings up a confirmation dialog box asking the user to accept or reject the value they entered for the field. If a user clicks **OK** in the dialog box, JavaScript returns `true` and the validator succeeds.

In this example, the `+` symbols are used to concatenate the current value of the field entered by the user into the 'Accept the value ' string literal and then adds the other ' for custom field?' string literal onto the end of the message. If the value entered into the form is `50`, the message that will appear in the dialog for this custom validator is:

Accept the value 50 for the custom field?

Example: Form item validation

The example file `dynamicForm_validators.html` (shown in Figure 6.6) provides a simple example for each of these validators.



required :	
custom - confirm dialog :	x
requiredIf - true :	
isBoolean :	x
isInteger :	x
integerRange 10-100 :	x
integerRange 10- :	x
integerRange -100 :	x
lengthRange 4-8 :	x
lengthRange 4- :	x
lengthRange -8 :	xxxxxxxxx
matchesField - field 1 :	x
isOneOf ['a','b','c'] :	x
contains 'abc' :	x
doesntContain ' ' :	x x
substringCount '.' == 3 :	x
regexp zipcode format :	x

Figure 6.6: Example of form item validation

If there are any errors during form validation, the error messages are saved in the `form.errors` property. After validation, the form is redrawn to display these errors. The form properties and methods described in Table 6.16 give you explicit control over the form error messages.

Table 6.16: Form error properties and methods

Property / Method	Description
`errors`	As given in Table 6.1, the value of the `errors` property can be set by server-side programming to reflect the results of server-side validation.
`unknownErrorMessage`	**Default:** `"Unspecified error"` The error message for a failed validator that does not specify its own `errorMessage`.
`setErrors(errors)`	Sets `form.errors` to `errors`. You must call `form.redraw()` to display the new error messages.

Property / Method	Description
`setError(itemName, error)`	Sets error message(s) for the specified *`itemName`* to the *`error`* string or array of strings. You must call *`form`*`.redraw()` to display the new error message(s).
`clearErrors()`	Clears all errors. You must call *`form`*`.redraw()` to clear any displayed error messages.

Handling form item events

When a user interacts with a form (e.g. typing text into fields, clicking buttons, selecting options in lists, etc.), the form generates events to which your scripts can respond. Form events are handled separately from the system events discussed in Chapter 3, *"Handling Events,"* but are nevertheless handled in a similar manner.

All data and button items provide *handler* properties, allowing you to execute scripts in response to form events. The available event handlers are listed in Table 6.17.

Table 6.17: Form item event handlers

Event Handler	Description
`focus`	Executed when a data item gets the input focus (i.e., a user clicks on or tabs to the item's element, or a script sets the input focus to the item's element).
`blur`	Executed when a data item loses the input focus.
`change`	Executed when the value of a data item's element is changed. For `text`, `password`, `upload`, or `textarea` items, `change` is only executed after the item's element loses the input focus.
`keyPress`	Executed when a key is pressed in a `text`, `password`, `upload`, or `textarea` item.
`click`	Executed when a `button`, `submit`, or `reset` item (either alone, or inside a *Toolbar* item) is clicked.

Like a system event handler, a form event handler can be specified either as a function to execute, or as a string of script to evaluate.

- Focus and blur handlers are passed a single parameter containing the current value of the element. In an evaluated handler, this parameter is named '`value`'.
- Change handlers are passed three parameters, `changedItemID`, `value`, and `oldValue`.
- The `keyPress` handlers are passed four parameters containing the current value of the element, the numerical code for the key that was pressed, the numerical code for the typed character, and the string representation of the typed character. In an evaluated handler, these parameters are named '`value`', '`keyNum`', '`charCode`', and '`charValue`'.

Form events may be used as triggers for implementing adaptive forms (i.e. forms that change their items, values, options, and other characteristics in response to partial input). For example, you might want a `select` item in a form to display a different set of options for each selected option in a group of radio buttons (in the same form). A `change` handler in the `radioGroup` item could look up or calculate a new set of options for the select item, and set them via the `setOptions` method, whenever the user selects a different radio button.

Adaptive form behavior can often be implemented using dynamic item properties (e.g. properties that can be specified as strings of script to evaluate on-the-fly). The following dynamic properties, discussed previously in this chapter, are all recalculated whenever the form is redrawn:

- `defaultDynamicValue`,
- `options`,
- `showIf`, and
- `requiredIf`.

If adaptive behavior is implemented using these properties, all the triggering event needs to do is redraw the form. This scenario is common enough that form data items provide a property to automatically redraw the form when an element's value is changed:

```
item.redrawOnChange
```

If *`item`*`.redrawOnChange` is `true`, the form is redrawn whenever the value of this item's element is changed. This redraw occurs after the item's change handler, if any, is executed.

Setting the `validateOnChange` property to `true` for an item causes any validation check(s) to be executed once the form item's contents have been changed and the form item has lost focus.

CHAPTER 7

ListGrids and DetailViewers

The Isomorphic SmartClient system provides two widget classes for displaying data in tables: *ListGrid* and *DetailViewer*. *ListGrids* display data horizontally. Field headings are given at the top of each column and all records are then displayed horizontally, one per row. *DetailViewer* widgets instead display data in vertical blocks. In this case, field headings are given at the beginning of each row, with the corresponding records displayed vertically, one per column. Typically, *DetailViewers* are used to display details about a selected set of records, whereas *ListGrids* are used to display summary lists of records returned by a query. This chapter covers how to create tables for data display within the ISC system, and how to sort and manipulate the data within these tables.

In this chapter:

Working with lists

If your applications involve any significant amount of data, you'll probably need to manipulate or present information in a tabular format at some point. Tabular data is organized into records and fields.

In JavaScript, tabular data is best expressed as an array of objects, where each object in the array represents a record and each named property of the objects represents a field. For example, the following array specifies four record objects with three field properties:

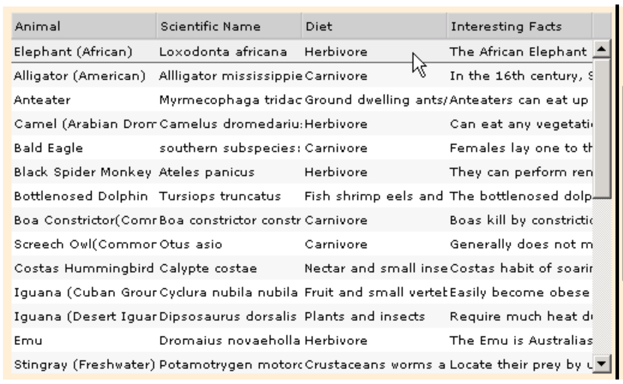
```
var fruits = [
   {type:"apple", color:"green", quantity:5},
   {type:"banana", color:"yellow", quantity:12},
   {type:"cherry", color:"red", quantity:50},
   {type:"durian", color:"yellow-green", quantity:3}
]
```

The ISC system provides several tools for working with tabular data in this format, including:

- array-object extensions for adding, removing, searching, and sorting rows,
- the *ListGrid* widget class for visually presenting tabular data horizontally as shown in Figure 7.1 below,
- the *DetailViewer* widget class for visually presenting tabular data vertically in blocks as shown in Figure 7.1 below, and
- the *Selection* class for maintaining and manipulating a selected set of rows.

This chapter focuses on the presentation and manipulation of tabular data using *ListGrid* and *DetailViewer* widgets. Many of the array-object extensions and *Selection* class features are discussed, but these objects are not covered exhaustively.

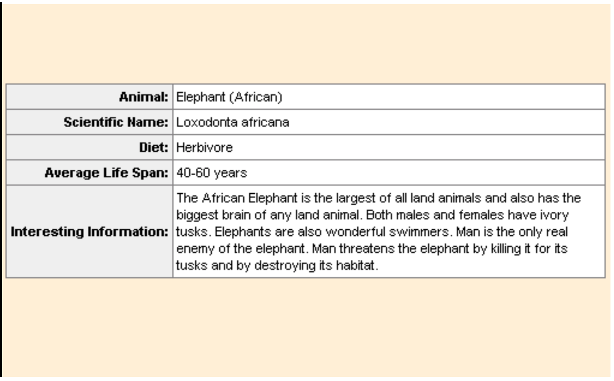
ListGrid



Animal	Scientific Name	Diet	Interesting Facts
Elephant (African)	Loxodonta africana	Herbivore	The African Elephant
Alligator (American)	Allligator mississippie	Carnivore	In the 16th century, S
Anteater	Myrmecophaga tridac	Ground dwelling ants/	Anteaters can eat up
Camel (Arabian Drom	Camelus dromedariu:	Herbivore	Can eat any vegetati
Bald Eagle	southern subspecies:	Carnivore	Females lay one to th
Black Spider Monkey	Ateles panicus	Herbivore	They can perform ren
Bottlenosed Dolphin	Tursiops truncatus	Fish shrimp eels and	The bottlenosed dolp
Boa Constrictor(Comr	Boa constrictor constr	Carnivore	Boas kill by constricti
Screech Owl(Commor	Otus asio	Carnivore	Generally does not m
Costas Hummingbird	Calypte costae	Nectar and small inse	Costas habit of soarir
Iguana (Cuban Grour	Cyclura nubila nubila	Fruit and small vertet	Easily become obese
Iguana (Desert Iguar	Dipsosaurus dorsalis	Plants and insects	Require much heat di
Emu	Dromaius novaeholla	Herbivore	The Emu is Australias
Stingray (Freshwater)	Potamotrygen motorc	Crustaceans worms a	Locate their prey by u

A ListGrid displays summary data in a list.

DetailViewer



Animal:	Elephant (African)
Scientific Name:	Loxodonta africana
Diet:	Herbivore
Average Life Span:	40-60 years
Interesting Information:	The African Elephant is the largest of all land animals and also has the biggest brain of any land animal. Both males and females have ivory tusks. Elephants are also wonderful swimmers. Man is the only real enemy of the elephant. Man threatens the elephant by killing it for its tusks and by destroying its habitat.

A DetailViewer displays record details vertically.

Figure 7.1: A ListGrid compared to a DetailViewer

Initializing a listGrid or detailViewer

There are three general steps to initializing a *ListGrid* or *DetailViewer* widget:

1. Specify record objects in the `data` array.

2. Specify field objects and their properties in the `fields` array.

3. Configure properties of the listGrid or detailViewer itself.

ListGrid and *DetailViewer* widgets have two fundamental properties that must be initialized. These properties are described in Table 7.1.

Table 7.1: ListGrid and DetailViewer fundamental properties

Property	Description	Default
viewer.data	An array of record objects, specifying data. In listGrids, the data array specifies rows.In detailViewers, the data array specifies columns. *ListGrids* automatically observe changes to the data array and update accordingly. See *"ListGrid styles" on page 129* for details. *DetailViewers* do not observe changes to the data array.	`null`
viewer.fields	An array of field objects, specifying the order, layout, dynamic calculation, and sorting behavior of each field in the listGrid object. With listGrids, the `fields` array specifies columns. With detailViewers, the fields array specifies rows.	`null`

Reference

Refer to Appendix A, "*Widget Initialization Templates*," for *ListGrid* and *DetailViewer* initialization templates that include all list, column, and row properties available for initialization. These properties include those documented elsewhere in this chapter and relevant properties inherited from the *Canvas* class.

Each record object in the data array specifies a set of *`fieldName:fieldValue`* pairs. The *`fieldNames`* correspond to a name value of a field object. In the following script, the data from the fruits example above is specified for a data array of a listGrid or detailViewer.

```
data:[
   {type:"apple", color:"green", quantity:5},
   {type:"banana", color:"yellow", quantity:12},
   {type:"cherry", color:"red", quantity:50},
   {type:"durian", color:"yellow-green", quantity:3}
]
```

The order of the fields (properties) in each record object is irrelevant, but the order of the record objects themselves in the data array is the order in which the records will be displayed. In a sortable listGrid, the order of records can be changed by user interaction. See *"Sorting listGrid records" on page 131* for details. *DetailViewer* widgets cannot be sorted.

ListGrid record objects may also specify two optional non-data configuration properties, listed in Table 7.2.

Table 7.2: ListGrid record object configuration properties

Property[1]	**Description**	**Default**
`*record*.enabled`	Affects the visual style and interactivity of the record. If `*record*.enabled` is `false`, the record (row in a listGrid) will not highlight when the mouse moves over it, nor will it respond to mouse clicks.	`true`
`*record*.isSeparator`	Defines a horizontal separator in the listGrid object. Typically this is specified as the only property of a record object, since a record with `isSeparator:true` will not display any values.	`false`

[1] Properties listed in this table are not available for detailViewers.

Each field object in a `fields` array has two fundamental properties, listed in Table 7.3.

Table 7.3: Fields array fundamental properties

Property	**Description**
field.name	The name of this field, corresponding to one of the property names used in the `data` array.
field.title	A title for this field, to display in the header of the listGrid or detailViewer object.

Continuing the fruits example above, a basic fields specification is:

```
fields:[
   {name:"type", title:"Fruit"},
   {name:"color", title:"Color"},
   {name:"quantity", title:"Quantity"}
]
```

The order of field objects in the `fields` array determines the order in which the fields will appear. In a listGrid, fields appear as columns. The order of fields in the `fields` array is the order in which columns will appear from left to right. In a detailViewer, fields appear as rows. The order of fields in the `fields` array is the order in which rows will appear from top to bottom within a block.

In this example, every record object property (type, color, quantity) in the `data` array has a corresponding field object in the `fields` array, and vice versa. This is typical, but not required. A record object may contain properties that are not displayed (and so have no corresponding field objects in the `fields` array), while a field object may generate dynamic values that are not associated with any record object in the `data` array. If a record does not have a named property value declared, an empty cell value will be displayed. See *"Working with listGrid values" on page 138* for details.

Record and field objects may have additional properties set to control other behaviors, as described in later sections of this chapter.

The `data` and `fields` properties of a listGrid or detailViewer may be set to different arrays after initialization using the setter methods given in Table 7.4.

Table 7.4: Data and fields setter methods

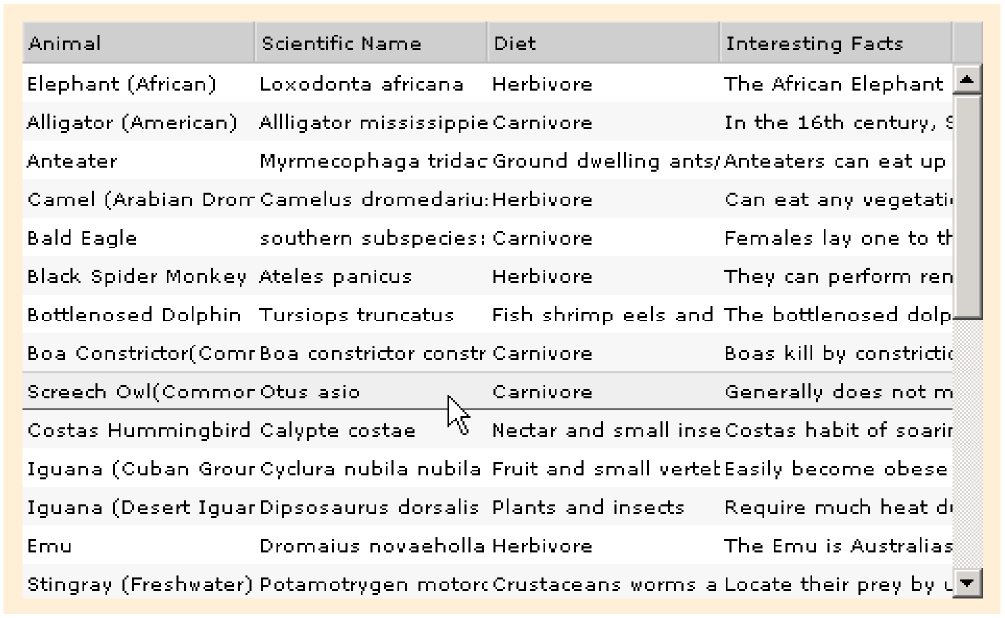
Method	Action
`setData(`*`recordList`*`)`	Set the `data` array to *`recordList`*.
`setFields([`*`fieldList`*`], [`*`fieldSizes`*`])`	Set the `fields` array and/or field widths to *`fieldList`* and *`fieldSizes`*, respectively. If a *`fieldSizes`* array (of numbers only) is not specified, the fields will be sized according to their individual size properties.

Tip

Since the `data` and `fields` properties often contain many values, you may want to set them to variables whose values are defined outside of the initialization block. This will improve the readability of your code, especially if you generate these properties with server-side programming. The following examples take this external-variable approach.

Example: ListGrid initialization

The example file `listGrid_init.html` (shown in Figure 7.2) creates and draws a list.



Animal	Scientific Name	Diet	Interesting Facts
Elephant (African)	Loxodonta africana	Herbivore	The African Elephant
Alligator (American)	Allligator mississippie	Carnivore	In the 16th century, S
Anteater	Myrmecophaga tridac	Ground dwelling ants/	Anteaters can eat up
Camel (Arabian Drom	Camelus dromedariu:	Herbivore	Can eat any vegetati
Bald Eagle	southern subspecies:	Carnivore	Females lay one to th
Black Spider Monkey	Ateles panicus	Herbivore	They can perform ren
Bottlenosed Dolphin	Tursiops truncatus	Fish shrimp eels and	The bottlenosed dolp
Boa Constrictor(Comr	Boa constrictor constr	Carnivore	Boas kill by constrictic
Screech Owl(Commor	Otus asio	Carnivore	Generally does not m
Costas Hummingbird	Calypte costae	Nectar and small inse	Costas habit of soarir
Iguana (Cuban Grour	Cyclura nubila nubila	Fruit and small vertel	Easily become obese
Iguana (Desert Iguar	Dipsosaurus dorsalis	Plants and insects	Require much heat d
Emu	Dromaius novaeholla	Herbivore	The Emu is Australias
Stingray (Freshwater)	Potamotrygen motorc	Crustaceans worms a	Locate their prey by u

Figure 7.2: Example of ListGrid initialization

The following script defines the data and fields for the sample list and creates the `listGrid` instance:

```
var animalData = [
   {commonName:'Elephant (African)',scientificName:'Loxodonta
      africana',diet:'Herbivore',lifeSpan:' 40-60 years',information:'The
      African Elephant is the largest of all land animals and also has the
      biggest brain of any land animal. Both males and females have ivory
      tusks. Elephants are also wonderful swimmers. Man is the only real
      enemy of the elephant. Man threatens the elephant by killing it for its
      tusks and by destroying its habitat.',status:'Threatened'},

   {commonName:'Alligator (American)',scientificName:'Alligator
      mississippiensis',diet:'Carnivore',lifeSpan:'50 years',information:'In
      the 16th century, Spanish explorers in what is now Florida encountered
      a large formidable animal which they called "el largo" meaning "the
      lizard". The name "el largo" gradually became pronounced
      "alligator".',status:'Not Endangered'},
   . . .
];

var animalFields = [
   {name:"commonName", title:"Animal"},
   {name:"scientificName", title:"Scientific Name"},
   // Note: Fields not included in this list will not show up in the
   // listGrid
   //{name:"lifeSpan", title:"Life Span"},
   //{name:"status",title:"Endangered Species Status"}
   {name:"diet", title:"Diet"},
   {name:"information", title:"Interesting Facts"}
],

ListGrid.create({
   ID:"animalList",
   data:animalData,
   fields:animalFields,
   canReorderRecords:true,
   left:50,
   top:75,
   width:500,
   height:300,
   alternateRecordStyles:true
});
```

This example implements the three initialization steps given at the beginning of this section as follows:

1. The listGrid's records (rows) are specified in the `animalData` array. This array contains five record objects, each containing information about an animal in three field properties ('`species`', '`color`', and '`behavior`').

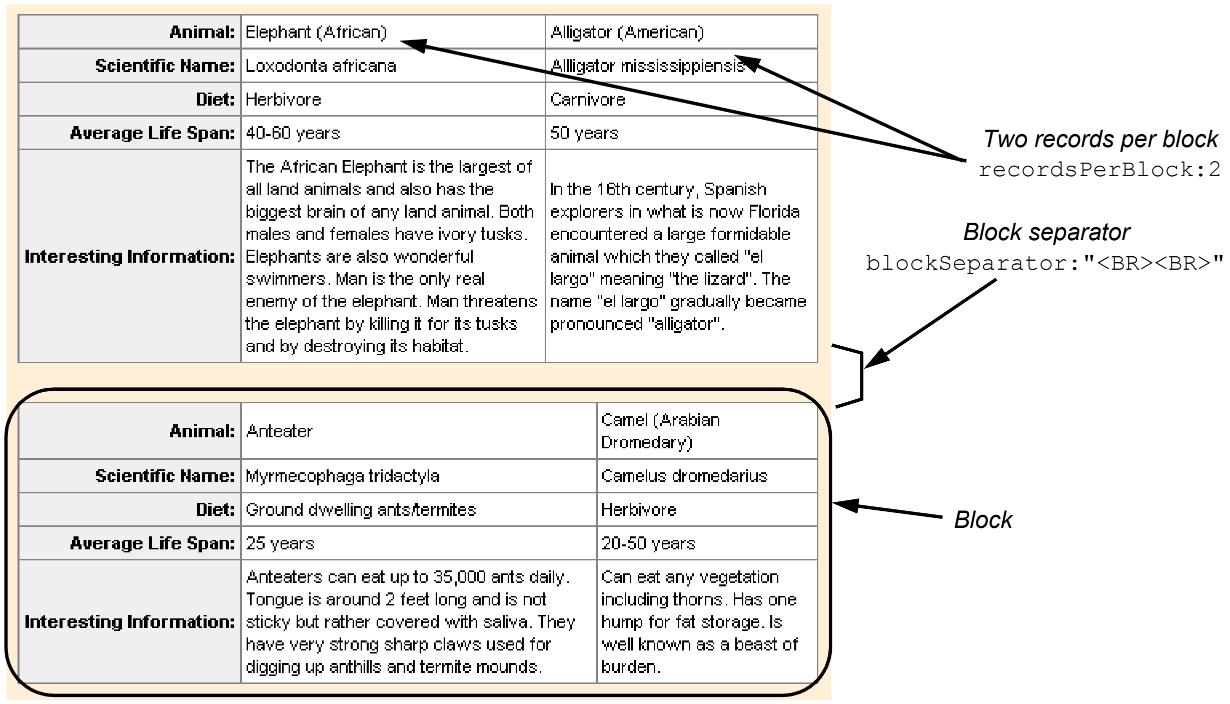
2. The listGrid's fields (columns) are configured in the `animalColumns` array. The `name` properties' values match the listGrid's `data` array field names.

3. The listGrid itself is configured in the call to the `create` method, setting its fundamental `data` and `fields` properties to the above arrays.

The width of the listGrid is divided equally amongst its columns. Click on the column headers to see the default listGrid sorting behavior. See *"Sorting listGrid records" on page 131* for information about sorting listGrid records. **Click**, **Shift-click**, and **Ctrl-click** on the listGrid rows to see the default listGrid selection behavior. See *"Selecting listGrid records" on page 133* for details on selecting listGrid rows.

Example: DetailViewer initialization

The example file `detailViewer_init.html` (shown in Figure 7.3) creates and draws a small detailViewer. This example uses the same data set as the listGrid example above.



Animal:	Elephant (African)	Alligator (American)
Scientific Name:	Loxodonta africana	Allligator mississippiensis
Diet:	Herbivore	Carnivore
Average Life Span:	40-60 years	50 years
Interesting Information:	The African Elephant is the largest of all land animals and also has the biggest brain of any land animal. Both males and females have ivory tusks. Elephants are also wonderful swimmers. Man is the only real enemy of the elephant. Man threatens the elephant by killing it for its tusks and by destroying its habitat.	In the 16th century, Spanish explorers in what is now Florida encountered a large formidable animal which they called "el largo" meaning "the lizard". The name "el largo" gradually became pronounced "alligator".

Animal:	Anteater	Camel (Arabian Dromedary)
Scientific Name:	Myrmecophaga tridactyla	Camelus dromedarius
Diet:	Ground dwelling ants/termites	Herbivore
Average Life Span:	25 years	20-50 years
Interesting Information:	Anteaters can eat up to 35,000 ants daily. Tongue is around 2 feet long and is not sticky but rather covered with saliva. They have very strong sharp claws used for digging up anthills and termite mounds.	Can eat any vegetation including thorns. Has one hump for fat storage. Is well known as a beast of burden.

Figure 7.3: Example of DetailViewer initialization

The following script defines the data and fields for the sample detailViewer and creates the `detailViewer` instance:

```
var animalData = [
   {commonName:'Elephant (African)',scientificName:'Loxodonta
      africana',diet:'Herbivore',lifeSpan:' 40-60 years',information:'The
      African Elephant is the largest of all land animals and also has the
      biggest brain of any land animal. Both males and females have ivory
      tusks. Elephants are also wonderful swimmers. Man is the only real
      enemy of the elephant. Man threatens the elephant by killing it for its
      tusks and by destroying its habitat.',status:'Threatened'},

   {commonName:'Alligator (American)',scientificName:'Alligator
      mississippiensis',diet:'Carnivore',lifeSpan:'50 years',information:'In
      the 16th century, Spanish explorers in what is now Florida encountered
      a large formidable animal which they called "el largo" meaning "the
      lizard". The name "el largo" gradually became pronounced
      "alligator".',status:'Not Endangered'},

   {commonName:'Anteater',scientificName:'Myrmecophaga
      tridactyla',diet:'Ground dwelling ants/termites',lifeSpan:'25
      years',information:'Anteaters can eat up to 35,000 ants daily. Tongue
      is around 2 feet long and is not sticky but rather covered with saliva.
      They have very strong sharp claws used for digging up anthills and
      termite mounds.',status:'Not Endangered'},
    {commonName:'Camel (Arabian Dromedary)',scientificName:'Camelus
       dromedarius',diet:'Herbivore',lifeSpan:'20-50 years',information:'Can
       eat any vegetation including thorns. Has one hump for fat storage. Is
       well known as a beast of burden.',status:'Not Endangered'}
],

animalFields = [
    {name:"commonName", type:"text", title:"Animal"},
    {name:"scientificName", type:"text", title:"Scientific Name"},
    {name:"diet", type:"text", title:"Diet"},
    {name:"lifeSpan", type:"text", title:"Average Life Span"},
    {name:"information", type:"text", title:"Interesting Information"}
],

DetailViewer.create({
    ID:"animalDetail",
    data:animalData,
    fields:animalFields,
    left:20,
    top:45,
    width:500
});
```

The only difference in the basic initialization of a detailViewer and a listGrid is the widget class specified in the `create` initializer. Beyond basic initialization, numerous properties can be set to affect the appearance and other behaviors of listGrids and detailViewers. These properties are covered in *"Configuring listGrid layout and appearance"* and *"Configuring detailViewer layout and appearance,"* respectively, for each widget class.

Configuring listGrid layout and appearance

ListGrid widgets and field objects provide a range of properties that affect a listGrid's appearance, including the listGrid's layout and styles.

The *ListGrid* and field object properties presented in Table 7.5 affect the layout of a listGrid.

Table 7.5: ListGrid layout properties

Property	Description	Default
field.width	The width of this field (column), specified as either an absolute number of pixels, a percentage of the remaining space, or '`*`' , the default value, to allocate an equal portion of the remaining space. Field widths may also be set by passing an array of numbers (absolute pixel sizes only) as the second parameter to the *listGrid*.`setFields` method. See Table 7.4.	`"*"`
field.align	The horizontal alignment of values in this field's column: • `left`, • `center`, or • `right`.	`null` (left aligned)

Property	Description	Default
field.cellAlign	The alignment of the header text and cell values within a cell: • `left`, • `center`, or • `right`.	`null` (left aligned)
listGrid.rowHeight	The height of each record's row in this listGrid, in pixels.	`20`
listGrid.cellSpacing	The amount of empty space, in pixels, between each cell in this listGrid.	`0`
listGrid.cellPadding	The amount of empty space, in pixels, surrounding each listGrid value in its cell.	`2`
listGrid.headerHeight	The height of this listGrid's header, in pixels.	`20`

The *ListGrid* properties described in Table 7.6 affect the appearance of a listGrid.

Table 7.6: ListGrid appearance properties

Property	Description	Default
field.showIf	A string of script that, if provided, is evaluated to conditionally determine the value of this field's `visible` property when the listGrid is drawn or redrawn or a direct call has been made to the `setFields` method.	`null`
listGrid.showEmptyMessage	Indicates whether the text of the `emptyMessage` property should be displayed if the listGrid has an empty `data` array—in other words, no records to show.	`true`
listGrid.emptyMessage	The string to display in the body of a listGrid with an empty `data` array, if `showEmptyMessage` is `true`.	`"No items to show."`
listGrid.alternateRecordStyles	Indicates whether listGrid records should be drawn in alternating styles, typically to apply different background colors, for easier reading across records.	`false`
listGrid.alternateRecordFrequency	The number of consecutive records to draw in the same style before alternating, when `alternateRowStyles` is `true`.	`1`

ListGrid styles

The properties listed in Table 7.7 set the CSS class names used to specify styles of *ListGrid* widgets. These properties name the CSS class in the `skin_styles.css` cascading style sheet to use for a part of the listGrid. Generally, you will not need to change these properties. To change a style, modify the CSS class description in `skin_styles.css` for the skin being loaded with your application. See *"Using and customizing ISC skins" on page 75* for more information on skins.

Sometimes, however, you may want to create multiple listGrids with their own styles. In that case, you can set new CSS classes using the properties described in Table 7.7.

Warning

You must create a new CSS class in the `skin_styles.css` for the skin you're using if you change the value of any of these properties. The new CSS class name must furthermore match the value of the corresponding listGrid style property.

Table 7.7: ListGrid style properties

Property	Description	Default
`baseStyle`	The base name for the CSS class applied to cells in the listGrid. This style is appended with `alternateColorStyleSuffix` and/or `"Over"`, `"Selected"`, or `"Disabled"` as appropriate.	`"cell"`
`emptyMessageStyle`	The CSS class applied to the `emptyMessage` string if displayed.	`"normal"`

Reference

All listGrid style properties are presented in Appendix B, "*Isomorphic SmartClient Styles*."

Configuring detailViewer layout and appearance

DetailViewer widgets provide properties that affect appearance, including the layout and styles. The properties presented in Table 7.8 control the layout of a detailViewer.

Table 7.8: DetailViewer layout properties

Property	Description	Default
`recordsPerBlock`	The number of records to display in a block. A block is a horizontal row on a page containing one or more records, as specified by the value of `recordsPerBlock`. The height of a block is equal to the height of a single record. The default setting of `1` causes each record to appear by itself in a vertical row. Setting `recordsPerBlock == 2` would cause records to appear side by side in groups of two.	`1`
`blockSeparator`	A string (HTML acceptable) that will be written to a page to separate blocks.	`"<BR><BR>"`
`cellPadding`	The amount of empty space, in pixels, surrounding each detailViewer value in its cell.	`2`

The *DetailViewer* property described in Table 7.9 controls the appearance of a detailViewer when there are no records in the data array to display.

Table 7.9: DetailViewer appearance properties

Property	Description	Default
`emptyMessage`	The string to display in the body of a detailViewer with no records.	`"No items to display."`

Adding and removing listGrid records

ListGrid widgets automatically "observe" their `data` arrays for changes caused by the following extended array methods:

- `add(`*`record`*`)`
- `addList(`*`recordList`*`)`
- `addAt(`*`record, position`*`)`
- `addListAt(`*`recordList, position`*`)`
- `removeAt(`*`position`*`)`
- `remove(`*`record`*`)`
- `removeList(`*`recordList`*`)`

Note

A listGrid will update automatically if entire records are added, removed, or replaced using the methods given above. It will not update automatically if individual values of existing records are changed. To update a listGrid after changing values in existing records, call *`listGrid`*`.markForRedraw()` to redraw the widget. See *"Drawing" on page 28* for more information on drawing-related methods.

To ensure that your listGrids update automatically when their `data` arrays change, you should use these methods instead of the standard JavaScript array methods—`concat`, `push`, `pop`, `shift`, `unshift`, `splice`, etc.—to add or remove records in the `data` array. *DetailViewer* widgets do not observe their `data` arrays for changes caused by the above methods.

Sorting listGrid records

ListGrid widgets provide interactive sorting behavior by default. Clicking on a field's column header in a listGrid sorts the listGrid by that field's values. Clicking again on the header of a sorted column reverses the direction of sorting. Columns in a listGrid are specified by field objects in the `fields` array. *DetailViewer* widgets do not provide sorting behavior.

You can customize a listGrid's sorting behavior with the listGrid and `field` properties presented in Table 7.10.

Table 7.10: ListGrid sorting properties

Property	Description	Default
listGrid.canSort	Enables or disables interactive sorting behavior for this listGrid. Does not affect sorting by direct calls to the `sort` method described below.	`true`
listGrid.sortFieldNum	Specifies the number of the field by which this listGrid is currently sorted. Column numbers start at `0` for the left-most column.	`null`
listGrid.sortDirection	Specifies the current sorting direction of this listGrid. If set to `ascending`, the lowest value in the sorted column is at the top of the listGrid. If set to `descending`, the highest value in the sorted column is at the top of the listGrid.	`"ascending"`
field.canSort	Enables or disables sorting by this column. If `false`, neither interactive nor scripted (via the `sort` method) instructions will sort the listGrid by this column.	`true`
field.sortDirection	Specifies the default sorting direction for this column. If set to `ascending`, the lowest value in the column is at the top of the listGrid. If set to `descending`, the highest value in the column is at the top of the listGrid. If not specified, the current value of *listGrid*.sortDirection is used instead.	`null`
field.sortNormalizer	An optional function to normalize listGrid values for sorting. If provided, this function should take (*recordObject*, *fieldName*) parameters and return a normalized value to use for sorting comparisons.	`null`
listGrid.showSortArrow	Indicates whether a sorting arrow should appear for the listGrid, and its location. Acceptable values are: • `none`—No sort arrow. • `corner`—Sort arrow appears in the top right corner of the listGrid. • `field`—Sort arrow appears in the column header of the column specified by *listGrid*.sortCol. • `both`—Sort arrows appear as if both corner and column are set. Clicking the sort arrow reverses the direction of sorting for the current sort column (if any), or sorts the listGrid by its first sortable column. The arrow image on the button indicates the current direction of sorting.	`"both"`

You can also sort a listGrid explicitly by calling its `sort` method:

```
listGrid.sort([sortCol],[sortDirection])
```

In this code sample, *sortCol* and *sortDirection* are optional parameters specifying new values for *listGrid*.sortCol and *listGrid*.sortDirection. If neither *sortCol* nor *listGrid*.sortCol is defined, *listGrid*.sortCol will be set to the first sortable column in this listGrid.

Selecting listGrid records

When a listGrid is created, it automatically creates a *Selection* object to maintain and manipulate its set of selected records. *ListGrid* records appear as rows. A listGrid's clickable-selection behavior is defined by the `selectionType` property:

```
listGrid.selectionType
```

The `selectionType` property can be set to one of four possible constants in the *Selection* class, given in Table 7.11.

Table 7.11: ListGrid selectionType values

Value	Behavior
`"none"`	Clicking on a record (row) has no effect.
`"single"`	Clicking on an enabled record selects that record and deselects any other record.
`"simple"`	Clicking on an enabled record toggles the selection state of that record, without affecting other records in the selection.
`"multiple"`	**Default:** Clicking on an enabled record with no modifier key pressed selects that record only, as with the `single` style above. If the **Alt**, **Ctrl**, or **Meta** key is pressed, the selection of state of the record is toggled, as with the `simple` style above. If the **Shift** key is pressed, the selection expands or contracts to include a continuous range of records. This range is bounded... • from the first record of the current selection to the clicked record, if the clicked record is after the first record in the current selection, or, • from the clicked record to the last record in the current selection, if the clicked record is before the first record in the current selection.

Clicking on a disabled record (i.e. one in which `record.enabled` is `false`) does not change the selection, regardless of the selection type.

The *Selection* object associated with a listGrid can be accessed with the listGrid's selection property:

```
listGrid.selection
```

This object provides a number of methods that manipulate the current set of selected records. These methods are described below.

Table 7.12: Selection methods available on ListGrid.selection

Method	Action
`getSelection()`	Returns an array of selected record objects.
`getSelectedRecord()`	Returns the first selected record object.
`anySelected()`	Returns `true` if any record in the listGrid is selected; and `false` otherwise.
`multipleSelected()`	Returns `true` if more than one record is selected, `false` otherwise.

Method	Action
`isSelected(record)`	Returns `true` if the specified record object is selected; and `false` otherwise.
`select(record)`	Selects the specified record object.
`deselect(record)`	Deselects the specified record object.
`selectSingle(record)`	Selects the specified record object and deselects any other records that were selected.
`selectList(recordList)`	Selects all record objects in the *`recordList`* array.
`deselectList(recordList)`	Deselects all record objects in the *`recordList`* array.
`selectAll()`	Selects all records in the listGrid.
`deselectAll()`	Deselects all records in the listGrid.
`selectItem(recordNum)`	Selects the record in the listGrid at index *`recordNum`* (starting with `0`).
`deselectItem(recordNum)`	Deselects the record in the listGrid at index *`recordNum`* (starting with `0`).
`selectRange(fromRecordNum, toRecordNum)`	Selects all records in the listGrid from index *`fromRecordNum`* to index *`toRecordNum`*, not including the record at index *`toRecordNum`*.
`deselectRange(fromRecordNum, toRecordNum)`	Deselects all records in the listGrid from index *`fromRecordNum`* to index *`toRecordNum`*, not including the record at index *`toRecordNum`*.

ListGrids automatically update their visible selections to reflect any changes made by calls to the methods above. For example, the following script selects (and therefore highlights) all records in a listGrid:

```
listGrid.selection.selectAll()
```

The array returned by the `getSelection` method may be manipulated by any of the extended array methods (see *"ListGrid styles" on page 129*), or passed as a parameter wherever an array is appropriate. For example, this script removes all selected records from a listGrid:

```
listGrid.data.removeList(listGrid.selection.getSelection())
```

Dragging and dropping listGrid records

ListGrid widgets support two kinds of drag-and-drop behavior:

- drag-and-drop reordering of records within a listGrid, and
- drag-and-drop of records between different listGrids.

ListGrid-record drag-and-drop is initiated by clicking on a record and dragging while the mouse button is held down. If dragging is enabled for the listGrid as a whole, and for each record in the current selection, all records in the current selection will be dragged. While these listGrid records are being dragged, a small "list" icon is attached to the mouse cursor. If this icon appears as a black square, the image can't be found. If the icon is not displayed at all, no records are being dragged. The selected records will be dragged only if:

- the listGrid's `canReorderRecords` or `canDragRecordsOut` property is `true`,
- no record in the current selection has a `canDrag` property set to `false`, or
- dragging is initiated from an enabled (`enabled != false`), non-separator (`isSeparator != true`) record.

As the cursor moves over a listGrid that can accept dragged records, an insertion line is displayed to indicate the position at which the records will be inserted if they are dropped. If this line does not appear, the records cannot be dropped.

The *ListGrid* properties listed in Table 7.13 control the overall drag-and-drop behavior of a listGrid.

Table 7.13: ListGrid drag-and-drop properties

Property	**Description**	**Default**
`canReorderRecords`	Indicates whether records can be reordered by dragging within this listGrid.	`false`
`canDragRecordsOut`	Indicates whether records can be dragged from this listGrid to other listGrids.	`false`
`canAcceptDroppedRecords`	Indicates whether this listGrid will accept records dragged and dropped from other listGrids.	`false`

For more control, each record (row) in a listGrid also supports two drag-and-drop properties, described in Table 7.14.

Table 7.14: ListGrid record drag-and-drop properties

Property	**Description**	**Default**
`record``.canDrag`	If `false`, this *`record`* cannot be dragged. If `canDrag` is `false` for any record in the current selection, none of the records will be dragged.	`null`
`record``.canAcceptDrop`	If `false`, other records cannot be dropped on (i.e., inserted immediately before) this *`record`*.	`null`

Editing listGrid fields

ListGrid widgets support in-line editing of values for individual fields in a record. If enabled, editing is initiated by double-clicking on a field. This changes the static field to a form field to allow user input according to the field's type—text fields will change to text input boxes, and selection fields to drop-down menus. (Other form item types are given inTable 6.2 on page 100.)

When the user completes editing and presses the **Enter** key, or clicks outside of the input area, validation on the field is performed. If validation succeeds, the new value is shown for that field. If not, the original value is restored. In both cases the editing mode ceases to show the field as a static value until it receives another '`double-click`' event.

Table 7.15: ListGrid in-line editor properties

Property	Description	Default
`canEdit`	If set to `true`, users will be able to interactively edit the values for fields of this listGrid in-line. Editing can be turned off for any individual fields by setting `canEdit` to `false` as a field property.	`null`

In-line editing is turned on by setting the `canEdit` property for the listGrid to `true`. This allows all fields to be edited by default. You can turn off editing for individual fields by setting the `canEdit` property to `false` on the field level.

Warning

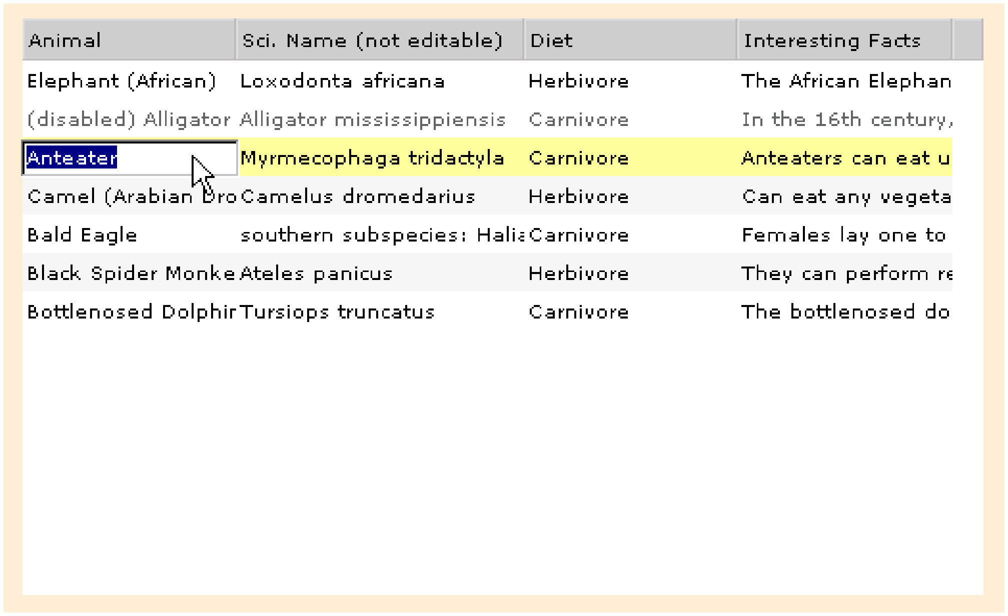
ListGrid editing requires functionality provided by the *DynamicForm* widget. Therefore, if your license with Isomorphic Software does not include the *DynamicForm* package, you will not be able to use editable *ListGrid* capabilities.

Table 7.16: ListGrid field in-line editor properties

Property	Description	Default
`field``.canEdit`	If the `canEdit` property is set to `true` for the listGrid as a whole, setting this property to `false` for the field will prevent it from being edited in-line. If the `canEdit` property is not set or is set to `false` for the listGrid, this property will be ignored for the field.	`null`
`field``.editorType`	FormItem type to use to edit this field.	`[varies by field.type]`

Example: Editable ListGrid initialization

The example file `editableListGrid_init.html` (shown in Figure 7.4) creates and draws a listGrid exactly like the one shown previously, only this version supports in-line editing. This example uses a subset of the data for listGrid example above.



Animal	Sci. Name (not editable)	Diet	Interesting Facts
Elephant (African)	Loxodonta africana	Herbivore	The African Elephan
(disabled) Alligator	Alligator mississippiensis	Carnivore	In the 16th century,
Anteater	Myrmecophaga tridactyla	Carnivore	Anteaters can eat u
Camel (Arabian Dro	Camelus dromedarius	Herbivore	Can eat any vegeta
Bald Eagle	southern subspecies: Halia	Carnivore	Females lay one to
Black Spider Monke	Ateles panicus	Herbivore	They can perform re
Bottlenosed Dolphir	Tursiops truncatus	Carnivore	The bottlenosed do

Figure 7.4: Example of editable ListGrid initialization

The following script defines the fields for the editable list and creates the `listGrid` instance (the changes from the last example are shown in bold):

```
// Data to be displayed
. . .

var animalFields = [
    {name:"commonName", title:"Animal"},
    {name:"scientificName", title:"Sci. Name (not editable)", canEdit:false,
        width:150},
    {name:"diet", title:"Diet", valueMap:{O:"Omnivore", C:"Carnivore",
        H:"Herbivore",I:"Insectivore",P:"Pescivore"}},
    {name:"information", title:"Interesting Facts", editorType:"textArea"}
];

ListGrid.newInstance({
    ID:"animalList",
    canEdit:true,
    data:animalData,
    fields:animalFields,
    canReorderRecords:true,
    left:50,
    top:75,
    width:500,
    height:300,
    alternateRecordStyles:true
});
```

The big change in this example is that the `canEdit` property has been set to `true` for the listGrid. This makes all text fields editable. Field properties are used to override the values given for the listGrid as a whole.

- The `scientificName` field sets `canEdit` to `false` to prevent it from being overwritten.
- The `diet` field uses a valueMap so that the editing is done through a drop-down menu of only valid choices.
- The `information` field is a long text area field, so it overrides the default editor with the `"textArea"` value.

Working with listGrid values

The values in a listGrid come from the record objects in the listGrid's `data` array, but you can process these values or even generate entirely new values for display. The value displayed in any cell in a listGrid is determined by the following sequence:

1. The raw value for the cell is taken from the appropriate field (property) of the appropriate record object in the listGrid's `data` array.

2. If `*field*.getCellValue` is defined for the cell's column, it is an expression evaluated or function called with the raw value whereby the result is used as the cell's value.

3. If `*field*.valueMap` is defined, the cell's value is used as a key to look up a different display value in the map.

4. If the cell's display value is empty and *`field`*`.emptyCellValue` is defined, that value is displayed.

5. If the cell's display value is empty and *`field`*`.emptyCellValue` is not defined, *`listGrid`*`.emptyCellValue` is displayed.

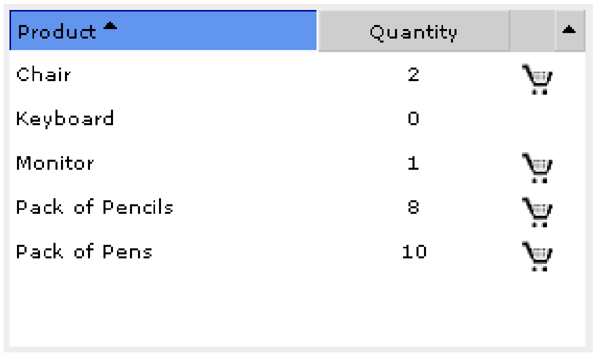
The three field object properties used in this sequence are explained in Table 7.17.

Table 7.17: ListGrid field value properties

Property	Description
`formatCellValue`	An expression or function evaluated/called to generate the cell values for the field (column). If defined as an expression, the following variables are available: • `value`–value stored in the record for this field • `record`—the current record object • `rowNum`—the row number in the current set of displayed records (e.g., 0 for the first displayed record) • `colNum`—the column number in the current set of displayed columns • grid-the ListGrid object If defined as a function, the order of the parameters is as given above, eg `formatCellValue(`*`value, record, rowNum, colNum, grid`*`)` Since `formatCellValue` is a field property, you can use the `this` keyword in the function body to refer to the current field object if necessary.
`valueMap`	A property list (or an expression that evaluates to a property list) specifying a mapping of internal values to display values for the field (column).
`emptyCellValue`	**Default:** `"&nbsp;"` The value to display for a cell whose value is `null` or the empty string (`""`) even after applying `getCellValue` and `valueMap`.

The above sequence is executed for each cell whenever a listGrid is drawn.

To illustrate the use of the field value properties given in Table 7.17, consider a simple office supply catalog where a listGrid shows three fields: the office supply product, the quantity in stock, and a shopping cart icon as shown in Figure 7.5.



Product	Quantity	
Chair	2	🛒
Keyboard	0	
Monitor	1	🛒
Pack of Pencils	8	🛒
Pack of Pens	10	🛒

Figure 7.5: Example of listGrid field value properties

```
function showCartValue(value) {
    if (value > 0)
        return "yes";
    else
        return "no";
}

function addToCart(record) {
    if (record.qty > 0) {
        alert(record.itemName + " added to cart.");
        record.qty = record.qty - 1;
        itemList.redraw();
    }
}

ListGrid.create({
    ID:"itemList",
    height:200,
    left:5,
    top:5,
    width:300,
    cellHeight:23,
    selectionType:"single",
    sortFieldNum:0,
    cellPadding:3,
    data:[
        {itemName:"Pack of Pens", qty:10, enabled:true},
        {itemName:"Pack of Pencils", qty:8, enabled:true},
        {itemName:"Chair", qty:2, enabled:true},
        {itemName:"Keyboard", qty:0, enabled:true},
        {itemName:"Monitor", qty:1, enabled:true}
    ],
    fields:[
        {name:"itemName", title:"Product", sortDirection:"descending",
            width:"100%"},
        {name:"qty", title:"Quantity", canSort:false, align:"center",
            width:"100"},
        {name:"showCart", title:"", canSort:false, width:"23", align:"center",
            formatCellValue:"showCartValue(record.qty)",
            valueMap:{"yes":"<img src='cart.gif' height=22 width=22>", "no":""},
            recordClick:"addToCart(record)"
        }
    ]
});
```

The valueMap maps the string `"yes"` to the shopping cart image, `cart.gif`, and the `getCellValue` property is used to determine if the cart should be shown or not. As given by the `showCartValue` function, if the product quantity is zero, the cart is not shown. Note that because the value `"no"` is mapped to an empty string, the cell will pick up the `emptyCellValue` property and display nothing.

When the user clicks on the cart field, the `addToCart` function is called. This function shows a JavaScript alert, and decreases the product quantity by one in the list. If the quantity is already zero, then it does nothing. When an item decreases to a quantity of zero, the cart disappears. See the following section on *"Handling listGrid record events"* for more information.

Handling listGrid record events

ListGrid widgets handle all of the standard mouse and drag-and-drop events to implement highlighting, selection, sorting, and other behaviors. SeeTable 3.6 on page 52 for a complete list of mouse events. You should not set your own handlers in a listGrid for these events, as doing so would disable the listGrid's built-in behavior.

ListGrid record events

However, *ListGrid* widgets generate two additional events for which you can set handlers:

- `recordClick`—executed when the listGrid receives a '`click`' event on an enabled, non-separator record, and
- `recordDoubleClick`—executed when the listGrid receives a '`doubleClick`' event on an enabled, non-separator record.

Event handling of record events

These listGrid-record events are handled separately from the system events discussed in Chapter 3, *"Handling Events."* They are not handled globally, nor "bubbled up" through parent widgets. Instead, they are handled by:

- field-level handlers (*`field`*`.recordClick` and *`field`*`.recordDoubleClick`), if defined for the field (column) in which the event occurred, and otherwise,
- listGrid-level handlers (*`listGrid`*`.recordClick` and *`listGrid`*`.recordDoubleClick`), if defined.

Event handler variables

In either case, a record event handler can be specified either as a function to execute, or as a string of script to evaluate. If the handler is defined as a string of script, the following variables may be used in the script:

- *`record`*—the current record object,
- *`recordNum`*—the number of the record in the current set of displayed records (e.g., `0` for the first displayed record),
- *`fieldNum`*—the number of the field in the *`listGrid`*`.fields` array,
- *`value`*—the display value of the cell, as returned from *`listGrid`*`.getCellValue` method, and
- *`rawValue`*—the raw value of the cell, from the appropriate field in the current record object.
- *viewer*-the ListGrid itself

If the handler is defined as a function, most of these variables are passed as parameters to the `recordClick` event in the following order when the function is called:

```
recordClick(viewer, record, recordNum, fieldNum, value, rawValue)
```

CHAPTER 8

TreeGrids

Some types of information are best organized and browsed hierarchically. For example, files on your computer are organized into a hierarchy of folders or directories for easier management. The Isomorphic SmartClient system supports hierarchical data—also referred to as tree data due to its "branching" organization—with:

- the *Tree* class, which manipulates hierarchical data sets, and
- the *TreeGrid* widget class, which extends the *ListGrid* class to visually present tree data in an expandable/collapsible format.

This chapter focuses on the presentation and manipulation of hierarchical data using *TreeGrid* widgets (and the specialized *TreeGrid* subclasses).

In this chapter:

Specifying tree data

Tree data, like tabular data, is expressed as a collection of JavaScript objects. An object in a tree is called a *node*, and is one of two types:

- a folder node, which contains other nodes, or
- a leaf node, which does not contain other nodes.

Each node object has three fundamental properties, as described in Table 8.1.

Table 8.1: TreeGrid node fundamental properties

Property	Description
`name`	The name of the node, used internally to construct its path in the tree, and used as the node's display name if the node does not define a title. The name of a folder node must end in the path delimiter (`"/"` by default) to distinguish them from leaves.
`title`	The display name of the node. If title is not specified, the node's name will be stripped of any trailing path delimiter and used as the display name instead.
`children`	An array of other node objects (folders and/or leaves) that are contained by this node. Only folder nodes may specify a children property.

A tree is defined by a single folder node, called the *root*, that contains all other nodes.

Data in a treeGrid is sorted initially using the same properties that apply to *ListGrid* widgets. *TreeGrid* sorting is case insensitive.

Reference

Since the *TreeGrid* extends the *ListGrid* class, you should read Chapter 7, "*ListGrids and DetailViewers,*" to familiarize yourself with *ListGrid* features before working with treeGrids.

The *Tree* class, a subclass of *TreeGrid*, is discussed in this chapter, but is not covered exhaustively. For more information on the methods provided by the *Tree* class, see Appendix D, "*Tree Methods*."

Example: Tree data

The following list represents possible items in a small inventory of a zoo:

- Bottlenose Dolphin
- Giant Pacific Octopus
- Freshwater Stingray
- Cuban Ground Iguana
- Desert Iguana
- Marbled Salamander
- Indian Rock Python
- Howler Monkey
- Orangutan
- Guinea Baboon
- Lion

Even with a list this small, hierarchical organization can provide a better "big picture" of what's available, making it easier to search for a specific thing. In this example, animals are sorted into the various areas of the zoo where they live:

```
Aquarium
   Saltwater
      Bottlenose Dolphin
      Giant Pacific Octopus
   Freshwater
      Freshwater Stingray
Reptile House
   Lizard House
      Cuban Ground Iguana
      Desert Iguana
      Marbled Salamander
   Snake House
      Indian Rock Python
Monkey House
   Howler Monkey
   Orangutan
   Guinea Baboon
Lion Enclosure
   Lion
```

Each animal from the original list is a leaf node in this tree, grouped under one of the new folder nodes. The topmost folder node (`Zoo`) is the root of the tree. We can represent this tree in JavaScript as follows:

```
Tree.create({
   ID:"animalTree",
   root: {name:"Zoo/", children:[

      {name:"Aquarium/", children:[
         {name:"Salt Water/", children:[
            {name:"Bottlenose Dolphin", quantity:5, scientificName:'Tursiops
               truncatus'},
            {name:"Giant Pacific Octopus", quantity:1, scientificName:'Octopus
               dofleini'}
         ]},
         {name:"Fresh Water/", children:[
            {name:"Freshwater Stingray", quantity:7,
               scientificName:'Potamotrygen motoro'}
         ]}
      ]},

      {name:"Reptile House/", children:[
         {name:"Lizard House/", children:[
            {name:"Cuban Ground Iguana", quantity:29, scientificName:'Cyclura
               nubila nubila'},
            {name:"Desert Iguana", quantity:14, scientificName:'Dipsosaurus
               dorsalis'},
            {name:"Marbled Salamander", quantity:6, scientificName:'Ambystoma
               opacum'}
         ]},
         {name:"Snake House/", children:[
            {name:"Indian Rock Python", quantity:1, scientificName:'Python
               molurus molurus'}
         ]}
      ]},
        {name:"Monkey House/", children:[
           {name:"Howler Monkey", quantity:15, scientificName:'Alouatta spp.'},
           {name:"Orangutan", quantity:7, scientificName:'Pongo pygmaeus'},
           {name:"Guinea Baboon", quantity:3, scientificName:'Papio papio'}
        ]},

        {name:"Lion Enclosure/", children:[
           {name:"Lion", quantity:12, scientificName:'Panthera leo'}
        ]}
     ]}
});
```

The *Tree* object class provides methods that manipulate tree data in this format, allowing you to:

- add, move, remove, and rename nodes,
- query and traverse the structure of a tree,
- load and unload nodes,
- open and close folders,
- query and traverse open nodes, and
- sort open nodes.

To create a new *Tree* object, use the `create` method and initialize the object's root property to a node object that contains all of the nodes in the tree:

```
Tree.create({
   ID:"animalTree",
   root: {name:"Zoo/", children:[
      . . .
   ]}
});
```

You can then access and manipulate the data by calling methods of this object. See *"Adding, moving, and removing tree nodes" on page 151* for details.

Initializing a treeGrid

The *TreeGrid* widget class provides an interactive, graphical interface to a *Tree* data object. A treeGrid displays all of the open nodes in a tree—all nodes whose paths from the root contain only open folder nodes—one per row. Users can then open and close these nodes.

The first column in the treeGrid displays the titles or names of the open nodes, with:

- indentation to indicate each node's level in the hierarchy, and
- an icon to indicate whether each node is a leaf, a closed folder, or an open folder.

Users can click on the icon to open or close the node. *TreeGrids* have the same two fundamental properties as *ListGrid* and *DetailViewer* classes. These properties are listed in Table 8.2.

Table 8.2: TreeGrid fundamental properties

Property	Description	Default
`data`	A tree object (i.e., an instance of the *Tree* class) specifying the tree data. The treeGrid automatically observes changes to data and updates accordingly.	`[]`
`fields`	An array of field objects, specifying the order, layout, dynamic calculation, and sorting behavior of each field in the treeGrid. By default, the first field definition given for a treeGrid will be used as the tree field. If you want to control which field becomes the tree field, you can set the `treeField` property to `true` inside any field definition.	`null`

TreeGrids support the same setter methods for these properties as *ListGrid* widgets. See Table 7.4 on page 125 for details.

By default, a treeGrid draws a single column containing the titles/names of all open nodes, with appropriate indentation and icons. You can define additional data columns, and generate field values dynamically, as you would for a listGrid. Refer to *"Initializing a listGrid or detailViewer" on page 123*, and *"Working with listGrid values" on page 138* for details.

Three additional *TreeGrid* properties, described in Table 8.3, control high-level filtering of the type and order of nodes displayed in the widget.

Table 8.3: TreeGrid filtering properties

Property	Description	Default
`displayNodeType`	Specifies the type of nodes displayed in the treeGrid. Set to one of the following values: • `null`—Both folders and leaves • `folders`—Folders only • `leaves`—Leaves only	`null`
`showRoot`	Specifies whether the root node should be displayed in the treeGrid.	`false`
`separateFolders`	Specifies whether folders and leaves should be segregated in the treeGrid display. With `separateFolders:true` and `sortDirection:descending`, folders are displayed before their sibling leaves; with `sortDirection:ascending`, leaves are displayed before their sibling folders.	`false`

Example: TreeGrid initialization

The example file `treeGrid_init.html` (shown in Figure 8.1) creates a treeGrid to display the `animalViewer` object given in the *"Example: Tree data"* above:

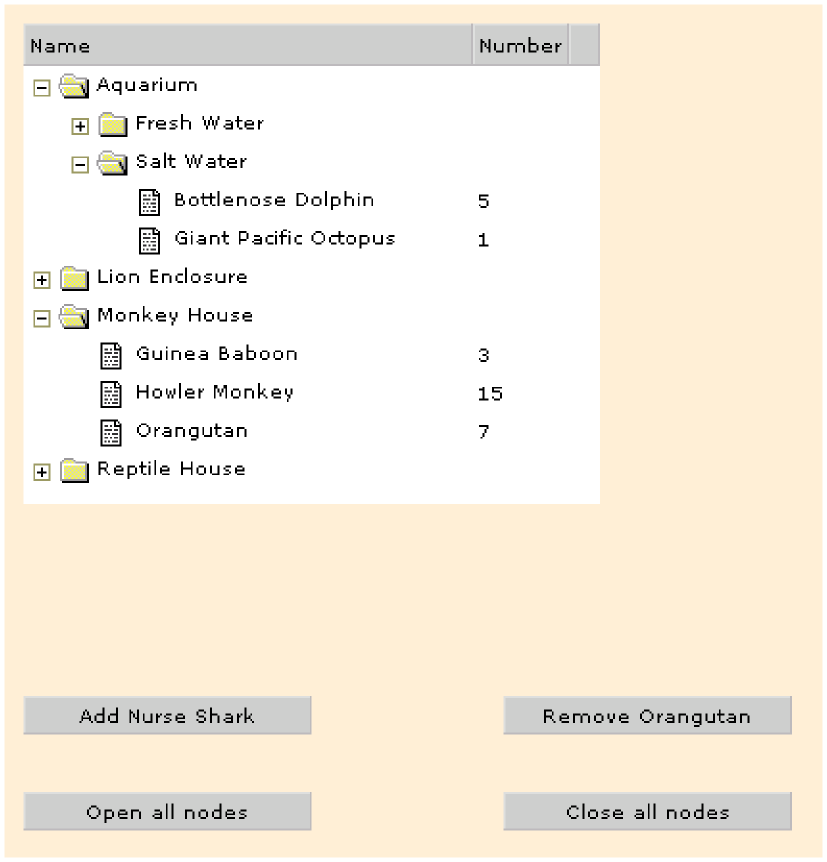


Figure 8.1: Example of TreeGrid initialization

This example adds a `quantity` property to each leaf node in `animalViewer`, and defines an additional column object to display this property:

```
var animalFields = [
   TreeGrid.TREE_FIELD,
   {name:"quantity", title:"Number", size:50}
];
```

The treeGrid itself is initialized with the following script:

```
TreeGrid.create({
   ID:"animalViewer",
   data:animalTree,
   fields:animalFields,
   left:50,
   top:50,
   width:300,
   height:250,
   canDragRecordsOut:true,
   canAcceptDroppedRecords:true
});
```

Configuring treeGrid appearance and behavior

TreeGrid widgets inherit much of their appearance and behavior, including layout, styles, selection, and sorting, from the *ListGrid* class.

The layout and styles of a treeGrid are specified with the same widget and column properties as a listGrid. Refer to *"Configuring listGrid layout and appearance" on page 128* for details. *TreeGrid* widgets also support several widget and node properties for specifying the icons displayed alongside each node's title, as detailed in Table 8.4.

Table 8.4: TreeGrid icon properties

Property	Description	Default[1]
`fileImage`	The filename of the default icon for all leaf nodes in this treeGrid. Use the `*node*.icon` property (`null` by default) to specify a custom image for an individual node.	`"[SKIN]/file.gif"`
`folderClosedImage`	The filename of the default icon for all closed folder nodes in this treeGrid. Use the `*node*.icon` property (`null` by default) to specify a custom image for an individual folder node. The same custom image will be used for both the open and closed folder images.	`"[SKIN]/folder_closed.gif"`
`folderOpenImage`	The filename of the default icon for all open folder nodes in this treeGrid.	`"[SKIN]/folder_open.gif"`
`folderDropImage`	The filename of the icon displayed for a folder node that will accept drag-and-drop data when the mouse is released. See *"Dragging and dropping tree nodes" on page 153* for details.	`"[SKIN]/folder_drop.gif"`
`manyItemsImage`	The filename of the icon displayed for multiple files and/or folders.	`"[SKIN]/folder_file.gif"`
`arrowOpenImage`	The filename of the icon displayed next to an open folder node.	`"[SKIN]/arrow_open.gif"`
`arrowClosedImage`	The filename of the icon displayed next to a closed folder node.	`"[SKIN]/arrow_closed.gif"`
`arrowOpeningImage`	The filename of the icon displayed next to a folder node while it is opening.	`"[SKIN]/arrow_opening.gif"`
`iconSize`	The standard size (same height and width, in pixels) of node icons in this treeGrid.	`16`
`indentSize`	The amount of indentation (in pixels) to add to a node's icon/title for each level down in this tree's hierarchy.	`16`

[1] All standard images and icons used by treeGrids are given relative to their skin root (e.g. `[ISOMORPHIC]/skins/standard/images/TreeGrid`). This is given as part of the path in the default values by the special directory variable `[SKIN]`.

Reference

TreeGrid widgets inherit the interactive sorting and selection behaviors, and all related properties and methods, of the *ListGrid* class. Refer to "*Sorting listGrid records" on page 131* and "*Selecting listGrid records" on page 133*, for details.

Refer to Appendix A, "*Widget Initialization Templates*," for *TreeGrid* initialization templates that include all *TreeGrid*, column, and node properties available for initialization, including properties documented elsewhere in this chapter and relevant properties inherited from the *Canvas* and *ListGrid* classes.

Overriding standard treeGrid icons

To customize the file image for a treeGrid, set the `fileImage` property to the filename of the custom image. This image is assumed to be in an images directory relative to the HTML file where the treeGrid is defined `(./images/image.gif)`.

```
TreeGrid.create({
    ID:"animalViewer",
    data:animalTree,
    fields:animalFields,
    left:50,
    top:50,
    width:300,
    height:250,
    canDragRecordsOut:true,
    canAcceptDroppedRecords:true,
    fileImage:'animal_image.gif'
});
```

To customize the icon for individual treeGrid nodes, set the node's `icon` property to the filename of the custom image. Again, this is assumed to be within an images directory relative to the HTML file where the treeGrid is defined.

```
Tree.create({
    ID:"animalTree",
    root: {name:"Zoo/", children:[
        {name:"Aquarium/", children:[
            {name:"Salt Water/", children:[
                {name:"Bottlenose Dolphin", quantity:5,
                    scientificName:'Tursiops truncatus', icon:'dolphin.gif'},
                {name:"Giant Pacific Octopus", quantity:1,
                    scientificName:'Octopus dofleini', icon:'octopus.gif'}
            ]},
            . . .
        ]}
        . . .
    ]}
})
```

To customize the look of all treeGrids within your application, create a new custom skin. Then replace the images in the `images/TreeGrid` directory of the new skin with your own custom images, and load the new skin with the page. For more information on creating custom skins, see *"Using and customizing ISC skins" on page 75*.

Adding, moving, and removing tree nodes

TreeGrid widgets automatically "observe" their tree data for changes and redraw accordingly. To add, move, or remove nodes in a tree, call the following methods of the tree data object (`treeGrid.data`):

- `add(node, parent)`
- `addList(nodeList, parent)`
- `move(node, newParent)`
- `moveList(nodeList, newParent)`
- `remove(node)`
- `removeList(nodeList)`

The `node`, `parent`, and `newParent` parameters are references to tree node objects, and `nodeList` is an array of references to tree node objects.

The `treeviewer_init.html` example (shown in Figure 8.1 on page 148) contains a button to add a nurse shark to the data set. This button's '`click`' property is set to the tree method below:

```
animalTree.add({name:'Nurse Shark', quantity:1, scientificName:'Ginglymostoma
    cirratum'}, saltWaterTank)
```

The data of the node is specified as an object literal, followed by the parent folder in which to add the node. If you have a string representing the path to a node (from the root), you can get the node's object reference with the `find(path)` method. For example, the parent folder in this case is set by the variable `saltWaterTank`, which is defined earlier in the code as:

```
var saltWaterTank = animalTree.find("Zoo/Aquarium/Salt Water/");
```

The `treeviewer_init.html` example also contains a button to remove an orangutan from the tree data set. This button's '`click`' property is set to the tree method below:

```
animalTree.remove(orangutanNode)
```

Similarly to above, `orangutanNode` is defined earlier in the code as:

```
var orangutanNode = animalTree.find("Zoo/Monkey House/Orangutan");
```

The paths set by this and the previous variable could instead be put directly into the tree method argument. Variables are used here for readability in the code.

Nodes may also be moved within or between treeGrids by drag-and-drop interactions. See *"Dragging and dropping tree nodes" on page 153* for details.

Expanding and collapsing tree nodes

One reason to organize and present data in a tree is to allow filtering of the displayed data by expanding and collapsing the folders in a treeGrid. By default, all folders in a tree are closed (and therefore collapsed in the treeGrid display). You can open and close folders interactively with a treeGrid, or programmatically with methods of the tree data object.

To interactively toggle the open state of a folder in a treeGrid, either:

- click in the folder's row before its title—in the indented area or on the folder's icon, or
- double-click anywhere in the folder's row.

To manipulate the open state of folders programmatically, call the methods of the tree data object (`treeGrid.data`) described in Table 8.5.

Table 8.5: Tree node manipulation methods

Method	**Description**
`isOpen(node)`	Returns `true` if the specified *node* is open.
`openFolder(node)`	Opens the specified *node*.
`closeFolder(node)`	Closes the specified *node*.
`toggleFolder(node)`	Changes the state of the *node* from opened to closed or vice versa.
`openAll(node)`	Opens all folders within and including the specified *node*. If no node is specified, all nodes will be opened.
`closeAll(node)`	Closes all folders within and including the specified *node*. If no node is specified, all nodes will be closed. If the root node is not visible, it will not be closed; only the subfolders of the root node will close. If the root node is visible, the root node will be closed along with its subfolders.

These methods may be called on a tree object at any time, regardless of whether the tree is assigned to a treeGrid widget. If the tree is assigned to a treeGrid widget that is already drawn, the treeGrid will automatically redraw to reflect changes to the visible hierarchy.

Reference

For more information on these and other `tree` methods, refer to the *Tree* class in the *Isomorphic SmartClient Object Reference* and Appendix D, "*Tree Methods*."

Dragging and dropping tree nodes

Like listGrids, treeGrids support drag-and-drop of data both within a widget, and between widgets. The same drag-and-drop properties apply to treeGrids and their nodes as to listGrids and their rows. Refer to *"Dragging and dropping listGrid records" on page 134* for details.

Dragging nodes in a treeGrid has a different effect than dragging rows in a listGrid. In a listGrid, rows can be reordered by dragging. In a treeGrid, nodes can be dragged to different folders in the hierarchy, but not to specific positions relative to their siblings. *TreeGrids* therefore do not display an insertion line during dragging. When a node or nodes are dragged over a folder that can accept them—a folder that is not one of, or a descendant of, one of the nodes being dragged—that folder's icon changes to the `treeGrid.folderDropImage` image. Nodes that are dropped on a folder are inserted in the current sorting order, or at the bottom of the folder if unsorted.

Warning

The *TreeGrid* and *ListGrid* classes support drag-and-drop reordering of data on the client-side only. Custom server code is required to move data on the server.

Handling treeGrid events

Like listGrids, treeGrids handle the standard mouse and drag-and-drop events to implement their standard behaviors; you should not set your own handlers for these events. *TreeGrids* also handle the widget-level `recordDoubleClick` psuedo-event to expand/collapse folders automatically, and to call an `openLeaf` handler for double-clicked leaf nodes. In brief, you can implement the following handlers for treeGrid events:

- `treeGrid.recordClick`
- `field.recordClick`
- `field.recordDoubleClick`
- `treeGrid.openFolder`
- `treeGrid.openLeaf`

The first three of these handlers are inherited from the *ListGrid* class. *"Handling listGrid record events" on page 141*, for details of their usage and parameters. The row variable/ parameter in this case refers to the relevant node object in the tree's data.

The last two of these handlers, `openFolder` and `openLeaf`, are specific to the *TreeGrid* class. These handlers must be specified as a function, whose single parameter is a reference to the relevant leaf or folder node in the tree's data. This handler might be used to display more information about a node, for example. Or, you might want to track which folder or leaf was double-clicked by overridding the `openFolder` or `openLeaf` method and calling its superclass method as follows:

```
myTreeGrid.openFolder = function (node) {
    this.Super("openFolder", arguments);
    alert("The node that was opened was " + node.title);
}
```

CHAPTER 9

Menus, Toolbars, and Menubars

In addition to the simple building-block widgets and the complex data-driven widgets, the Isomorphic SmartClient system provides two widget classes for command and/or navigation tools. The *Menu* and *Toolbar* classes are covered in this chapter.

- The *Menu* class implements interactive menu widgets, with optional icons, submenus, and shortcut keys. You can assign an action to each menu item, to be executed when the user selects that item. A menu can be displayed with a clickable header (as in a standard menu bar), or as a headerless context menu or submenu.
- The *Toolbar* class creates a row or column of buttons as child widgets, each of which may carry an action to execute when clicked. Toolbars automatically size and position their buttons to fit the allocated space, and automatically handle the mutual exclusivity of radio buttons.
- The *Menubar* class creates a row or column of menus as child widgets, essentially combining the features of the *Toolbar* and *Menu* classes.

In this chapter:

Menu widgets

The *Menu* class is a subclass of the *ListGrid* class, but its default functionality is sufficiently different that you should usually treat it as a standalone class. There are two general steps to initializing a *Menu* widget:

1. Specify the menu's items.
2. Configure properties of the menu itself.

Menu items

Menu items, like list rows, are specified in the widget's `data` property. Each menu item has the fundamental properties described in Table 9.1.

Table 9.1: Menu item fundamental properties

Property	Description
`item``.title`	The text displayed for the menu item.
`item``.submenu`	An object reference to another menu widget, to display as a submenu when the menu item is selected.
`item``.isSeparator`	If `true`, defines a horizontal separator in the menu. Typically specified as the only property of a menu item, since a menu item with `isSeparator:true` will not display a title or respond to mouse events.
`item``.enabled`	Affects the visual style and interactivity of the menu item. If `item.enabled` is `false`, the menu item will not respond to mouse rollovers or clicks.

Menu properties

A `menu` widget itself has the fundamental properties listed in Table 9.2.

Table 9.2: Menu widget fundamental properties

Property	Description
`menu``.title`	The text displayed in the menu's clickable header, accompanied by a small down-arrow image. If *`menu`*`.title` is not specified or if *`menu`*`.headerHeight` is `0`, the menu will not display a header.
`menu``.data`	An array of menu item objects. See Table 9.1 for fundamental properties that can be specified for each menu item, and *"Example: Menu initialization"* for how menu items are given in the `data` array.

Warning Neither submenus nor context menus should display a header. Do not specify a title for a menu widget that will be used as a submenu or context menu.

When a menu is shown, it activates the "click mask", an event handler that captures all mouse events outside of the menu. See *"Mouse events" on page 53* for details. Clicking anywhere on the page outside of the menu will hide the menu (and deactivate the click mask).

Menu items, like form items, are usually specified outside of the widget initialization block for code readability and maintenance.

Reference Refer to Appendix A, "*Widget Initialization Templates*," for *Menu* initialization templates that include all available menu and menu item properties available, including properties documented elsewhere in this chapter and relevant properties inherited from the *Canvas* and *ListGrid* classes.

Example: Menu initialization

The example file `menu_init.html` (shown in Figure 9.1) creates a simple **File** menu that lists several standard file operations, including a **Recent Files** submenu.

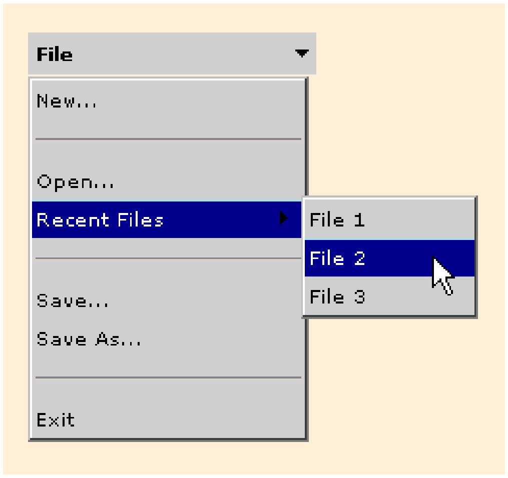


Figure 9.1: Example of menu initialization

The top-level menu is defined and drawn with the following script:

```
var fileMenuItems = [
    {title:"New..."},
    {isSeparator:true},
    {title:"Open..."},
    {title:"Recent Files", submenu:recentMenu},
    {isSeparator:true},
    {title:"Save..."},
    {title:"Save As..."},
    {isSeparator:true},
    {title:"Exit"}
],

Menu.create({
    ID:"fileMenu",
    left:50,
    top:50,
    width:100,
    data:fileMenuItems,
    title:"File"
});
```

The **Recent Files** menu item specifies a submenu. The submenu widget *must* be defined before the menu item in which it will appear. The following script defines the submenu for this example:

```
    var recentMenuItems = [
        {title:"File 1"},
        {title:"File 2"},
        {title:"File 3"}
    ],

    Menu.create({
        ID:"recentMenu",
        width:100,
        data:recentMenuItems
    });
```

Warning

Submenus must be defined before the menus that contain them.

A title for the submenu widget is not specified because only the menu items should be displayed when a submenu is shown.

Configuring menu appearance

Menu widgets and menu item objects provide a range of properties that affect a menu's appearance, including the menu's layout and icons.

Menus provide the sizing properties described in Table 9.3.

Table 9.3: Menu sizing properties

Property	**Description**	**Default**
`width`	The width of the menu.	`150`
`menuButtonHeight`	The height of the menu button.	`22`
`menuButtonWidth`	The width of the menu button.The value of width will be used unless `menuButtonWidth` is explicitly set.	`null`

Property	Description	Default
showMenuBelow	If true, the menu will appear above the menu button instead of below it.	false
cellHeight	The height of each item in the menu, in pixels.	20

Each item in a menu may display either a checkmark icon or a custom icon to the left of its title. The menu and menu item properties given in Table 9.4 specify the display of these images.

Table 9.4: Menu icon properties

Property	Description	Default
item.checked	If true, this item displays a standard checkmark image to the left of its title.	null
item.icon	The base filename for this item's custom icon. If *item*.icon and *item*.checked are both specified, only the custom icon will be displayed. The path to the loaded skin directory and the skinImgDir are prepended to this filename to form the full URL.	null
item.iconWidth	The width applied to this item's icon, if *item*.icon is specified.	16
item.iconHeight	The height applied to this item's icon, if *item*.icon is specified.	16
menu.iconWidth	The default width applied to custom icons in this menu. This is used whenever *item*.iconWidth is not specified.	16
menu.iconHeight	The default height applied to custom icons in this menu. This is used whenever *item*.iconHeight is not specified.	16

Menus make use of six standard images, described in Table 9.5.

Table 9.5: Menu images

Image	Image Name	Description
▼	menu_button.gif	Down-arrow image displayed at the far right of a menu header, if any.
▼	menu_button_disabled.gif	Down-arrow image displayed at the far right of a disabled menu header, if any.
▶	submenu.gif	Right-arrow image displayed at the far right of a menu item with a submenu.
▶	submenu_disabled.gif	Right-arrow image displayed at the far right of a menu item with a disabled submenu.
✔	check.gif	Checkmark image displayed at the far left of menu items with checked:true.
✔	check_disabled.gif	Checkmark image displayed at the far left of disabled menu items with checked:true.

These images must be located in the subdirectory specified by the loaded skin image directory (default: `"[ISOMORPHIC]/skins/standard"`) appended with `*menu*.skinImgDir` (default: `"images/Menu/"`).

Each menu item may display up to four parts. Each of these parts corresponds to a column in the menu. This is one case where it's useful to think about the *Menu* class as a subclass of the *ListGrid*. The contents of these columns, in left-to-right order, are:

- checkmark or custom icon,
- title,
- shortcut key (60 pixels), and
- submenu arrow icon (20 pixels).

You should *not* need to set these, since columns are defined and drawn automatically based on the initial settings of the menu's items. However, if you want to explicitly hide or show the icon, key, or submenu columns, you can do so by initializing the `menu` column properties listed in Table 9.6.

Table 9.6: Menu column properties

Property	**Description**	**Default**
`showIcons`	A boolean, indicating whether the checkmark/custom icon column should be displayed. If `showIcons` is not set, the menu will show the icon column only if one of its items specifies an `icon`, `checked`, `checkIf`, or `dynamicIcon` property.	`null`
`showKeys`	A boolean, indicating whether the shortcut key column should be displayed. If `showKeys` is not set, the menu will show the key column only if one of its items specifies a `keys` property. If `showKeys` is `false`, the keys will not be displayed, but will still function.	`null`
`showSubmenus`	A boolean, indicating whether the submenu indicator column should be displayed. If `showSubmenus` is not set, the menu will show the indicator column only if one of its items specifies a `submenu` property. If `showSubmenus` is `false`, the submenu arrows will not be displayed, but submenus will still appear on rollover.	`null`

Defining menu actions

When a user selects an item in a menu, the menu generates an event to which your scripts can respond. Menu events are handled separately from the system events discussed in Chapter 3, *"Handling Events,"* , but are nevertheless handled in a similar manner. You can assign actions to individual menu items, or to the menu widget as a whole, by setting the event handler properties described in Table 9.7.

Table 9.7: Menu event handlers

Property	Description
item.click	A function (or string of script that is automatically converted to a function) to execute when this menu item is clicked by the user. The `click` function is passed a target parameter that refers to either the menu widget or to *menu*.target if specified.
menu.itemClick	A function to execute when a menu item with no click handler is clicked by the user. The `itemClick` method is passed an item parameter that is a reference to the clicked menu item.

You can also assign one or more shortcut keys to each menu item, allowing the user to execute that item's action without opening the menu. Pressing an item's shortcut key has the same effect as clicking on the item. The relevant `click` or `itemClick` handler is executed in both cases. The properties listed in Table 9.8 specify and control a menu's shortcut keys.

Table 9.8: Menu shortcut key properties

Property	Description
menu.useKeys	A boolean indicating whether this menu should use shortcut keys. Set `useKeys` to `false` in a menu's initialization block to explicitly disable shortcut keys.
item.keys	A single-character string, a character code, or an array of single-character strings or codes, specifying the shortcut key(s) for this item.
item.keyTitle	A string to display in the shortcut-key column for this item. If *item*.keyTitle is not specified, the first value in *item*.keys will be used by default.

Implementing dynamic menus

It's often useful to change a menu "on the fly" to reflect the current context. For example, you might want to enable a particular menu item only when a user has selected one or more rows of a list.

Each menu item provides the four properties described in Table 9.9 as dynamic alternatives to its `enabled`, `checked`, `title`, and `icon` properties.

Table 9.9: Dynamic menu item properties

Property	Description
item.enableIf	A string of script that is evaluated to a boolean value for the item's `enabled` property whenever the menu is shown or a shortcut key is pressed.
item.checkIf	A string of script that is evaluated to a boolean value for the item's `checked` property whenever the menu is shown or a shortcut key is pressed.

Property	Description
`item.dynamicTitle`	A string of script that is evaluated to a string value for the item's `title` property whenever the menu is shown or a shortcut key is pressed.
`item.dynamicIcon`	A string of script that is evaluated to a string value for the item's `icon` property whenever the menu is shown or a shortcut key is pressed.

Menu widgets also provide the setter methods detailed in Table 9.10 for directly setting these item properties "on the fly."

Table 9.10: Menu item setter methods

Method	Action
`menu.setItemEnabled(item, newState)`	Sets the `enabled` property of the menu item to a `newState` boolean value. Returns `true` if this changes the value of the `enabled` property.
`menu.setItemChecked(item, newState)`	Sets the `checked` property of the menu item to a `newState` boolean value. Returns `true` if this changes the value of the `checked` property.
`menu.setItemTitle(item, newTitle)`	Sets the `title` property of the menu item to a `newTitle` string value. Returns `true` if this changes the value of the `title` property.
`menu.setItemIcon(item, newIcon)`	Sets the `icon` property of the menu item to a `newIcon` image. Returns `true` if this changes the value of the `icon` property.

Example: Visual properties menu

The example file `menu_features.html` (shown in Figure 9.2) uses many of the features described in the previous sections of this chapter to implement a menu that controls the visual properties of another widget:

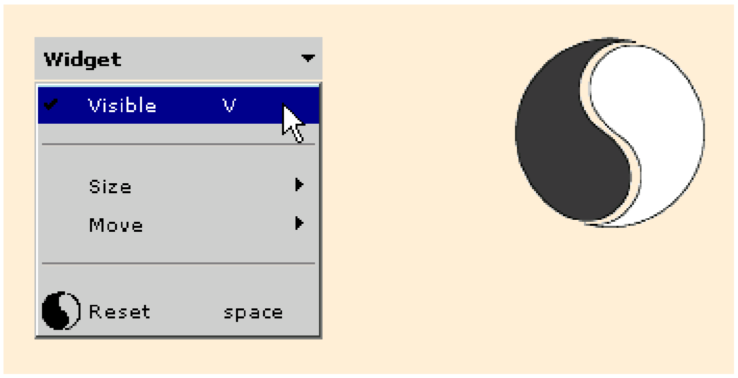


Figure 9.2: Example of a visual properties menu

The following script creates the top-level menu in this example:

```
var menuItems = [
   {title:"Visible",
      checkIf:"widget.isVisible()",
      click:"widget.isVisible() ? widget.hide() : widget.show()",
      keys:["V","v"]},
        {isSeparator:true},
        {title:"Size",
            enableIf:"widget.isVisible()",
            submenu:sizeMenu},
        {title:"Move",
            enableIf:"widget.isVisible()",
            submenu:moveMenu},
        {isSeparator:true},
        {title:"Reset",
            click:"widget.setRect(300,50,100,100); widget.show()",
            icon:"yinyang_icon.gif",
            iconWidth:20,
            iconHeight:20,
            keys:" ",
            keyTitle:"space"}
    ],

    Menu.create({
        ID:"mainMenu",
        data:menuItems,
        left:50,
        top:50,
        width:150,
        title:"Widget"
    });

    // Set the new menu to also work as a context menu
    widget.contextMenu = mainMenu;
```

The `checkIf` and `enableIf` properties dynamically check/uncheck and enable/disable menu items without additional scripting. The `checkIf` property is also used in the menu items for the `sizeMenu` submenu.

An array value for the keys property of the **Visible** menu item allows the user to type either an uppercase or a lowercase shortcut key for this item. The first string in the array is displayed in the menu, since no `keyTitle` is specified. For the **Reset** menu item, `keyTitle` allows a visible name for a spacebar shortcut key.

The menu is set to be the widget context menu so that right-clicking on the widget (in this case the image) will bring up the menu as a context menu. To define a context menu, the *widget*`.contextMenu` property must be set to a pre-defined menu. You must set it to a variable that defines the context menu. You cannot set *widget*`.contextMenu` to a script.

To show a context menu, you do not need to set the `showContextMenu` event handler for the widget. If a context menu is defined for a widget, it will automatically show when a user right-clicks on it.

Toolbar widgets

Like the *Menu* class, the *Toolbar* class implements a widget that presents a graphical interface for selecting from a set of actions. Toolbars present these actions as a set of buttons, arranged in a row or column.

Toolbar properties

A toolbar widget has the fundamental properties described in Table 9.11.

Table 9.11: Toolbar widget fundamental properties

Property	Description
`buttons`	An array of button object initializers. See *"Button widget properties" on page 88* for standard button properties, and Table 9.12 for additional properties and exceptions.
`vertical`	**Default:** `false` Indicates whether the buttons are drawn horizontally from left to right—`false`, or vertically from top to bottom—`true`.
`buttonDefaults`	Settings to apply to all buttons of a toolbar. Properties that can be applied to button objects can be applied to all buttons of a toolbar by specifying them in `buttonDefaults` using the following syntax: `buttonDefaults:{property:value, ...}` See *"Button widgets" on page 88* for more information on these button properties.

The button-object initializers set in a toolbar's `buttons` property may contain any standard button properties, including `title`, `selected`, `enabled`, `actionType`, and `click`. See *"Button widgets" on page 88* for more information. To set button properties for all buttons, include those properties in the toolbar's `buttonDefaults`. To set button properties for individual buttons, include those properties in the desired button's object initializer.

Since the toolbar handles sizing and positioning of its buttons, you do not need to set the left or top properties for each button. If they are provided, these properties will be ignored. Instead, each button-object initializer supports the toolbar-specific properties listed in Table 9.12.

Table 9.12: Toolbar button sizing and positioning properties

Property	Description
`button``.width`	**Default:** `"*"` Specifies the width of this button. The `width` property may be set to one of the following: • an absolute number of pixels, • a named property of the toolbar that specifies an absolute number of pixels, • a percentage of the remaining space (e.g. `'60%'`), or • `"*"` to allocate an equal portion of the remaining space.
`button``.height`	Specifies the height of this button.
`button``.extraSpace`	Specifies an optional amount of extra space, in pixels, to separate this button from the next button in the toolbar.

You can mix different button types (*standard*, *checkbox*, and *radio*) in the same toolbar.

Defining toolbar actions

When a toolbar is created, it creates a child widget for each object initializer in the buttons list. Since each toolbar button is itself a widget, you can access its properties and methods directly. Since the toolbar has created the buttons, though, you'll need to get each button's object reference from the toolbar, by calling:

```
toolbar.getButton(buttonNum)
```

where *buttonNum* is the index of the button's object initializer in *toolbar*.buttons.

Toolbars observe any changes to the selected state of their buttons, and automatically deselect radio buttons as appropriate to enforce mutual exclusivity. All radio buttons in a toolbar are mutually exclusive. To create more than one group of toolbar radio buttons you must create multiple toolbars.

You can select or deselect toolbar buttons directly with the *button*.select and *button*.deselect methods, or by using the toolbar methods listed in Table 9.13.

Table 9.13: Toolbar button selection methods

Method	Description
selectButton(*buttonNum*)	Selects the button that was defined by the object at index *buttonNum* in *toolbar*.buttons.
deselectButton(*buttonNum*)	Deselects the button that was defined by the object at index *buttonNum* in *toolbar*.buttons.

To respond to user actions in a toolbar, you can define a standard 'click' handler in each button's object initializer. As a standard system event, click will "bubble up" from a button widget to the toolbar widget by default. If a 'click' event is not specified for a button, the itemClick handler will be used instead, if it has been defined. If you want to handle toolbar-button clicks at the toolbar level, without having to determine exactly where the clicks occurred, do not define a 'click' handler for the relevant button(s). Instead, define an itemClick handler for the toolbar:

```
toolbar.itemClick(item, itemNum);
```

where the handler parameters are:

- *item*—a reference to the clicked button widget, and
- *itemNum*—the index of the clicked button's initializer in *toolbar*.buttons starting with 0.

Note

A toolbar's itemClick handler will not be executed if you have specified a click handler for the clicked button.

Example: Visual properties toolbar

The example file `toolbar_features.html` (shown in Figure 9.3) implements a toolbar that provides the same functions as the menu in the `menu_features.html` example (shown in Figure 9.2 on page 162):

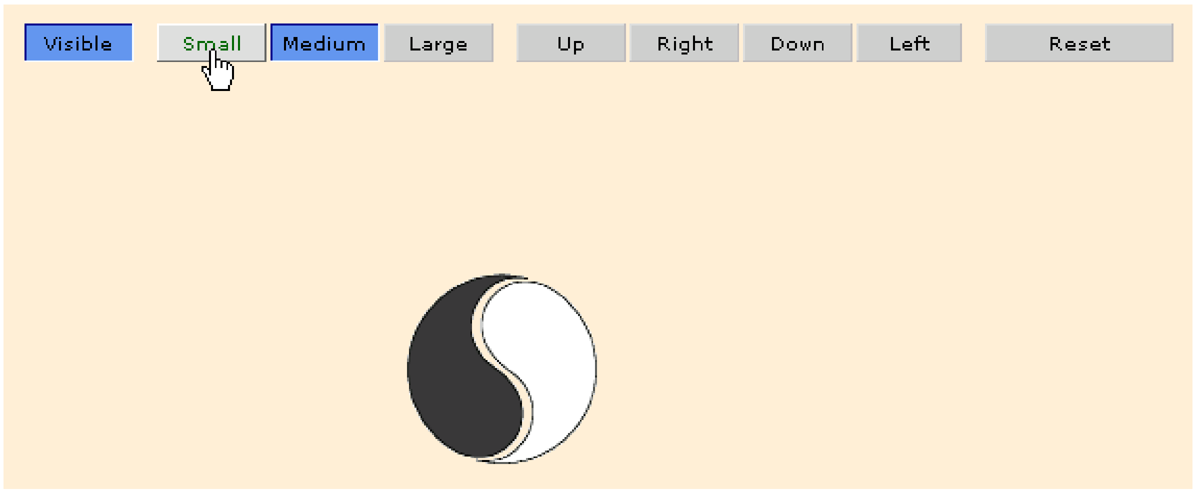


Figure 9.3: Example of a visual properties toolbar

The following script creates the toolbar in this example:

```
Toolbar.create({
    ID:"toolbar",
    left:50,
    top:50,
    width:600,
    membersMargin:2,
    buttons:[
        {title:"Visible",
            actionType:"checkbox",
            selected:true,
            click:"widget.isVisible() ? widget.hide() : widget.show()",

// Note: both 'click' and 'doubleClick' events are defined here - double
// clicking will execute the click event handler once, and the doubleClick
// event handler once. The state of the checkbox button (checked vs
// unchecked) will be changed twice - once on each mouseUp event.

            doubleclick:"widget.isVisible() ? widget.hide() : widget.show()",
            extraSpace:10},
        {title:"Small",
            actionType:"radio",
            click:"widget.setWidth(50);widget.setHeight(50)"},
        {title:"Medium",
            actionType:"radio",
            selected:true,
            click:"widget.setWidth(100);widget.setHeight(100)"},
        {title:"Large",
            actionType:"radio",
            click:"widget.setWidth(200);widget.setHeight(200)",
            extraSpace:10},
        {title:"Up",
            click:"widget.moveBy(0,-20)"},
        {title:"Right",
            click:"widget.moveBy(20,0)"},
        {title:"Down",
            click:"widget.moveBy(0,20)"},
        {title:"Left",
            click:"widget.moveBy(-20,0)", extraSpace:10},
        {title:"Reset",
            click:"widget.setRect(250,180,100,100);
            widget.show();
            this.parentElement.selectButton(0);
            this.parentElement.selectButton(2)",
            size:100}
    ]
});
```

In this example, the object initializers for the toolbar buttons are embedded in the toolbar's initialization block. You may prefer to define the list of button initializers in a separate variable, depending on the complexity of your toolbar widget.

Most of the button properties set in this example are standard toolbar button properties: `title`, `actionType`, `selected`, and `click`. Only the **Reset** button specifies a size property. All other buttons are sized equally to fill the remaining space. The **Visible**, **Large**, and **Left** buttons each specify an `extraSpace` property to separate buttons into functional groups.

The `this.parentElement` is used in the **Reset** button's click handler. Since `selectButton` is a *Toolbar* method, it cannot be called directly by a button object. However, since *Toolbar* buttons are created as children of the `toolbar` widget, they can refer to the toolbar via the `parentElement` property. This syntax can be used within any *Toolbar* button's `click` handler to access the toolbar's properties and methods.

Menubar widgets

The *Menubar* class implements a widget that allows menus to be grouped into a toolbar. It is a subclass of the *Toolbar* class but, instead of having an array of buttons, the *Menubar* class has an array of menus. Menubars combine the features of both toolbars and menus to allow users to select actions from drop-down menu buttons arranged in a row or column.

Menubar properties

A menubar widget inherits all its fundamental properties from the *Toolbar* class except the `menus` property described in Table 9.14.

Table 9.14: Menubar widget fundamental properties

Property	Description
`menus`	An array of menu object initializers or instantiated menu objects. See *"Menu widget fundamental properties" on page 156* for fundamental menu properties and other properties given in this chapter.

The menu-object initializers set for a menubar's `menus` property may contain any standard menu properties and will have actions defined just as they were for menus. See *"Menu widgets" on page 156* for more information.

Create menu objects with the `autoDraw` property set to `false` before specifying them in the menubar `menus` property. When the menubar itself is initialized and drawn, the menus themselves will be drawn accordingly.

Since the menubar automatically handles positioning of its menus, you do not need to set positioning properties for each individual menu. If they are provided, these properties will be ignored. Menus can, however, be sized using the menu sizing properties listed in Table 9.3. Use the menubar sizing and positioning properties to set the location and width of the menubar on the page.

Tip

If you want to extend the width of the menubar set its `backgroundColor` property to match the default color of each menu button—`"#CCCCCC"`, by default.

Example: Visual properties menubar

The example file `menubar_features.html` (shown in Figure 9.4) implements a menubar that provides the same functions as the menu in the `toolbar_features.html` and `menu_features.html` examples presented previously:

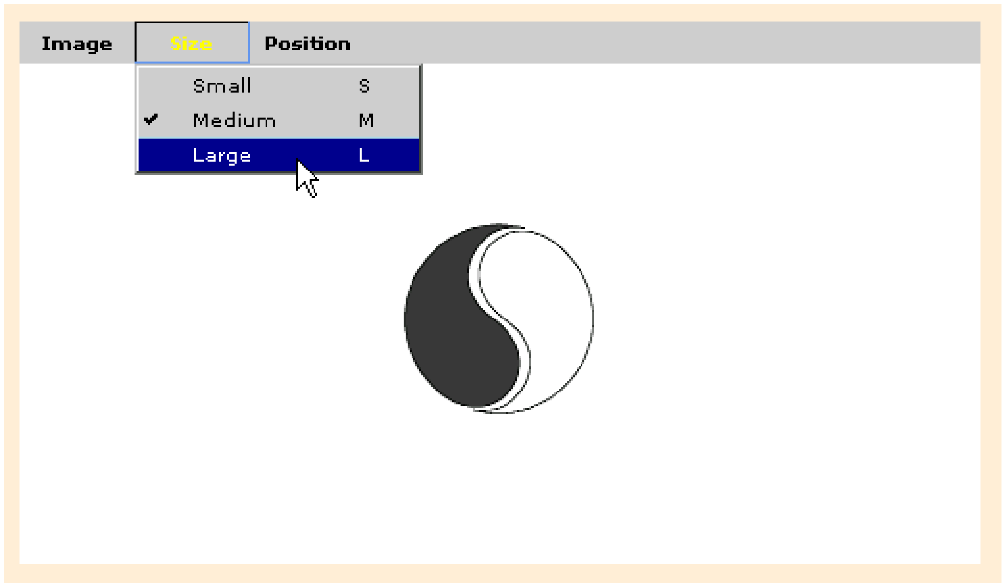


Figure 9.4: Example of a visual properties menubar

The following script creates the menubar in this example. For simplicity, the `data` arrays for the individual menus themselves have been omitted. Notice that the menus are created with `autoDraw:false` to prevent them from drawing separately from the menubar.

```
Menu.create({
   ID:"imageMenu",
   autoDraw:false,
   cellHeight:18,
   menuButtonWidth:60,
   data:[
          . . .
   ],
   title:"Image"
});

Menu.create({
   ID:"sizeMenu",
   autoDraw:false,
   cellHeight:18,
   menuButtonWidth:60,
   data: [
          . . .
   ],
   title:"Size"
});

Menu.create({
   ID:"positionMenu",
   autoDraw:false,
   cellHeight:18,
   menuButtonWidth:60,
   data:[
          . . .
   ],
   title:"Position"
});

Menubar.create({
   ID:"menuBar",
   top:75,
   left:50,
   width:500,
   backgroundColor:"#CCCCCC",
   menus:[imageMenu, sizeMenu, positionMenu]
});
```

In this example, a white canvas visually displays a bounding box. The menubar is created with a width equal to the canvas and a `backgroundColor` that is the same color as the menu buttons.

The `click` methods for the menu items in the **Position** menu are defined so that the `yinyang_icon.gif` image is not allowed to move outside of the bounding box. Also, the `click` methods for the menu items in the **Size** menu have been changed so that the the `yinyang_icon.gif` image is returned to the center of the bounding box when it is resized.

Otherwise all menu items for this example have been defined exactly as they were for the *"Example: Visual properties menu" on page 162*. The menubar simply provides an anchor for holding several menus so that the submenus in the previous menu example could be eliminated.

APPENDIX A

Widget Initialization Templates

The following scripts are provided as a starting point for the creation of your own widget initialization templates—in other words, code fragments that you can configure as macros (aka "snippets", "clips", etc.) in your web-page editor of choice. Once you've set up your own templates, inserting a complex element like a treeGrid into a web page could be as simple as pressing a shortcut key!

These scripts include the most commonly used properties specific to a widget class, along with their default values. Since the default values are provided, using one of these scripts is equivalent to calling the `create` method with no arguments at all.

For convenience, the positioning and sizing properties (`left`, `top`, `width`, `height`) of the *Canvas* superclass are also reproduced in every other class's initialization script. Many of the other *Canvas* properties may be initialized for its subclasses, but some properties are overridden by the subclass and therefore should not be changed. The *Canvas* properties that you would be most likely to add to a subclass's initialization scripts are generally the ones for which it is safe to do so, but you should test your scripts to be sure.

Widget templates are presented in JavaScript declarative format first, followed by XML declarative format.

To configure your own initialization templates:

1. Remove the `property:value` pairs (or `property="value"` pairs if you're using XML) for any properties whose values you never intend to change.

> **Warning** If you remove the last `property:value` pair in the JavaScript declarative format, remember to remove the comma following the new last pair!

2. Edit the default values of the remaining properties to suit your requirements.

3. Save the template in the appropriate macro mechanism in your editor. You may want to create multiple templates/macros for different uses of the same widget.

If possible, configure your macros to automatically place the text-insertion point where the new widget's name should be, so you can immediately name the new instance. All variable names in these scripts are shown in italics, as a reminder that they must be changed. In Internet Explorer, an "Expected Identifier" error will be raised if you neglect to replace one of these bracketed names.

JavaScript Widget Templates

The following templates include all the properties and default values in the JavaScript declarative format.

Canvas

```
Canvas.create({
    ID:"canvasName",
    backgroundColor:"gray",
    className:"normal",
    appImgDir:"",
    backgroundImage:"",
    backgroundRepeat:"repeat",
    border:"",
    canAcceptDrop:false,
    canDrag:false,
    canDragReposition:false,
    canDragResize:false,
    canDrop:false,
    cursor:"default",
    dragAppearance:"outline",
    dragIntersectStyle:"mouse",
    dragRepositionCursor:"move",
    height:100,
    left:0,
    margin:0,
    overflow:"visible",
    padding:0,
    position:"absolute",
    top:0,
    visibility:"inherit",
    width:100,
    contents:"Canvas contents",
    contextMenu:Menu.create({
        constructor:"Menu",
        ID:"contextMenu",
        data:[
            {title:"Context Menu Item"}
        ]
    })
});
```

Label

```
Label.create({
    ID:"labelName",
    backgroundColor:"gray",
    className:"normal",
    align:"left",
    backgroundImage:"blank.gif",
    height:100,
    left:0,
    top:0,
    valign:"center",
    width:100,
    padding:0,
    wrap:true,
    contents:"Label contents"
});
```

Img

```
Img.create({
    ID:"imgName",
    appImgDir:"",
    canDragReposition:true,
    height:100,
    left:0,
    src:"blank.gif",
    state:"",
    top:0,
    width:100
});
```

StretchImg

```
StretchImg.create({
    ID:"stretchImgName",
    appImgDir:"",
    capSize:2,
    height:100,
    imageType:"tile",
    left:0,
    src:"blank.gif",
    state:"",
    top:0,
    vertical:true,
    width:100,
    items:[
        {height:"capSize", name:"start", width:"capSize"},
        {height:"*", name:"stretch", width:"*"},
        {height:"capSize", name:"end", width:"capSize"}
    ]
});
```

Button

```
Button.create({
   ID:"buttonName",
   actionType:"button",
   align:"center",
   borderSize:1,
   canDrag:false,
   canDrop:false,
   dragAppearance:"outline",
   enabled:true,
   height:20,
   left:0,
   selected:false,
   showDown:true,
   showRollOver:true,
   title:"Untitled Button",
   top:0,
   valign:"center",
   width:100,
   wrap:false
});
```

Scrollbar

```
Scrollbar.create({
   ID:"scrollbarName",
   scrollTarget:"",
   autoEnable:true,
   height:100,
   left:0,
   showCorner:false,
   top:0,
   vertical:true,
   width:100
});
```

Progressbar

```
Progressbar.create({
   ID:"progressbarName",
   height:12,
   left:0,
   percentDone:50,
   top:0,
   width:100
});
```

DynamicForm

```
DynamicForm.create({
   ID:"dynamicFormName",
   target:"",
   values:{field1:"field1 value", field2:"field2 value"},
   errors:{field1:"field1 error", field2:"field2 error"},
   action:"#",
   height:100,
   left:0,
    method:"post",
    top:0,
    width:400,

// See Form Item templates for various item types
    items:[
        {title:"field1 title", type:"text", name:"field1"},
        {title:"field2 title", type:"text", name:"field2"},
        {title:"Submit Form", type:"submit"}
    ]
});
```

Form items (all types)

```
DynamicForm.create({
    ID:"dynamicFormName",
    values:{field1:"field1 value", field2:"field2 value"},
    errors:{field1:"field1 error", field2:"field2 error"},

    // data items
    items:[
        {defaultValue:"value", length:"10", title:"TextTitle", type:"text",
            name:"itemName", required:false, showTitle:true, width:150},
        {defaultValue:"value", length:"6", title:"PasswordTitle",
            type:"password", name:"itemName", required:false, showTitle:true,
            width:150},
        {title:"UploadTitle", type:"upload", name:"itemName", required:false,
            showTitle:true, width:150},
        {defaultValue:"value", title:"TextAreaTitle", type:"textArea",
            height:100, name:"itemName", required:false, width:150},
        {defaultValue:"false", title:"CheckboxTitle", type:"checkbox",
            name:"itemName"},
        {defaultValue:"false", title:"RadioTitle", type:"radio",
            name:"itemName"},
        {defaultValue:"false", itemHeight:"20", title:"RadioGroupTitle",
            type:"radioGroup", vertical:"true", name:"itemName", required:false,
            valueMap:{value2:"Value 2", value1:"Value 1"}
        },
        {defaultValue:"value", multiple:"false", title:"SelectTitle",
            type:"select", name:"itemName", required:false,
            valueMap:{value2:"Value 2", value1:"Value 1"}
        },
        {defaultValue:"value", title:"SelectOtherTitle", type:"selectOther",
            name:"itemName", required:false,
            valueMap:{value2:"Value 2", value1:"Value 1"}
        },
        {defaultValue:"2002-06-04", endDay:"31", endMonth:"12", endYear:"2010",
            startDay:"1", startMonth:"1",startYear:"1970", title:"DateTitle",
            type:"date", name:"itemName", required:false},
        {defaultValue:"13:30:00", show24HourTime:"true", showSeconds:"false",
            title:"TimeTitle", type:"time", name:"itemName", required:false,
            width:100},
        {defaultValue:"value", type:"hidden",  name:"itemName", required:false},

        // button items
        {click:"alert(1)", title:"ButtonTitle", type:"button"},
        {title:"SubmitTitle", type:"submit"},
        {title:"ResetTitle", type:"reset"},
        {cellSpacing:"20", type:"toolbar",
            buttons:[
                {type:"button", click:"alert('Button 1')", title:"Button 1 Title"},
                {type:"button", click:"alert('Button 2')", title:"Button 2 Title"}
            ]
        },

        // display items
        {defaultValue:"value", outputAsHTML:"false", title:"StaticTextTitle",
            type:"staticText", wrap:"true", name:"itemName"},
        {defaultValue:"displayText", outputAsHTML:"false", type:"blurb",
            name:"itemName"},
        {defaultValue:"headerText", outputAsHTML:"false", type:"header",
            name:"itemName"},
        {type:"rowSpacer", height:20},
        {type:"spacer", height:20, width:20}
    ]
});
```

ListGrid

```
ListGrid.create({
    ID:"listGridName",
    canSort:true,
    height:300,
    left:0,
    top:0,
    width:300,
    data:[
        {col1:"value 1,1", col2:"value 1,2", enabled:true},
        {col1:"value 2,1", col2:"value 2,2", enabled:true},
        {col1:"value 3,1", col2:"value 3,2", enabled:true},
        {col1:"value 4,1", col2:"value 4,2", enabled:true},
        {col1:"value 5", col2:"value 5", isSeparator:true},
        {col1:"value 6,1", col2:"value 6,2", enabled:false},
        {col1:"value 7,1", col2:"value 7,2", enabled:false}
    ],
    fields:[
        {sortDirection:"descending", name:"col1", title:"Field 1", width:"50%"},
        {align:"right", name:"col2", title:"Field 2", width:"50%"}
    ]
});
```

DetailViewer

```
DetailViewer.create({
    ID:"detailViewerName",
    data:[
        {col2:"value 1,2", col1:"value 1,1"},
        {col2:"value 2,2", col1:"value 2,1"}
    ],
    cellPadding:3,
    height:300,
    left:0,
    recordsPerBlock:1,
    showBorder:true,
    top:0,
    width:300,
    fields:[
        {name:"col1", title:"Field 1"},
        {name:"col2", title:"Field 2"}
    ]
});
```

TreeGrid

```
TreeGrid.create({
    ID:"treeGridName",
    height:300,
    left:0,
    separateFolders:false,
    showRoot:false,
    sortDirection:false,
    top:0,
    treeFieldTitle:"Name",
    width:300,
    fields:[
        TreeGrid.TREE_FIELD,
        {name:"value", title:"Value"}
    ],
    data:Tree.create({
        constructor:"Tree",
        root:{
            name:"Root/",
            children:[
                {
                    name:"Child 1/",
                    children:[
                        {name:"Leaf 1,1", value:"Leaf 1,1 value"}
                    ]
                }
            ]
        }
    })
});
```

Note

You do not need to include the `TreeGrid.TREE_FIELD` constant to initialize a treeGrid, since the first field definition given will be used as the tree field by default. It is included in the template above for backward compatibility.

Menu

```
Menu.create({
    ID:"menuName",
    title:"Untitled Menu",
    height:20,
    left:0,
    top:0,
    useKeys:true,
    width:100,
    data:[
        {
            checkIf:"true",
            click:"alert('Any function here.')",
            keyTitle:"I",
            keys:["I","i"],
            title:"itemTitle"
        },
        {
            keyTitle:"J",
            keys:["J","j"],
            enabled:false,
            title:"itemTitle2"
        },
        {isSeparator:true},
        {
            click:"alert()",
            title:"itemTitle3",
            submenu:Menu.create({
                constructor:"Menu",
                data:[
                    {
                        checkIf:"true",
                        click:"alert('Any function here.')",
                        keyTitle:"K",
                        keys:["K","k"],
                        title:"subItemTitle"
                    }
                ]
            })
        }
    ]
});
```

Toolbar

```
Toolbar.create({
   ID:"toolbarName",
   left:0,
   top:0,
   height:20,
   width:100,
   membersMargin:0,
   vertical:false,
   buttonDefaults:{height:30, width:50},
   buttons:[
      {click:"alert('Button 1')", title:"Button 1"},
      {click:"alert('Button 2')", title:"Button 2"},
      {click:"alert('Button 3')", title:"Button 3"}
   ]
});
```

Menubar

```
Menubar.create({
   ID:"menubarName",
   left:0,
   top:0,
   width:100,
   membersMargin:0,
   vertical:false,
   menus:[menu1, menu2, menu3]
});
```

XML Widget Templates

The following templates include all the properties and default values in the XML declarative format.

Canvas

```
<Canvas
   ID="canvasName"
   left="0"
   top="0"
   width="100"
   height="100"
   position="absolute"
   visibility="inherit"
   overflow="visible"
   appImgDir=""
   className="normal"
   backgroundColor="gray"
   backgroundImage=""
   backgroundRepeat="repeat"
   margin="0"
   padding="0"
   border=""
   cursor="default"

   canDrag="false"
   dragIntersectStyle="mouse"
   canDragReposition="false"
   dragRepositionCursor="move"
   canDragResize="false"
   dragAppearance="outline"
   canDrop="false"
   canAcceptDrop="false" >

   <contents>Canvas contents</contents>
   <contextMenu>
      <Menu ID="contextMenu" >
         <data>
            <item title="Context Menu Item"/>
         </data>
      </Menu>
   </contextMenu>
</Canvas>
```

Label

```
<Label
   ID="labelName"
   left="0"
   top="0"
   width="100"
   height="100"

   className="normal"
   align="left"
   valign="center"
   padding="0"
   wrap="true"
   backgroundColor="gray"
   backgroundImage="blank.gif" >

   <contents>Label contents</contents>
</Label>
```

Img

```
<Img
   ID="imgName"
   left="0"
   top="0"
   width="100"
   height="100"

   src="blank.gif"
   appImgDir=""

   state=""
   canDragReposition="true"
/>
```

StretchImg

```
<StretchImg
   ID="stretchImgName"
   left="0"
   top="0"
   width="100"
   height="100"

   src="blank.gif"
   appImgDir=""
   imageType="tile"

   vertical="true"
   capSize="2"
   state="" >

   <items>
      <item name="start" width="capSize" height="capSize"/>
      <item name="stretch" width="*" height="*"/>
      <item name="end" width="capSize" height="capSize"/>
   </items>
```

```
</StretchImg>
```

Button

```
<Button
   ID="buttonName"
   left="0"
   top="0"
   width="100"
   height="20"

   title="Untitled Button"

   wrap="false"
   selected="false"
   actionType="button"
   showRollOver="true"
   showDown="true"
   align="center"
   valign="center"
   borderSize="1"
   enabled="true"

   canDrag="false"
   dragAppearance="outline"
   canDrop="false"
>
</Button>
```

Scrollbar

```
<Scrollbar
   ID="scrollbarName"
   left="0"
   top="0"
   width="100"
   height="100"

   vertical="true"
   showCorner="false"
   autoEnable="true"
   scrollTarget=""
/>
```

Progressbar

```
<Progressbar
   ID="progressbarName"
   left="0"
   top="0"
   width="100"
   height="12"
   percentDone="50"
/>
```

DynamicForm

```
<DynamicForm
   ID="dynamicFormName"
   action="#"
   target=""
   method="post"

   left="0"
   top="0"
   width="400"
   height="100" >

<!-- See Form Item templates for various item types -->
   <items>
      <item type="text" name="field1" title="field1 title"/>
      <item type="text" name="field2" title="field2 title"/>
      <item type="submit" title="Submit Form" />
   </items>
   <values field1="field1 value" field2="field2 value"/>
   <errors field1="field1 error" field2="field2 error"/>
</DynamicForm>
```

Form items (all types)

```
<DynamicForm
   ID="dynamicFormName" >

   <items>
      <!-- data items  -->
      <item type="text" name="itemName" title="TextTitle" defaultValue="value"
         showTitle="true" width="150" required="false" length="10"/>
      <item type="password" name="itemName" title="PasswordTitle"
```

```
        defaultValue="value" showTitle="true" width="150" required="false"
        length="6" />
    <item type="upload" name="itemName" title="UploadTitle" showTitle="true"
        width="150" required="false"/>
    <item type="textArea" name="itemName" title="TextAreaTitle"
        defaultValue="value" width="150" height="100" required="false"/>
    <item type="checkbox" name="itemName" title="CheckboxTitle"
        defaultValue="false"/>
    <item type="radio" name="itemName" title="RadioTitle"
        defaultValue="false"/>
    <item type="radioGroup" name="itemName" title="RadioGroupTitle"
        itemHeight="20" vertical="true" defaultValue="false" required="false">
        <valueMap>
            <value ID="value1">Value 1</value>
            <value ID="value2">Value 2</value>
        </valueMap>
    </item>
    <item type="select" name="itemName" title="SelectTitle" multiple="false"
        defaultValue="value" required="false">
        <valueMap>
            <value ID="value1">Value 1</value>
            <value ID="value2">Value 2</value>
        </valueMap>
    </item>
    <item type="selectOther" name="itemName" title="SelectOtherTitle"
        defaultValue="value" required="false">
        <valueMap>
            <value ID="value1">Value 1</value>
            <value ID="value2">Value 2</value>
        </valueMap>
    </item>
    <item type="date" name="itemName" title="DateTitle"
        defaultValue="2002-06-04" required="false" startDay="1" endDay="31"
        startMonth="1" endMonth="12" startYear="1970" endYear="2010" />
    <item type="time" name="itemName" title="TimeTitle"
        defaultValue="13:30:00" width="100" required="false"
        showSeconds="false" show24HourTime="true" />
    <item type="hidden" name="itemName" defaultValue="value"
        required="false" />

<!-- button items -->
    <item type="button" title="ButtonTitle" click="alert(1)"/>
    <item type="submit" title="SubmitTitle"/>
    <item type="reset" title="ResetTitle"/>
    <item type="toolbar" cellSpacing="20">
        <buttons>
            <item type="button" title="Button 1 Title"
                click="alert('Button 1')"/>
            <item type="button" title="Button 2 Title"
                click="alert('Button 2')"/>
        </buttons>
    </item>

<!-- display items -->
    <item type="staticText" name="itemName" title="StaticTextTitle"
        wrap="true" defaultValue="value" outputAsHTML="false"/>
    <item type="blurb" name="itemName" defaultValue="displayText"
        outputAsHTML="false"/>
    <item type="header" name="itemName" defaultValue="headerText"
```

```
            outputAsHTML="false"/>
         <item type="rowSpacer"height="20"/>
         <item type="spacer" width="20" height="20"/>
      </items>

      <values field1="field1 value" field2="field2 value"/>
      <errors field1="field1 error" field2="field2 error"/>
   </DynamicForm>
```

ListGrid

```
   <ListGrid
      ID="listGridName"
      left="0"
      top="0"
      width="300"
      height="300"
      canSort="true" >

      <data>
         <obj col1="value 1,1" col2="value 1,2" enabled="true"/>
         <obj col1="value 2,1" col2="value 2,2" enabled="true"/>
         <obj col1="value 3,1" col2="value 3,2" enabled="true"/>
         <obj col1="value 4,1" col2="value 4,2" enabled="true"/>
         <obj col1="value 5" col2="value 5" isSeparator="true"/>
         <obj col1="value 6,1" col2="value 6,2" enabled="false"/>
         <obj col1="value 7,1" col2="value 7,2" enabled="false"/>
      </data>

      <fields>
         <field name="col1" title="Field 1" width="50%"
            sortDirection="descending"/>
         <field name="col2" title="Field 2" width="50%" align="right"/>
      </fields>
   </ListGrid>
```

DetailViewer

```
<DetailViewer
   ID="detailViewerName"
   left="0"
   top="0"
   width="300"
   height="300"

   recordsPerBlock="1"
   showBorder="true"
   cellPadding="3" >

   <data>
      <obj col1="value 1,1" col2="value 1,2"/>
      <obj col1="value 2,1" col2="value 2,2"/>
   </data>

   <fields>
      <field name="col1" title="Field 1"/>
      <field name="col2" title="Field 2"/>
   </fields>
</DetailViewer>
```

TreeGrid

```
<TreeGrid
   ID="treeGridName"
   left="0"
   top="0"
   width="300"
   height="300"

   treeFieldTitle="Name"
   showRoot="false"
   separateFolders="false"
   sortDirection="descending" >

   <fields>
      <JS>TreeGrid.TREE_FIELD</JS>
      <field name="value" title="Value"/>
   </fields>

   <data>
      <Tree>
         <root name="Root/">
            <children>
               <obj name="Child 1/">
                  <children>
                     <obj name="Leaf 1,1" value="Leaf 1,1 value"/>
                  </children>
               </obj>
            </children>
         </root>
      </Tree>
   </data>
</TreeGrid>
```

Note You do not need to include the `TreeGrid.TREE_FIELD` constant to initialize a treeGrid, since the first field definition given will be used as the tree field by default. It is included in the template above for backward compatibility.

Menu

```
<Menu
   ID="menuName"
   left="0"
   top="0"
   width="150"
   height="20"

   title="Untitled Menu"
   useKeys="true" >

   <data>
      <item title="itemTitle" checkIf="true" click="alert('Any function
         here.')" keyTitle="I">
         <keys>I</keys>
         <keys>i</keys>
      </item>
      <item title="itemTitle2" enabled="false" keyTitle="J">
         <keys>J</keys>
         <keys>j</keys>
      </item>
      <item isSeparator="true"/>
      <item title="itemTitle3" click="alert()">
         <submenu>
            <Menu>
               <data>
                  <item title="subItemTitle" checkIf="true" click="alert('Any
                     function here.')" keyTitle="K">
                     <keys>K</keys>
                     <keys>k</keys>
                  </item>
               </data>
            </Menu>
         </submenu>
      </item>
   </data>
</Menu>
```

Toolbar

```
<Toolbar
   ID="toolbarName"
   left="0"
   top="0"
   width="100"
   height="20"

   membersMargin="0"
   vertical="false" >

   <buttonDefaults width="50" height="30"/>
   <buttons>
      <Button title="Button 1" click="alert('Button 1')"/>
      <Button title="Button 2" click="alert('Button 2')"/>
      <Button title="Button 3" click="alert('Button 3')"/>
   </buttons>
</Toolbar>
```

Menubar

```
<Menubar
   ID="menubarName"
   left="0"
   top="0"
   width="100"

   membersMargin="0"
   vertical="false"

   <menus>
      <menu>menu1</menu>
      <menu>menu2</menu>
      <menu>menu3</menu>
   </menus>
</Menubar>
```

APPENDIX B

Isomorphic SmartClient Styles

Isomorphic SmartClient uses a cascading style sheet (CSS) in each skin to reference widget styles. The following table describes the CSS classes used in `skin_styles.css`.

General styles

CSS Class (style)	Description
`normal`	Text that has no other style specified for it.
`pageHeader`	Page header that appears above component items.
`printPageHeader`	Style applied when the page header is printed as part of a report.

Form styles

CSS Class (style)	Description
`formRow`	Rows of a form (which can contain multiple columns). This CSS class generally only specifies background color. Other attributes could be specified, but since each row contains form cells, it is better to set the style for text to the `formCell` CSS class.
`labelAnchor`	Style of clickable text next to checkboxes.
`formCell`	Form element—in other words, style in which to display inputted data.
`formTitle`	Title that appears next to a form element.
`formError`	Text that appears above a form element after validation attempt results in error(s).
`formTitleError`	Title that appears next to a form element after validation attempt results in error(s).
`formHint`	Instructive text that appears to the right of the form element.
`staticTextItem`	Non-editable text element of a form.
`textItem`	Text for form elements.
`selectItem`	Text for options in a select box.
`buttonItem`	Form buttons.
`headerItem`	Header that appears above a form.

ListGrid styles

CSS Class (style)	Description
`cell`	Standard listGrid cell.
`cellOver`	A listGrid cell on mouseOver.
`cellSelected`	A listGrid cell when selected.
`cellSelectedOver`	A listGrid cell when selected and mouse is over.
`cellDisabled`	A listGrid cell with `enabled:false`.
`cellDark`	Alternate cell style—visible if `alternateRowStyles:true`.
`cellOverDark`	Alternate cell on `mouseOver`.
`cellSelectedDark`	Alternate cell when selected.
`cellSelectedOverDark`	Alternate cell when selected on mouseOver.
`cellDisabledDark`	Alternate cell with `enabled:false`.
`listTable`	Settings for the listGrid table.
`printHeader`	Style for printing listGrid headers.
`printCell`	Style for printing listGrid cells.

DetailViewer styles

CSS Class (style)	Description
`detailLabel`	Field name in a detailViewer. The field name appears to the left of the record(s).
`detail`	Record in a detailViewer.
`detailHeader`	Header that appears above a detailViewer.

Menu styles

CSS Class (style)	Description
`menu`	Menu item.
`menuSelected`	Selected menu item.
`menuOver`	Menu item on mouseOver.
`menuSelectedOver`	Selected menu item on mouseOver.
`menuDisabled`	Menu item with `enabled:false`.
`menuTable`	Table that menu is drawn in.
`menuButtonText`	Text in menu buttons.
`menuButton`	Menu button appearance.
`menuButtonOver`	Menu button on mouseOver.
`menuButtonDown`	Menu button on mouseDown.
`menuButtonSelected`	Menu button when selected.
`menuButtonSelectedDown`	Selected menu button on mouseDown.
`menuButtonSelectedOver`	Selected menu button on mouseOver.
`menuButtonDisabled`	Disabled menu button.

Button styles

CSS Class (style)	Description
`button`	Unselected, enabled button.
`buttonOver`	Unselected enabled button on mouseOver.
`buttonDown`	Button receiving a `click` event.
`buttonSelected`	Selected and enabled button.
`buttonSelectedDown`	Selected button receiving a `click` event.
`buttonSelectedOver`	Selected button on mouseOver.
`buttonSelectedDisabled`	Selected and disabled button.
`buttonDisabled`	Button with `enabled:false`.

INDEX