



LiveCycle Designer Scripting Basics

Adobe® LiveCycle® Designer
Version 8.0

© 2006 Adobe Systems Incorporated. All rights reserved.

Adobe® LiveCycle® Designer 8.0 Scripting Basics for Microsoft® Windows®
Edition 1.0, November 2006

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names and company logos in sample material or sample forms included in this software are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, Adobe logo, Acrobat, LiveCycle, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

JavaScript is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries.

Microsoft and Windows are either trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries.

All other trademarks are the property of their respective owners.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>).

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

Notice to U.S. Government End Users. The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

Preface	6
Introduction	6
Who should read this guide?	6
Purpose and scope	6
Related documentation	7
1 About Scripting in LiveCycle Designer	9
How scripting works.....	9
Objects that support calculations and scripts	10
2 Configuring LiveCycle Designer Workspace for Scripting	11
About the Script Editor.....	11
To show the Script Editor	12
To change from single-line to multiline view	12
To set the default scripting language.....	12
To set the default processing application.....	13
To display Arabic, Hebrew, Thai, and Vietnamese characters.....	13
Using the workspace to debug calculations and scripts	14
3 Creating Calculations and Scripts	16
About creating calculations and scripts.....	16
Naming conventions for form design objects and variables	16
Choosing a scripting language	17
To create a calculation or script	18
To find text or other items	18
To replace text or other items.....	19
To use statement completion to create calculations and scripts	19
To insert object reference syntax automatically.....	20
Determining when to run your calculation or script.....	20
To view scripting events and scripts	21
Determining where to run your calculation or script.....	23
Testing and debugging calculations and scripts.....	23
4 Events	24
About events.....	24
Types of events	24
Process events	24
Interactive events	25
Application events	27
List of events	28
calculate event.....	28
change event.....	29
click event.....	29
docClose event	30
docReady event.....	31
enter event	31
exit event	32

form:ready event	32
full event.....	33
indexChange event	33
initialize event	34
layout:ready event.....	34
mouseDown event	35
mouseenter event.....	35
mouseleave event.....	36
mouseup event	36
postPrint event	37
postSave event.....	37
preOpen event	37
prePrint event.....	38
preSave event.....	38
preSubmit event.....	39
validate event.....	40
5 Scripting Languages.....	41
Differences between FormCalc and JavaScript functions	41
Using FormCalc	47
About using FormCalc.....	47
Using built-in functions	47
To attach a FormCalc function to an object	48
Built-in function syntax.....	48
Creating basic calculations	49
Using JavaScript.....	51
About using JavaScript	51
Creating scripts using JavaScript.....	51
To attach a JavaScript script to an object.....	52
6 Variables.....	53
About variables	53
Naming variables.....	53
To define a text variable.....	53
To view a text variable definition	54
To delete a text variable.....	54
Using variables in calculations and scripts.....	54
7 Referencing Objects in Calculations and Scripts.....	56
About referencing objects in calculations and scripts.....	56
Referencing object properties and values	56
Referencing unnamed and repeated objects	58
Referencing the current object	60
FormCalc reference syntax shortcuts.....	60
8 Creating and Reusing JavaScript Functions	65
About the script object	65
To create a script object.....	65
To add script to a script object	65
To reference JavaScript functions stored in a script object	66

9	Debugging Calculations and Scripts	67
	About debugging calculations and scripts.....	67
	Providing debugging feedback using the messageBox method	67
	To enable the JavaScript Debugger for LiveCycle Designer.....	68
	To prevent the JavaScript Debugger from disappearing in LiveCycle Designer	69
	Executing scripts in the JavaScript Console	69
	Providing debugging feedback using the JavaScript Console.....	70
10	Working with a Host Application	71
	About working with a host application.....	71
	Comparing the host scripting model functionality	72
11	Working with the Event Model	74
	About the event model	74
	Event model properties and methods.....	74
12	Moving from Scripting in Acrobat to LiveCycle Designer	76
	About moving from scripting in Acrobat to LiveCycle Designer	76
	Converting Acrobat forms that contain scripts.....	77
	Using JavaScript objects from Acrobat in LiveCycle Designer	77
	JavaScript objects from Acrobat supported in LiveCycle Designer	78
13	Examples of Common Scripting Tasks	91
	About the scripting examples	91
	Changing the background colors of fields, fillable areas, and subforms	91
	Hiding and showing objects	93
	Changing the visual properties of an object on the client.....	97
	Getting the current or previous value of a drop-down list	100
	Adjusting the height of a field at run time	102
	Setting a field as required at run time	102
	Calculating the field sums	103
	Highlighting fields in response to form filler interaction	104
	Resetting the values of the current subform.....	107
	Changing the presence of a form design object	108
	Using the properties of the instance manager to control subforms	109
	Using the methods of the instance manager to control subforms.....	110
	Using the instance manager to control subforms at run time.....	112
14	Index	117

Preface

Introduction

Welcome to the Adobe® LiveCycle® Designer Scripting Basics guide. This scripting guide is designed to provide you with an overview of how you can use LiveCycle Designer calculations and scripts to develop and enhance forms created in LiveCycle Designer. For example, you can use calculations and scripts to perform the following actions:

- Change the behavior and appearance of objects at run time
- Control the presentation of field values
- Provide interaction with form fillers using dialogs and visual cues
- Automate form filling
- Control the hosting environment
- Interact with Web Services
- Interacting with databases and populate forms with data from data sources

Here you will find detailed information and examples of what the LiveCycle Designer calculation and scripting capabilities are and how to access them, as well as descriptions of the usage of the LiveCycle Designer Script Editor. LiveCycle Designer calculations and scripts are a powerful way to enhance and extend your LiveCycle Designer forms.

Who should read this guide?

This guide is intended for form authors and form developers interested in using calculations and scripts to extend their LiveCycle Designer forms. However, it is assumed that you have a working knowledge of scripting languages, especially JavaScript, as well as object models. You should also be familiar with Adobe Acrobat® Professional or Acrobat Standard, and be comfortable working in a structured XML environment.

Purpose and scope

The purpose of this guide is to provide you the following information:

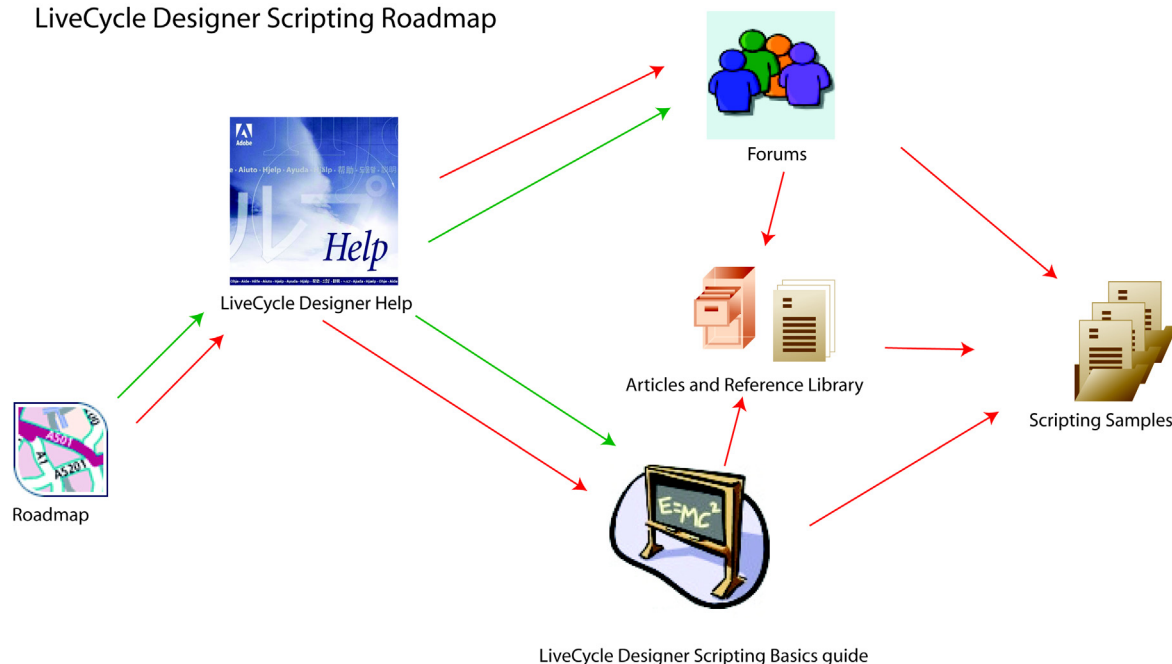
- An introduction to using LiveCycle Designer calculations and scripts to extend your forms
- Easily understood, detailed information and examples of the LiveCycle Designer calculation and scripting features
- References to other resources where you can learn more about LiveCycle Designer scripting and related technologies

After reading this guide, you should be equipped to start using LiveCycle Designer calculations and scripts. During the development process, you will find that the descriptions and examples in this guide will provide you with enough direction and background to enable you to successfully complete your projects.

Related documentation

Adobe has a wide variety of resources dedicated to LiveCycle Designer scripting focused at both the form author and the form developer audiences. The following illustration and section outline the different resources available and where to find them.

LiveCycle Designer Scripting Roadmap



Roadmap

The roadmap document is the starting point for learning about the scripting resources available to you. Follow the paths and discover the different sources of scripting information and how they can help you meet your scripting needs.

LiveCycle Designer Help

The *LiveCycle Designer Help* contains detailed information on using the product, including information on using calculations and scripts, and should be the first place you search for information on any topics related to LiveCycle Designer.

LiveCycle Designer Scripting Basics guide

This guide provides an overview of creating calculations and scripts for use with LiveCycle Designer. This guide is intended to help you create calculations and scripts using FormCalc and JavaScript.

Articles and reference library

The Adobe XML Form Object Model Reference is a detailed reference of the models, objects, properties, and methods that you can use with LiveCycle Designer. This PDF is intended as reference material only, and is not intended to give users information on how to create calculations or scripts.

www.adobe.com/devnet/livecycle/articles/Adobe_XML_Form_Object_Model_Reference.pdf

User Forums

The LiveCycle Designer Forum is a meeting place for professionals who are interested in discussing issues related to LiveCycle Designer. Respond to reader questions, report bugs or issues with the product, or post questions of your own to other form designers and Adobe experts.

www.adobeforums.com

Scripting samples

The scripting samples are working forms or packages that include instructions on how the sample was created, and include any sample data used to create and view the form. New samples are added on an ongoing basis by both Adobe experts and third party organizations.

www.adobe.com/devnet/livecycle/samples.html

As part of the form design process, a form developer can use calculations and scripts to provide a richer user experience. You can add calculations and scripts to most form fields and objects. For example, the following JavaScript script multiplies the values of two numeric fields together and displays the result in a third numeric field:

```
NumericField3.rawValue = NumericField1.rawValue * NumericField2.rawValue;
```

At a more advanced level, you can create your own functions tailored towards your own custom form processing needs.

LiveCycle Designer supports two scripting languages, each geared towards the needs of a particular type of form developer. FormCalc is a straightforward, easy-to-use calculation language that is modelled on common spreadsheet functionality. It includes a variety of built-in functions designed to reduce the amount of time you need to spend developing your form design. JavaScript, a powerful scripting language, provides you with a great deal of flexibility when creating your scripts and allows you to leverage any existing knowledge of the language.

Remember that scripting on a form is entirely optional. You can choose to take advantage of scripting to provide a richer user experience, but many of the most powerful features available during form creation are available in LiveCycle Designer without the use of scripts. However, through scripting, you can manipulate and control almost all aspects of your form design.

How scripting works

LiveCycle Designer scripting uses an event-based model that allows you to alter various aspects of objects on a form at run time. As a form designer, you add scripts to objects based on when you want the script to execute. For example, you might place the following script on the `click` event of a button object so that at run time, when a user clicks the button, a message box appears with a message:

```
xfa.host.messageBox("This is a message for a form filler.", "User Feedback", 3);
```

Scripts associated with a particular event execute whenever that event occurs. Some events can occur multiple times within the same form filling session. For example, the following script adds one to the current value of a numeric field:

```
NumericField1.rawValue = NumericField1.rawValue + 1;
```

If you add this script to the `calculate` event for `NumericField1`, when you open the form for the first time, `NumericField1` displays the value 2. This indicates that the `calculate` event occurred twice in the sequence of events that occurred when the form was opened.

Objects that support calculations and scripts

The following table provides a quick reference of scripting support for the standard objects that are included in the Library palette in LiveCycle Designer.

Objects that support calculations and scripts	Objects that do not support calculations and scripts
Barcodes	Circle
Button	Content Area
Check Box	Line
Date/Time Field	Rectangle
Decimal Field	Image
Document Signature Field	Subform Sets
Drop-Down List	Table sections
Email Submit Button	Text
HTTP Submit Button	
Image Field	
List Box	
Numeric Field	
Paper Forms Barcode	
Password Field	
Print Button	
Radio Button	
Reset Button	
Subform	
Table (including body rows, header rows, and footer rows)	
Text Field	

About the Script Editor

The Script Editor within LiveCycle Designer is where you create, modify, and view the calculations and scripts of a particular form.

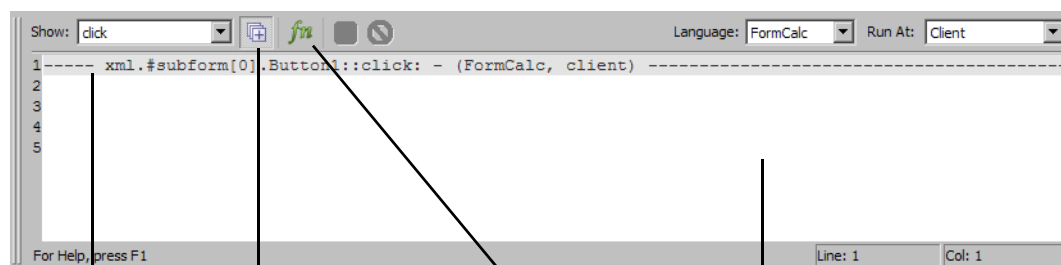
The Script Editor itself has both a single-line view and a multiline view, which you can easily switch between, depending on your current needs. Single-line view is designed to maximize the amount of space dedicated to the Layout Editor and other palettes, whereas multiline view is designed to maximize the amount of space for writing script.

Single-line View



Script editing field

Multiline view




Reference syntax


Show Events for Child
Object button

Functions button


Script editing field


Show Lists all form design events that support user-defined scripting. Any events that do not apply to a particular object appear dimmed. Events that contain a calculation or script display an asterisk (*) beside the name of the event.

Show Events for Child Objects  Displays the event you have currently selected in the Show list for the current object and all of its child objects. If you select the uppermost object in the Hierarchy palette, this option displays the event you have currently selected in the Show list for all objects on your form.

Functions  Displays a list of available built-in FormCalc or JavaScript functions, depending on the scripting language you currently have selected in the Language list.

To place a function onto your script editing field, select a function from the list and press Enter.

Enter Script Source Changes  Saves your script to your form design. Click this button after you have finished writing your script. If you do not save your script in this manner, it may not execute correctly or may be lost.

Cancel Script Source Changes  Undoes any changes you made to a particular calculation or script. This button applies only to the current editing session and does not act as a multiple undo feature.

Language Specifies the scripting language you want to use for the current calculation or script. Two options are available:

- **FormCalc** is a native Adobe calculation language typically used for shorter scripts, such as simple calculations.
- **JavaScript** is a robust and flexible scripting language suitable for more complex scripts.

Run At Specifies where the calculation or script will execute. Three options are available:

- **Client** Calculations and scripts execute while the client application (for example, Acrobat, Adobe Reader, or a web browser) processes the form.
- **Server** Calculations and scripts execute while the server application (for example, LiveCycle Forms) processes the form.
- **Client and server** Calculations and scripts execute while the server application (for example, LiveCycle Forms) processes the form, except in cases where the HTML client application supports client-side scripting. For example, a script that accesses a database to prefill data on a form.

To show the Script Editor

- ❖ Select Window > Script Editor.

Note: You can use the Expand button to quickly dock or undock the Script Editor when it is displayed in the LiveCycle Designer workspace.

To change from single-line to multiline view

- ❖ Drag the Script Editor palette bar until the palette is the required size.

Note: Multiline view adds the All Events and Events with Scripts options to the Show list. The All Events option displays all of the events for a particular form design object, even if the events do not contain any calculations or scripts. The Events with Scripts option displays only those events of a particular object that contain calculations or scripts.

To set the default scripting language

1. Select File > Form Properties.
2. Click the Defaults tab.
3. Select your default scripting language from the Default Language list.

Note: This procedure sets the value of the default scripting language only for the current instance of the form.

To avoid changing the default scripting language each time you create a form, you must modify the corresponding form template file that is used to create a new form design.

► To change the default scripting language for a form template

1. Create a new form design.
2. Select File > Form Properties.

3. Click the Defaults tab.
4. Select your default scripting language from the Default Language list.
5. Make a backup of the original form template file located in the Templates folder where LiveCycle Designer is installed.
6. Save the new form design as a TDS file and overwrite the corresponding form template. For example, save the file as Letter.tds and overwrite the Letter.tds file located in Templates\Blank folder.

To set the default processing application

1. Select File > Form Properties.
2. Click the Defaults tab.
3. Select your default processing application from the Default Run At list.

Note: This procedure only sets the value of the default processing application for the current instance of the form.

To avoid changing the default processing application each time you create a form, you must modify the corresponding form template file that is used to create a new form design.

► To change the default processing application for a form template

1. Create a new form design.
2. Select File > Form Properties.
3. Click the Defaults tab.
4. Select your default processing application from the Default Run At list.
5. Make a backup of the original form template file located in the Templates folder where LiveCycle Designer is installed.
6. Save the new form design as a TDS file and overwrite the corresponding form template. For example, save the file as Letter.tds and overwrite the Letter.tds file located in Templates\Blank folder.

To display Arabic, Hebrew, Thai, and Vietnamese characters

To display Arabic, Hebrew, Thai, and Vietnamese characters in the Script Editor or XML Source tab, you must change the font settings that LiveCycle Designer uses in these tabs. Otherwise, LiveCycle Designer displays rectangles where the language-specific characters should be.

Note: LiveCycle Forms 7.1 does not support form designs that contain Arabic, Hebrew, Thai, and Vietnamese fonts.

1. Select Tools > Options and select Workspace from the list on the left.
2. Select one of the following options:
 - FormCalc Syntax Formatting to set the font in the Script Editor when you use FormCalc

- JavaScript Syntax Formatting to set the font in the Script Editor when you use JavaScript
 - XML Source Syntax Formatting to set the font in the XML Source tab
3. In the Font box, select a font that supports your language. For example, Adobe Arabic supports Arabic, Adobe Hebrew supports Hebrew, Adobe Thai supports Thai, and Myriad Pro and Minion Pro support Vietnamese. You can locate the font you need for your language on the Internet if it is not already on your system.
 4. Click OK.
 5. Click OK to close the Options dialog box.

Using the workspace to debug calculations and scripts

The LiveCycle Designer workspace provides a number of ways to assist you with debugging your calculations and scripts.

The following table provides the location and purpose of some helpful debugging information located on various LiveCycle Designer palettes and tabs.

Workspace location	Purpose
Warning tab in the Report palette	<p>The Warning tab is the first place form developers should check when they encounter scripting errors or other unexpected form behavior. The Warning tab acts as a log of issues reported by LiveCycle Designer and updates when you preview your form design using the Preview PDF tab.</p> <p>Note: For run-time error checking in JavaScript, you should activate the JavaScript Console. For more information, see “To enable the JavaScript Debugger for LiveCycle Designer” on page 68.</p>
Bindings tab in the Report palette	<p>If you include fields on your form design that are bound to a data source, the Binding tab can assist you by displaying lists of fields based on how you defined their data binding. For example, you can list only fields with Global Data Binding or only those with no data binding defined. This feature is especially useful on forms that have a large number of data bound fields.</p>
Hierarchy palette	<p>You can use the Hierarchy palette to determine the location of a form object for a reference syntax. The Hierarchy palette is a graphical representation of the structure of a form. It displays the contents of the Master Pages and Design View tabs.</p> <p>The Hierarchy palette also displays referenced objects under the Referenced Objects node. A <i>referenced object</i> is an object that is added to a form only when it is required. Overflow leader and trailer subforms are examples of referenced objects. Whenever data flows across multiple pages or content areas, the overflow leader and trailer subforms are inserted into the form in the appropriate places.</p>

Workspace location	Purpose
Binding tab in the Object palette	Every LiveCycle Designer object that can be bound to a data source includes a Binding tab in the Object palette. If you bind an object on your form design to a particular data node from your data connection, the Default Binding (Open, Save, Submit) list displays a valid FormCalc reference syntax for accessing that data node. You can use the FormCalc reference syntax in other calculations or scripts for testing purposes.
XML Source tab	<p>The XML Source tab contains the form design's XML code. The XML source code defines all aspects of the form. You can use the XML Source tab to view the XML Form Object Model structure of a form design and to understand the relationships between objects and properties. In the XML source, the XML element names are equivalent to the object names in the XML Form Object Model, and attributes are equivalent to properties.</p> <p>When you select an object in the Hierarchy palette and then click the XML Source tab, the first line of the corresponding element is highlighted. The object name in LiveCycle Designer, as listed in the Hierarchy palette, becomes the value of the <code>name</code> attribute in the XML source.</p> <p>You can set options in the Tools > Options dialog box for viewing the source in the XML Source tab, such as showing or hiding line numbers and setting the syntax coloring.</p> <p>Caution: It is recommended that you do not edit the XML source code directly.</p>

You may also find it useful to change the default options for the Script Editor to make it easier to debug your calculations and scripts. These options are in the Workspace panel of the Options dialog box, which is available by selecting Tools > Options and then selecting Workspace from the list on the left. For example, you can choose to display line numbers in the Script Editor or change the formatting of FormCalc or JavaScript syntax.

About creating calculations and scripts

LiveCycle Designer provides a wide range of calculation and scripting features that you can use to perform a variety of tasks. For example, the following script changes the color of a text field border and the font size of the text field value:

```
TextField1.border.edge.color.value = "255,0,0";  
TextField1.font.typeface = "Courier New";
```

More complex forms can take advantage of scripting to perform data source connectivity and data manipulation at run time. For examples of common scripting tasks, see [“Examples of Common Scripting Tasks” on page 91](#).

Creating calculations and scripts in LiveCycle Designer involves following a general process each time you attach a calculation or script to an object. Although not all aspects of the process are required each time you create a calculation or script, following the process helps to eliminate potential errors and unexpected results.

In general, each time you create a calculation or script, you perform the following tasks:

- Select the object to which you want to attach a calculation or script. Although you can create calculations and scripts that manipulate almost any object on your form design, not all form design objects support form events. For a list of standard objects included in the Library palette in LiveCycle Designer that support scripting, see [“Objects that support calculations and scripts” on page 10](#).
- Select the form event to which you want to assign the calculation or script. The calculation or script associated with an event executes each time the form event triggers. For example, calculations or scripts attached to the `click` event of a button trigger each time the form filler clicks the button while filling the form.
- Choose the scripting language you want to use. Depending on the form you create, you may decide to use only FormCalc, only JavaScript, or a mixture of both languages. You can only use one scripting language on a particular instance of a form object event.
- Choose where the calculation or script runs, either on the client application, such as Acrobat or Adobe Reader, or on the server using LiveCycle Forms.
- Write your calculation or script in the script editing field of the Script Editor.
- Test the calculation or script either by using the Preview PDF tab or in your test environment.

Naming conventions for form design objects and variables

When creating calculations or scripts to enhance your form, you should be aware of the form design object and variable names on your form. In general, your form should avoid using the names of XML Form Object Model properties, methods, and objects for form design objects and variables. Using XML Form Object Model property, method, or object names can result in calculations and scripts not executing properly.

For example, if you create a new text field named `x` within a subform object named `Subform1`, you access the text field object using the following syntax:

```
Subform1.x.[expression]
```

However, subform objects already have an XML Form Object Model property named `x` that represents the horizontal position of the subform on the form design.

To avoid naming conflicts, you need to choose field naming conventions that differ from the XML Form Object Model naming conventions. For example, you can use any of the following field names for the text field in the example above:

- `horizontalValue`
- `x_value`
- `xLetter`
- `hValue`

For more information and a list of the XML Form Object Model property, method, and object names, see the *Adobe XML Form Object Model Reference* at www.adobe.com/devnet/lifecycle/designing_forms.html.

Choosing a scripting language

LiveCycle Designer supports scripting with both FormCalc and JavaScript. Each scripting language presents its advantages that you should be aware of before you write any scripts on your form.

FormCalc is a calculation language that includes a wide range of built-in functions to simplify the most common form functionality. For example, you can use FormCalc financial functions to evaluate the size of a loan payment based on the principle amount, interest rate, and number of payment periods.

JavaScript is a more powerful and diverse scripting language, intended to give you more flexibility and leverage your existing scripting knowledge. For example, you can reuse your existing JavaScript functions in LiveCycle Designer to reduce the amount of new scripting you need to create.

Note: LiveCycle Designer supports JavaScript version 1.6 or earlier.

The following table highlights some of the key differences between FormCalc and JavaScript.

FormCalc	JavaScript
Native Adobe calculation language valid in LiveCycle Designer and LiveCycle Forms	Standard scripting language used in many popular software applications
Shorter scripts (typically one line only)	Potential for longer scripts, if necessary, with the ability to use looping
Contains a variety of useful built-in functions to reduce the amount of scripting required to accomplish common form design tasks	Provides access to the Acrobat Object Model and the JavaScript capabilities of Acrobat
Support for international dates, times, currencies, and number formats	Debugging possible by using the JavaScript debugger in Acrobat
Built-in URL functions for <code>Post</code> , <code>Put</code> , and <code>Get</code> allow web-based interactions	Create custom functions for your own specific needs
Compatible on all LiveCycle Designer and LiveCycle Forms supported platforms	Compatible on all LiveCycle Designer and LiveCycle Forms supported platforms

To create a calculation or script


1. Select an object on your form design that supports events. For example, add a button to a new, blank form.
2. In the Script Editor, from the Show list, select one of the events that apply to the object. The event you choose specifies when the script will execute. If you are writing a calculation or script that affects an object that does not support events, you must add your calculation or script to a form design object that does support form events. For example, using the new button object, select the `click` event in the Show list.
3. In the Language list, select your scripting language. For example, for the new button object, select JavaScript.
4. In the Run At list, select where you want the script to execute. For example, for the new button object, select Client.

You can choose to run calculations or scripts on your client-based application (for example Acrobat or a web browser) or on your server-based process (for example, LiveCycle Forms). When set to Client, processing of calculations and scripts initiates after the form renders. When set to Server, processing of calculations and scripts initiates during the form rendering process. Previewing your form by using the Preview PDF tab simulates opening the form in Acrobat; therefore, scripts set to run at Client or Client and Server execute.

Note: Selecting Client And Server from the Run At list causes a script to execute in either the client application or the server application, depending on which application is used to process the form.

5. In the Script Source field, insert your FormCalc calculation or JavaScript script. You can take advantage of the statement completion functionality of LiveCycle Designer to help you create reference syntaxes for your calculation or script. For example, add the following JavaScript script to the new button object:

```
xfa.host.messageBox("Hello World!", "Creating a new script", 3);
```

6. Click Enter Script Source Changes  to add the script to your form.
7. After you complete your form design, test and debug your calculations and scripts before putting them into production. For example, for the new button object, preview the PDF of the form using the Preview PDF tab. Click the button object to display message box specified in step 5.

For more information about the LiveCycle Designer objects that support scripting, see [“Objects that support calculations and scripts” on page 10](#).

To find text or other items

You can quickly search for every occurrence of a specific word or phrase when you are in the XML Source tab or in the Script Editor.

1. In the XML Source tab or the Script Editor, select Edit > Find or right-click for the context menu.
2. In the Find What box, enter the text that you want to search for.
3. Select any other options that you want.
4. Click Find Next.


To cancel a search in progress, press Esc or select the Cancel button.

Caution: Although it is possible to edit XML source code directly in the XML Source tab, it is recommended that you do not make any changes unless you are familiar with the Adobe XML Forms Architecture. For more information about the XML Forms Architecture, see www.adobe.com/devnet/lifecycle/.

To replace text or other items

You can automatically replace text. For example, you can replace *Corp.* with *Corporation*.

1. In the Script Editor, select Edit > Replace.
2. In the Find What box, enter the text that you want to search for.
3. In the Replace With box, enter the replacement text.
4. Select any other options that you want.
5. Click Find Next, Replace, or Replace All.
6. To cancel a search in progress, press Esc or select the Cancel button.

To replace text that appears in scripts attached to multiple objects on your form, select the root subform of your form (by default: `form1`) and select Show Events for Child Objects  and then perform the procedure above.

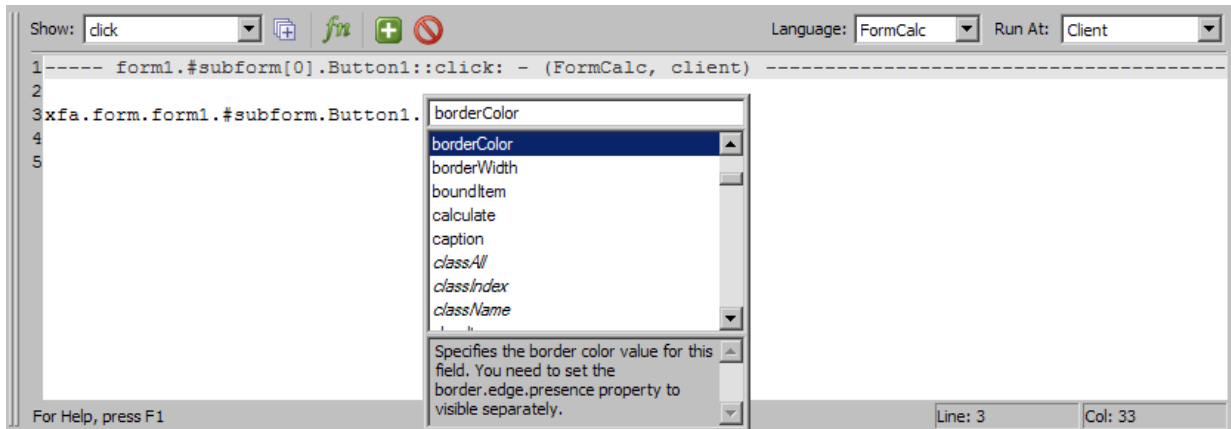
Caution: Although it is possible to edit XML source code directly in the XML Source tab, it is recommended that you do not make any changes unless you are familiar with the Adobe XML Forms Architecture. For more information about the XML Forms Architecture, see www.adobe.com/devnet/lifecycle/.

To use statement completion to create calculations and scripts

The statement completion functionality within the Script Editor lets you build your calculations and scripts interactively.

When writing a calculation or script, each time you enter a period (.) immediately following a form object or property name, the statement completion functionality displays a list of available methods and properties. If the statement completion list does not appear, verify that you have typed the object or property name correctly and that the object is within the scope of the object where you are creating your script. For more information about referencing objects in calculations and scripts, see [“Referencing Objects in Calculations and Scripts” on page 56](#).

1. Type the name of a form design object, property, or a valid FormCalc shortcut, followed by a period.




2. Select the method or property you want to apply for the form design object and continue writing the script. To close the statement completion list without selecting a function, press the Esc key.

The list of available XML Form Object Model properties changes depending on the form design object or property that appears before the period.

Note: The statement completion list appears only when accessing objects, properties, and methods in the XML Form Object Model. It does not appear when working with standard JavaScript objects or methods.

To insert object reference syntax automatically

As an alternative to using the statement completion list to create object reference syntax, you can use the insert object reference syntax feature to automatically add reference syntax to your calculation or script. This feature inputs an abbreviated reference syntax for the object you select from the canvas into the Script Source field of the Script Editor. This reduces the time required to create calculations and scripts and ensures that the reference syntax is accurate.

1. Ensure that the Script Source field of the Script Editor has the focus and the cursor is positioned where you want to insert the object reference.
2. On your form, Ctrl+click the object you want to reference. The cursor changes to  to assist you when selecting an object.

Determining when to run your calculation or script

When creating calculations and scripts, you must associate each entry with a specific form event. Each form event represents a change in the form's state that initiates at a specific time. The change in form state can occur during form rendering on the server by LiveCycle Forms, or on the client by Acrobat or Adobe Reader while a user is filling a form. When a change in the state of the form occurs, any calculations or scripts associated with the event are processed automatically.

The event you use when creating a calculation or script will, to some extent, determine what you must include in your calculation or script. For example, the amount and type of information available on a form may be different depending on the event timing you choose; therefore, a calculation or script that

retrieves a value from a field may have different results if run before instead of after a form filler performs certain actions. For more information about events, see [“Events” on page 24](#).

Depending on the type of form you are creating, some events may never occur. For example, if you are creating a form that has a fixed layout and no interactive objects, then interactive events associated with form filler actions may never occur and, as a result, any scripts associated with those events will not run.

Although LiveCycle Designer includes support for a wide variety of form events, it is possible to accomplish a wide variety of common calculation and scripting tasks by using only a few events that occur at major changes in a form's state, such as the following events:

- **docReady** ●Initiates immediately after the form opens in Acrobat or Adobe Reader® and immediately before the form filler can begin interacting with form objects. This event is the last event to occur before control of the form is given to the form filler.
- **enter** ●Initiates when the form filler changes the focus to a particular field, button, or subform.
- **exit** ●Initiates when the form filler changes the focus from a particular field, button, or subform, to another object.
- **change** ●Initiates when a form filler makes a change to a field value. This event is most commonly used with drop-down lists or list boxes to execute a script when a form filler makes a change to the current value.
- **click** ●Initiates when a form filler clicks a field or button. This event is most commonly used with buttons to execute a script when a form filler clicks the button.

To view scripting events and scripts

The Script Editor, provides several ways to view scripting events for objects in your form, depending on the type of object or objects you select, and the quantity of events you want to display.


Before you begin, you must perform the following actions:

- If the Script Editor is not displayed on the screen, select Window > Script Editor.
- If the Script Editor is not large enough to display more than one line of script at a time, drag its lower line down to increase its size.

► To view a scripting event for a single object in the Script Editor

1. Select an object in your form.
2. In the Show list, select a valid scripting event.

► To view a scripting event for a container object and its children in the Script Editor

1. If it is not already in multiline mode, expand the Script Editor to display multiple lines of script and ensure that the Show Events for Child Objects  option is selected.
2. Select a container object, such as a subform.
3. In the Show list, select a valid scripting event.

The events appear in the script editing field of the Script Editor, separated by the reference syntax for each event. Note that certain events only apply to specific types of objects. When you select a script event, the script editing field of the Script Editor only displays valid instances of the event. For example, if you select a subform that contains a drop-down list and select the `preOpen` event, the Script editor displays a single

entry representing the drop-down list. This is because the `preOpen` event only applies to drop-down lists. Alternatively, selecting the `enter` event displays two entries, one for the drop-down list and one for the subform.


Note: The Show list denotes events that contain scripts using a trailing asterisk (*) after the name of the event. If an event contains a script, when you select the event from the Show list, the source appears in the script editing field of the Script Editor.

► **To view all scripting events for a single object in the Script Editor**

1. Select an object in your form.
2. In the Show list, select All Events.

The events appear in the script editing field of the Script Editor, separated by the reference syntax for each event.

► **To view all scripting events for a container object and its children in the Script Editor**

1. If it is not already in multiline mode, expand the Script Editor to display multiple lines of script and ensure the Show Events for Child Objects  option is selected.
2. Select a container object, such as a subform.
3. In the Show list, select All Events.


The events appear in the script editing field of the Script Editor, separated by the reference syntax for each event.

► **To view all scripts for a single object in the Script Editor**

1. Select an object that has scripts attached.
2. In the Show list, select Events With Scripts.

The scripts appear in the script editing field of the Script Editor, separated by the reference syntax for each event.

► **To view all scripts for a container object and its children in the Script Editor**

1. If it is not already in multiline mode, expand the Script Editor to display multiple lines of script and ensure that the Show Events for Child Objects  option is selected.
2. Select a container object, such as a subform. All events for the container object and any child objects appear in the Script Editor.
3. In the Show list, select All Events.

The scripts appear in the script editing field of the Script Editor, separated by the reference syntax for each event.

Determining where to run your calculation or script

For each calculation and script created in LiveCycle Designer, you must specify the location where you want the calculation or script to run.

Unless you are using server-based processing such as LiveCycle Forms, you should ensure that all of your calculations and scripts are set to run on the client application (for example, on Acrobat or a web browser).

Note: FormCalc calculations and scripts do not work on forms rendered as HTML and are ignored during form filling.

If you are using server-based processing, you can choose between running calculations on the client application, or running them on the server. By choosing to have calculations and scripts run on the server, you are choosing to run the scripts at a specific point during the form-rendering process.

If you choose Client And Server from the Run At list, your calculation or script is available to both client and server-based applications. This option is useful, for example, if you do not know whether your users will have client or server applications when they attempt to use your form. It is also useful if you want certain form objects to behave one way to a client application and another to a server-based application.

Testing and debugging calculations and scripts

After you have created your calculation or script and tested your form, you may discover scripting errors or other unexpected field values as a result of scripting errors.

LiveCycle Designer includes three primary methods for testing and debugging your calculations and scripts:

- Using the LiveCycle Designer workspace palettes. For more information, see [“Using the workspace to debug calculations and scripts” on page 14](#).
- For JavaScript only, using the JavaScript Debugger to assist you in testing your scripts. For more information on using the debugger, see [“To enable the JavaScript Debugger for LiveCycle Designer” on page 68](#).
- Using the host model and event model properties and methods to troubleshoot your form.

The host model and event model provide functionality that lets you interact with either the host application or the various form events. These models are useful for returning information that can assist you in debugging calculation and scripts.

For example, the following script returns a message at run time indicating the name of the event on which the script is placed. This indicates that a particular event has fired:

```
xfa.host.messageBox(xfa.event.name) // FormCalc  
xfa.host.messageBox(xfa.event.name); // JavaScript
```

Another example of using the host model and event model methods is to obtain the value of a field on an interactive form before a user manually changed it. This is useful for observing how the objects on your form design respond to user-entered data:

```
xfa.host.messageBox(xfa.event.prevText) // FormCalc  
xfa.host.messageBox(xfa.event.prevText); // JavaScript
```

About events

Every calculation or script you attach to a form object is associated with a specific event. An *event* is defined as a particular occurrence or action that can change the state of a form and, when the change of state occurs, automatically invoke a calculation or script associated with the event. Events occur at various times, from the beginning of the form rendering process when merging data with a form design, all the way through to a form filler interacting with objects on a form in a client application. By applying calculations and scripts to specific events, you can control every aspect of how you present form objects, as well as form data, and how the objects and data respond to form filler interaction.

A single change of state or form filler action may trigger multiple events. For example, tabbing from the current field to the next field triggers both the `exit` event for the current field and the `enter` event for the next field. If the current and next fields are in different subforms, a total of four events are triggered; namely, `exit` events for the current field and subform, and `enter` events for the next field and subform. In general, each of the different categories of form events follow a predictable ordering.

Types of events

Form events fall into one of the following categories:

Process events This type of event initiates automatically as the result of an internal process or action related to objects on a form. For example, if a form filler clicks a button that adds a new page to the form, the `initialize`, `calculate`, `validate`, and `layout:ready` process events initiate automatically for the new page.

Interactive events This type of event initiates as a direct result of form filler actions. For example, if a form filler moves the pointer over a field on a form, the `mouseenter` event initiates in response to the action.

Application events This type of event initiates as a result of the actions that either a client application or a server application performs. For example, you can create a calculation or script to perform a task immediately after the form is saved by using the `postPrint` event.

Process events

Process events initiate automatically as the result of an internal process or action related to a form or objects on a form. These events initiate immediately following significant form changes; for example, after a form design is merged with data or after the form pagination process finishes. Process events also initiate immediately after interactive events initiate. For example, immediately after any interactive event initiates, the `calculate` event initiates followed by the `validate` event.

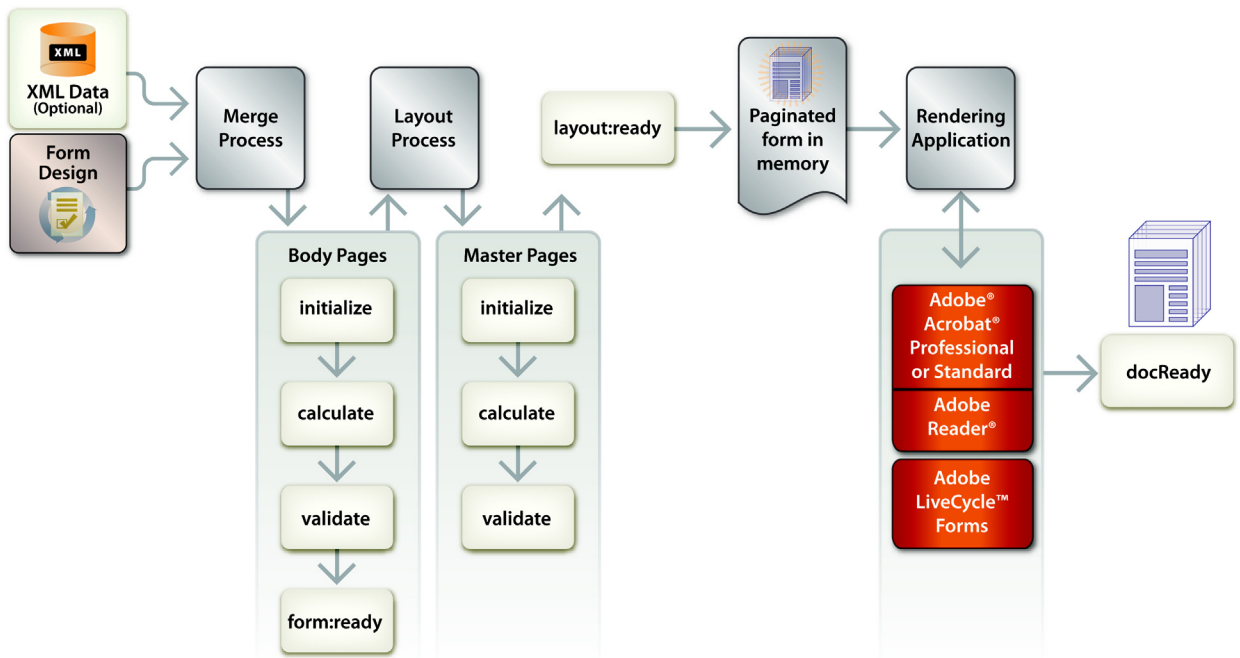
The following list contains the process events, which are available from the Show list in the Script Editor:

- `calculate`
- `form:ready`
- `indexChange`

- initialize
- layout:ready
- validate

Process events can initiate many times as a result of dependencies; that is, actions associated with a single event that ultimately initiates one or more additional events. Using an example of a form filler clicking a button to reveal a previously hidden portion of the form, after the form filler clicks the button, not only does a series of interactive and processing events initiate for the button itself, but a number of process events for the new subform initiates as well.

The following image represents the general flow of events leading up to a PDF form opening in Acrobat or Adobe Reader.



After the form opens in Acrobat or Adobe Reader, these process events may still initiate as the result of changes made to the form. For example, the `calculate`, `validate`, and `layout:ready` events for an object initiate immediately after some interactive events occur; therefore, calculations and scripts attached to the processing events will run multiple times.

Interactive events

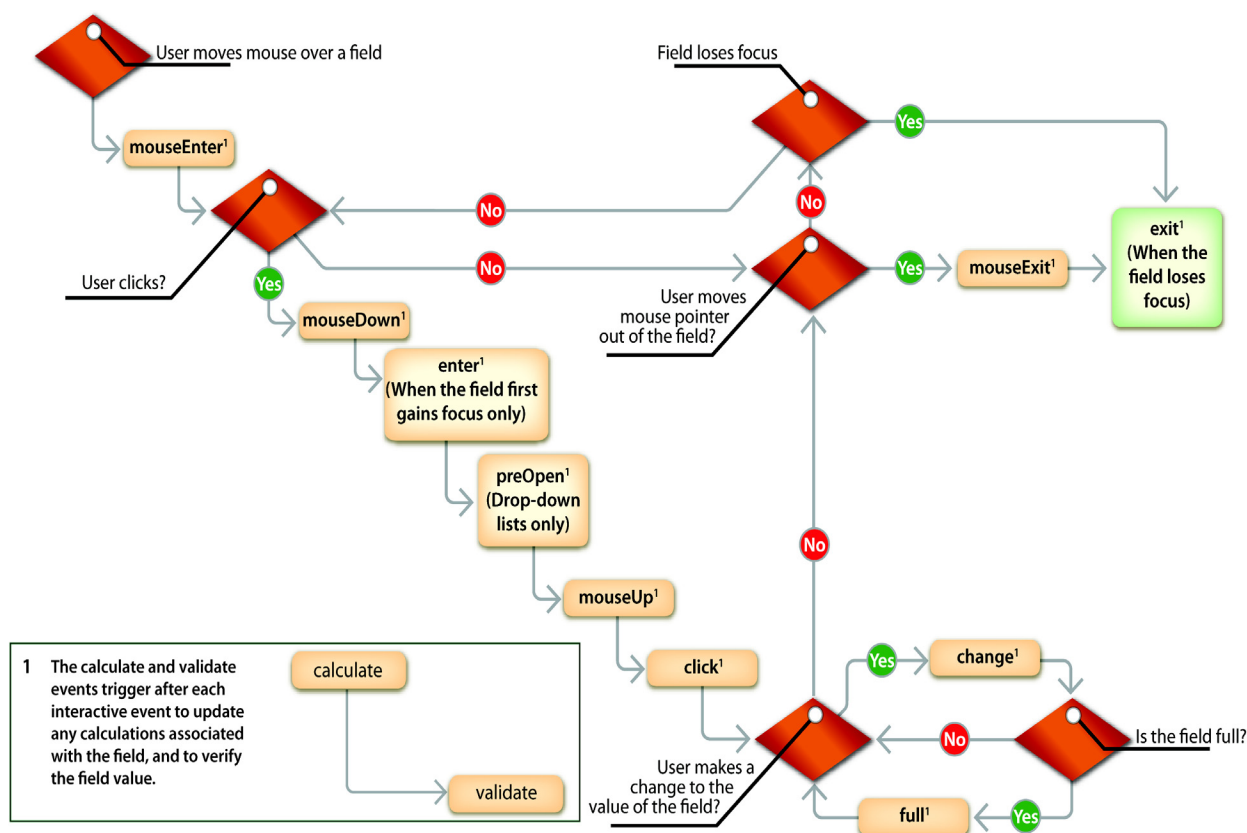
Interactive events initiate as a direct result of form filler actions, which makes these events useful for a variety of calculation and scripting tasks. For example, you can add a script to the `mouseenter` event for a text field that changes the border color of the field to blue and a script to the `mouseleave` event that changes the border color back to the original color. This action creates a highlighting effect when form fillers move the pointer over the field to visually assist them while filling the form. Interactive events are also useful for changing form data in response to a form filler selection. For example, you can add a script to the `change` event for a drop-down list that updates the data values in multiple fields in response to the value the form filler selects in the drop-down list.

The following list contains the interactive events, which are available from the Show list in the Script Editor:

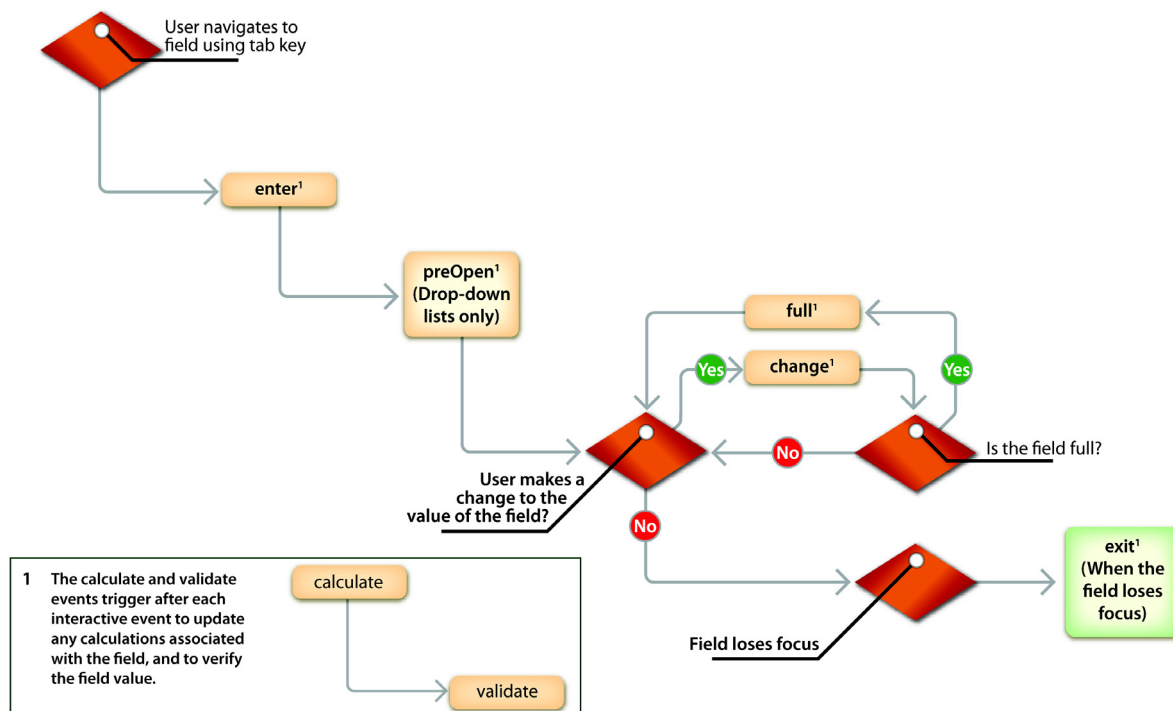
- change
- click
- enter
- exit
- mouseDown
- mouseEnter
- mouseExit
- mouseUp
- preOpen

The following image displays the general flow of events for form fillers who use the mouse to select an object and change its value.

Note: This image provides a general flow of events; however, certain form filler actions and form objects can cause alternate event ordering. For example, if a form filler selects a value from a drop-down list, the `mouseExit` event occur after the `click` event but before the `change` or `full` events. Similarly, if a form filler selects a field, holds down the mouse, and then exits the field while still holding down the mouse button, the `mouseUp` event occurs out of the order described in this image.



The following image displays the general flow of events for form fillers who use the keyboard to select an object and change its value.



Application events

Application events initiate as a result of the actions that either a client application or a server application performs, either in response to a form filler action or as a result of an automated process. For example, if a form filler saves the form in Acrobat or Adobe Reader, the `preSave` event initiates immediately before the save operation, followed by the `calculate`, `validate`, and `layout:ready` events, in order, for all objects on the form. The same event sequence initiates if the form contains a script that programmatically saves the form.

The following list contains the processing events, which are available from the Show list in the Script Editor:

- `docClose`
- `docReady`
- `postPrint`
- `postSave`
- `prePrint`
- `preSave`
- `preSubmit`

Application events do not exist as part of a general flow of events. They are single events corresponding to actions that the client or server application performs.

List of events

calculate event

Description

Initiates in the following situations:

- When your form design and data merge into your finished form.
- When a change occurs to any value that the calculation is dependent on, such as the value of a particular field, unless the form filler has manually overridden the calculated value. As a result, the object will display the return value of the event. The properties for manually overridden fields are located in the Value tab of the Object palette.
- When a field loses focus; for example, when a form filler clicks or uses the Tab key to exit a field.

When using the `calculate` event to perform calculations or scripts, consider the following potential issues:

- Calculations and scripts on the `calculate` event must not make any changes to the structure of the form, except for the form field and data values.
- Content inserted by the `calculate` event must conform to the associated validations for the object; otherwise, validation errors will occur.
- Calculations and scripts must not include an infinite loop because it causes the form to update the value continuously. For example, a script that increments the value of a field as part of a looping expression, such as a `while` or `for` loop, could create an infinite loop.
- The last expression evaluated in the `calculate` event is used to populate the value of the current form object. For example, if the script on the `calculate` event first sets the value of the current field to 500 and then sets the value of another field to 1000, both fields will display the value 1000 at run time. As a result, you need to limit the scripting that you add to the `calculate` event to those that deal specifically with setting the value of the current field.

Type

Processing event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✓

Example

Use the `calculate` event for updating numeric values in fields because this event initiates immediately after most other events. For example, on a purchase order form, you can use the `calculate` event for a field to determine the percentage of sales tax due based on the cost of the order. The calculation will initiate every time a change is made to the values in the form fields, ensuring that the value displayed for the sales tax is always correct.

However, because the `calculate` event can initiate many times, you must ensure that the calculation or script you add to the event will not cause data values to increment unnecessarily. For example, if your sales

tax calculation adds the value of the sales tax to the total cost each time the `calculate` event initiates, the resulting total cost value on your form may be too large.

For a detailed example of using the `calculate` event, see [“Calculating the field sums” on page 103](#).

change event

Description

Initiates when a form filler changes the content of a field by performing one of these actions:

- Types a keystroke providing the field has keyboard focus
- Pastes data into the field
- Makes a selection from a list box or drop-down list
- Selects or deselects a check box
- Changes the setting of a group of radio buttons

This event does not initiate in response to changes in the object values as a result of calculations or scripts, or by the merging of the form design with data.

Type

Interactive event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✓ (Only for drop-down lists)

Example

Use this event for any calculations or scripts that must initiate in response to a form filler changing the value of a field. For example, you can use the `change` event for a drop-down list to highlight specific rows in a table. Using this technique, each time the form filler selects a value in the drop-down list, the corresponding row of the table appears highlighted.

For a detailed example of using the `change` event, see [“Getting the current or previous value of a drop-down list” on page 100](#).

click event

Description

Initiates when a mouse click occurs within the region. When a `click` event initiates for a text or numeric field, calculations or scripts execute immediately. However, the value of the field does not change in response to calculations and scripts until the field loses focus.

Note: You cannot place a calculation or script on the `click` event of a submit button because the calculation or script will override the submission action. Instead, place any calculations and scripts on the `preSubmit` event for a submit button.

For more information about form submission actions, see the *LiveCycle Designer Help*.

Type

Interactive event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✓

Example

Use this event for performing an action as a direct response to a form filler clicking a button or selecting a radio button or check box on a form. For example, you can use the `click` event for a check box to hide and show a field on the form.

For a detailed example of using the `click` event, see [“Changing the visual properties of an object on the client” on page 97](#).

docClose event

Description

Initiates at the very end of processing a form, only if all form validations complete with no errors.

Type

Application event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✗

Example

This event initiates too late to modify a saved form and is intended to provide the ability to generate an exit status or completion message. For example, you can use the `docClose` event to display a message to a form filler indicating that the form is completed.

docReady event

Description

Initiates immediately after the form opens in Acrobat or Adobe Reader.

Type

Application event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✗

Example

This event is the first one that initiates after the form opens in Acrobat or Adobe Reader. Any calculation or scripting tasks that require the full form, or that should only run once when the form filler first opens the form, should use this event. For example, you can use the `docReady` event to check the version of Acrobat or Adobe Reader and return a message to the form filler if the form filler must upgrade the application before filling the form.

enter event

Description

Initiates when a field or subform gains keyboard focus, whether caused by a form filler action (tabbing into a field or clicking in it) or by a script programmatically setting the focus.

Type

Interactive event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✓

Example

You can use this event to provide help text or other messages to a form filler while entering the current field or subform. For example, if a field requires a value in a specific format, or if filling a field requires special instructions, you can use this event to provide a message to the form filler indicating the special needs.

For a detailed example of using the `enter` event, see [“Highlighting fields in response to form filler interaction” on page 104](#).

exit event

Description

Initiates when the field or subform loses keyboard focus, whether caused by a form filler action (tabbing to another field or clicking outside it) or by a script programmatically removing the focus.

Note: If the purpose of your script is to manipulate the value of the current field, you need to consider attaching your script to the `calculate` event.

Type

Interactive event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✓

Example

You can use this event to provide verification of field data as a form filler moves the focus away from a field. For example, if a field requires a value, you can use this event to provide a message to the form filler indicating that the field requires some data before the form can be submitted.

For a detailed example of using the `exit` event, see [“Highlighting fields in response to form filler interaction” on page 104](#).

form:ready event

Description

Initiates after the form design and data are merged, the finished form exists in memory, and the `initialize`, `calculate`, and `validate` events are complete.

Type

Processing event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✗

Example

You can use this event to perform tasks after the form design and data are merged but before the layout is established. For example, you can use this event to customize the ordering or placement of subforms on your form before the form is paginated and rendered.

full event

Description

Initiates when the form filler attempts to enter more than the maximum allowed amount of content into a field. For example, if the Limit Length property for a field is set to 5, and a form filler attempts to enter the string `abcdef`, the `full` event initiates when the form filler types the letter `f`.

Note: The Limit Length property for a field is located in the Field tab in the Object palette.

Type

Interactive event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✗

Example

Use this event to indicate to a form filler that a field has reached its maximum capacity. For example, you can output a message to the form filler indicating that the field is full and provide any steps that should be taken to correct the issue.

indexChange event

Description

Initiates as a result of a subform being inserted, moved, or removed from the form by merging new data with the form or by using scripting.

Note: This event is received only by the subform instances that are controlled by the instance manager; the event is ignored for subform sets.

Type

Processing event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✗

Example

You can use this event to set properties based on the instance value of a particular object. For example, you can use this event to coordinate the shading of alternate rows in a table.

initialize event

Description

Initiates for all objects after the form design is merged with data.

Type

Processing event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✓

Example

You can use this event to perform actions when an object is first created, either as the result of a form filler action or during the form creation process. For example, you can use this event to control settings for new instances of a subform object that a form filler adds to the form by using a button.

layout:ready event

Description

Initiates after the form design and data are merged, the form exists, and the form's layout is applied. At this time, the finished form has not been rendered; therefore, a calculation or script set to run on this event could modify the layout before the forms is rendered. This event also occurs after the form is rendered if a calculation or script changes the data or causes a change to the form in Acrobat or Adobe Reader.

Type

Processing event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✗

Example

You can use this event to perform tasks immediately after the form layout is established. For example, you can use this event to determine the number of pages the form contains.

mouseDown event

Description

Initiates when a form filler presses the mouse button at the same time that the pointer is within a field.

Note: When a `mouseDown` event initiates for a text or numeric field, calculations or scripts run immediately. However, the value of the field does not change in response to calculations and scripts until the field loses focus. When a `mouseDown` event initiates for a document signature field, the event initiates before the signature process begins.

Type

Interactive event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✓

Example

You can use this event to perform an action as a direct response to a form filler clicking a button, or selecting a radio button or check box on a form. For example, you can use the `mouseDown` event for a check box to hide and show a field on the form. This event is conceptually similar to the `click` event and has a similar purpose.

mouseenter event

Description

Initiates when the form filler moves the pointer into the area of the field, without necessarily pressing the mouse button. This event is not initiated when the pointer moves into the field for a different reason; for example, because an overlapping window closes.

Type

Interactive event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✗

Example

You can use this event to provide visual feedback to a form filler in conjunction with the `mouseExit` event. For example, you can use this event to change the border or background color of an object to help visually indicate to form fillers that they are working in a specific field.

For a detailed example of using the `mouseenter` event, see [“Highlighting fields in response to form filler interaction” on page 104](#).

mouseExit event

Description

Initiates when a form filler moves the pointer out of the field, even if the form filler is pressing the mouse button. It is not initiated when the pointer moves out of the field for a different reason; for example, because an overlapping window opens.

Type

Interactive event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✗

Example

You can use this event to provide visual feedback to a form filler in conjunction with the `mouseEnter` event. For example, you can use this event to return the border or background color of an object to its original value to help visually indicate to form fillers that they are no longer working in a specific field.

For a detailed example of using the `mouseExit` event, see [“Highlighting fields in response to form filler interaction” on page 104](#).

mouseUp event

Description

Initiates when a form filler releases the mouse button at the same time that the pointer is within a field.

Note: When a `mouseUp` event occurs for a text or numeric field, calculations or scripts run immediately. However, the value of the field does not change in response to calculations and scripts until the field loses focus.

Type

Interactive event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✓

Example

You can use this event to perform actions as a direct response to a form filler clicking a button, or selecting a radio button or check box on a form. For example, you can use the `mouseUp` event for a check box to hide and show a field on the form. This event is conceptually similar to the `click` event and has a similar purpose.

postPrint event

Description

Initiates immediately after the rendered form is sent to the printer, spooler, or output destination.

Type

Application event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✗

Example

You can use this event to display information messages to the form filler after the form is printed. For example, you can create a script on the `postPrint` event to remind form fillers what steps they need to take to submit the form by hand.

postSave event

Description

Initiates immediately after a form filler saves a form in PDF or XDP format. This event does not initiate when you export a subset of the form (for example, only form data) to XDP.

Type

Application event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✗

Example

You can use this event to display information messages to the form filler after the form data is saved. For example, you can create a script on the `postSave` event to remind form fillers how much time remains for them to successfully complete and submit the form.

preOpen event

Description

Initiates when a form filler performs an action that causes the drop-down list to appear, such as clicking the arrow icon on the drop-down list or by tabbing into the drop-down list and using the down arrow. This event initiates before the contents of the drop-down list are displayed.

Note: This event applies only to the Drop-down List object.

Type

Interactive event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✗

Example

You can use this event to control the loading of large numbers of list items. For example, you can use this event to load a fixed number of records from a data source into a drop-down list. This improves the performance of the form for the form filler at run time.

prePrint event

Description

Initiates immediately before the process of rendering a form for printing begins. You cannot cancel printing using this event.

Caution: Avoid using this event to hide or show form objects. For example, if a form filler has already digitally signed the form, using this event to hide all button objects prior to printing will impact the state of the signature.

Type

Application event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✓

Example

You can use this event to change the presence of an object to prevent it from printing. For example, you can use this event to hide text or instructions intended for the form filler to use while filling the form online.

preSave event

Description

Initiates immediately before form data is saved in PDF or XDP format. This event does not initiate when the form data or another subset of the form is exported to XDP.

Type

Application event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✓

Example

You can use this event to change form data immediately before the data is saved. For example, you can create a script on the `preSave` event to scan the data and display a reminder message to the form filler if certain required fields remain empty.

preSubmit event

Description

Initiates when a form submits data to the host through the HTTP protocol. At this point, the data is organized into a data set but has not been sent to the host. Calculations and scripts associated with this event can examine and alter the data prior to the form submission. If the calculation or script is set to run on the server, the form sends the data to the server indicating that it should run the calculation or script before performing any additional processing.

Note: This event does not distinguish between submissions initiated by instances of clicking buttons or to different URLs. Any script that needs to make these distinctions must include code to determine which button was clicked. In general, the `preSubmit` event is conceptually similar to the `preSave` event and serves a similar purpose.

Type

Application event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✓ (Only for submit buttons)

Example

You can use this event to change form data immediately before the data is submitted. For example, you can create a script on the `preSubmit` event to scan the amount of data and display a message to the form filler estimating how long the data submission may take.

validate event

Description

Initiates when the form design and data merge to create your form and when a field loses focus; for example, when a form filler clicks or uses the Tab key to exit a field. This event initiates again each time the value of a field changes. Calculations and scripts placed on the `validate` event provide a method to perform validations that are more specific than those available through the Value tab of the Object palette.

Calculations and scripts on the `validate` event are required to return `true` or `false` (expressed in a format appropriate to the scripting language) corresponding to a validation that succeeds or fails, and must not affect the overall form structure of form values. In addition, calculations and scripts should not attempt to provide feedback to a form filler because that form filler may not be using the form in a client application such as Acrobat.

Note: Because validations are performed against the content of the form, they cannot be used to verify presentation formatting caused by field patterns.

Type

Processing event

Support

Client application	Availability
Acrobat and Adobe Reader	✓
HTML browser	✓

Example

You can use this event to verify object values, particularly in situations where object data must conform to specific rules. For example, you can create a script on the `validate` event to verify that a total cost field on a purchase order form does not have a negative value.

For a detailed example of using the `validate` event, see [“Setting a field as required at run time” on page 102](#).

Differences between FormCalc and JavaScript functions

Although FormCalc and JavaScript are geared towards two different types of users, there is some overlap between the types of built-in functions they offer. The following table lists all available FormCalc functions and lists whether a comparable function exists within JavaScript.

For more information about FormCalc functions and their parameters, see [“Built-in function syntax” on page 48](#).

FormCalc function	Description	JavaScript method equivalent
<code>Abs (n1)</code>	Returns the absolute value of a numeric value or expression.	<code>Math.abs (n1)</code>
<code>Apr (n1, n2, n3)</code>	Returns the annual percentage rate for a loan.	None
<code>At (s1, s2)</code>	Locates the starting character position of a string within another string.	<code>String.search (s1)</code>
<code>Avg (n1 [, n2...])</code>	Evaluates a set of number values and/or expressions and returns the average of the non-null elements contained within that set.	None
<code>Ceil (n1)</code>	Returns the whole number greater than or equal to a given number.	<code>Math.ceil (n1)</code>
<code>Choose (n1, s1 [, s2...])</code>	Selects a value from a given set of parameters.	None
<code>Concat (s1 [, s2...])</code>	Returns the concatenation of two or more strings.	<code>String.concat (s1, s2 [, s3 ...])</code>
<code>Count (n1 [, n2...])</code>	Evaluates a set of values and/or expressions and returns the number of non-null elements contained within the set.	None

FormCalc function	Description	JavaScript method equivalent
<code>CTerm(n1, n2, n3)</code>	Returns the number of periods needed for an investment earning a fixed, but compounded, interest rate to grow to a future value.	None
<code>Date()</code>	Returns the current system date as the number of days since the epoch.	<code>Date.getDate()</code> Note: The JavaScript Date object does not use the epoch as a reference point.
<code>Date2Num(d1 [, f1 [, k1]])</code>	Returns the number of days since the epoch, given a date string.	The JavaScript Date object does not use the epoch as a reference point.
<code>DateFmt([n1 [, k1]])</code>	Returns a date format string, given a date format style.	None
<code>Decode(s1 [, s2])</code>	Returns the decoded version of a given string.	Partial support JavaScript only supports URL encoded values that contain no escape characters.
<code>Encode(s1 [, s2])</code>	Returns the encoded version of a given string.	Partial support JavaScript only supports URL encoded values that contain no escape characters.
<code>Eval()</code>	Returns the value of a given form calculation.	<code>eval(s1)</code>
<code>Exists(v1)</code>	Determines whether the given parameter is a valid reference syntax to an existing object.	None
<code>Floor(n1)</code>	Returns the largest whole number that is less than or equal to the given value.	<code>Math.floor(n1)</code>
<code>Format(s1, s2)</code>	Formats the given data according to the specified picture format string.	None

FormCalc function	Description	JavaScript method equivalent
<code>FV(n1, n2, n3)</code>	Returns the future value of consistent payment amounts made at regular intervals at a constant interest rate.	None
<code>Get(s1)</code>	Downloads the contents of the given URL.	None
<code>HasValue(v1)</code>	Determines whether the given parameter is a valid reference syntax with a non-null, non-empty, or non-blank value.	None
<code>IPmt(n1, n2, n3, n4, n5)</code>	Returns the amount of interest paid on a loan over a set period of time.	None
<code>IsoDate2Num(d1)</code>	Returns the number of days since the epoch, given an valid date string.	None
<code>IsoTime2Num(d1)</code>	Returns the number of milliseconds since the epoch, given a valid time string.	None
<code>Left(s1, n1)</code>	Extracts a specified number of characters from a string, starting with the first character on the left.	<code>String.substring(n1, n2)</code>
<code>Len(s1)</code>	Returns the number of characters in a given string.	<code>String.length</code>
<code>LocalDateFmt([n1[, k1]])</code>	Returns a localized date format string, given a date format style.	None
<code>LocalTimeFmt([n1[, k1]])</code>	Returns a localized time format string, given a time format style.	None
<code>Lower(s1[, k1])</code>	Converts all uppercase characters within a specified string to lowercase characters.	<code>String.toLowerCase(s1)</code>

FormCalc function	Description	JavaScript method equivalent
<code>Ltrim(s1)</code>	Returns a string with all leading white space characters removed.	None Note: You can use JavaScript regular expressions to perform this operation.
<code>Max(n1 [, n2...])</code>	Returns the maximum value of the non-null elements in the given set of numbers.	<code>Math.max(n1, n2)</code>
<code>Min(n1 [, n2...])</code>	Returns the minimum value of the non-null elements of the given set of numbers.	<code>Math.min(n1, n2)</code>
<code>Mod(n1, n2)</code>	Returns the modulus of one number divided by another.	Use the modulo (%) operator.
<code>NPV(n1, n2 [, ...])</code>	Returns the net present value of an investment based on a discount rate and a series of periodic future cash flows.	None
<code>Num2Date(n1[, f1 [, k1]])</code>	Returns a date string, given a number of days since the epoch.	None
<code>Num2GMTTime(n1 [,f1 [, k1]])</code>	Returns a GMT time string, given a number of milliseconds from the epoch.	None
<code>Num2Time(n1 [,f1 [, k1]])</code>	Returns a time string, given a number of milliseconds from the epoch.	None
<code>Oneof(s1, s2 [, s3...])</code>	Returns true (1) if a value is in a given set, and false (0) if it is not.	None This function is similar to the <code>String.search(s1)</code> method and <code>String.match(expression)</code> method.
<code>Parse(s1, s2)</code>	Analyzes the given data according to the given picture format.	None

FormCalc function	Description	JavaScript method equivalent
<code>Pmt (n1, n2, n3)</code>	Returns the payment for a loan based on constant payments and a constant interest rate.	None
<code>Post (s1, s2 [, s3 [, s4 [, s5]]])</code>	Posts the given data to the specified URL.	None
<code>PPmt (n1, n2, n3, n4, n5)</code>	Returns the amount of principal paid on a loan over a period of time.	None
<code>Put (s1, s2 [, s3])</code>	Uploads the given data to the specified URL.	None
<code>PV (n1, n2, n3)</code>	Returns the present value of an investment of periodic constant payments at a constant interest rate.	None
<code>Rate (n1, n2, n3)</code>	Returns the compound interest rate per period required for an investment to grow from present to future value in a given period.	None
<code>Ref ()</code>	Returns a reference to an existing object.	None
<code>Replace (s1, s2 [, s3])</code>	Replaces all occurrences of one string with another within a specified string.	<code>String.replace (s1, s2)</code>
<code>Right (s1, n1)</code>	Extracts a number of characters from a given string, beginning with the last character on the right.	<code>String.substring (n1, n2)</code>
<code>Round (n1 [, n2])</code>	Evaluates a given numeric value or expression and returns a number rounded to the given number of decimal places.	<code>Math.round (n1)</code>
<code>Rtrim (s1)</code>	Returns a string with all trailing white space characters removed.	None Note: You can use JavaScript regular expressions to perform this operation.

FormCalc function	Description	JavaScript method equivalent
<code>Space (n1)</code>	Returns a string consisting of a given number of blank spaces.	None
<code>Str (n1 [, n2 [, n3]])</code>	Converts a number to a character string. FormCalc formats the result to the specified width and rounds to the specified number of decimal places.	<code>String (n1)</code> or <code>Number.toString (radix)</code>
<code>Stuff (s1, n1, n2 [, s2])</code>	Inserts a string into another string.	None
<code>Substr (s1, n1, n2)</code>	Extracts a portion of a given string.	<code>String.substring (n1, n2)</code>
<code>Sum (n1 [, n2...])</code>	Returns the sum of the non-null elements of a given set of numbers.	None
<code>Term (n1, n2, n3)</code>	Returns the number of periods needed to reach a given future value from periodic constant payments into an interest-bearing account.	None
<code>Time ()</code>	Returns the current system time as the number of milliseconds since the epoch.	<code>Date.getTime ()</code> Note: The JavaScript Date object does not use the epoch as a reference point.
<code>Time2Num (d1 [, f1 [, k1]])</code>	Returns the number of milliseconds since the epoch, given a time string.	None
<code>TimeFmt ([n1 [, k1]])</code>	Returns a time format, given a time format style.	None
<code>UnitType (s1)</code>	Returns the units of a unitspan. A unitspan is a string consisting of a number followed by a unit name.	None
<code>UnitValue (s1 [, s2])</code>	Returns the numeric value of a measurement with its associated unitspan, after an optional unit conversion.	None

FormCalc function	Description	JavaScript method equivalent
<code>Upper(s1 [, k1])</code>	Converts all lowercase characters within a string to uppercase.	<code>String.toUpperCase()</code>
<code>Uuid(n1)</code>	Returns a Universally Unique Identifier (UUID) string to use as an identification method.	None
<code>Within(s1, s2, s3)</code>	Returns true (1) if the test value is within a given range, and false (0) if it is not.	<code>String.search(s1)</code>
<code>WordNum(n1 [, n2 [, k1]])</code>	Returns the English text equivalent of a given number.	None

Using FormCalc

About using FormCalc

FormCalc is a simple yet powerful calculation language modeled on common spreadsheet software. Its purpose is to facilitate fast and efficient form design without requiring a knowledge of traditional scripting techniques or languages. With the use of a few of the built-in functions, users new to FormCalc can expect to quickly create forms that save end users from performing time-consuming calculations, validations, and other verifications. In this manner you can create a basic set of rules for the form design that allows the resulting form to react according to the data it comes into contact with.

Within LiveCycle Designer, FormCalc is the default scripting language in all scripting locations with JavaScript as the alternative. For information on setting your default scripting language, see [“About the Script Editor” on page 11](#).

Caution: If you are developing forms for use with a server-based process (for example, using LiveCycle Forms), with the intent of rendering your forms in HTML, you should develop your calculations and scripts in JavaScript. FormCalc calculations are not valid in HTML browsers, and are removed prior to the form being rendered in HTML.

FormCalc treats each new line in the Script Editor as a new expression to evaluate.



Using built-in functions

The built-in functions that make up FormCalc cover a wide range of areas, including mathematics, dates and times, strings, finance, logic, and the web. These areas represent the types of functionality that typically occur in forms, and the purpose of the functions is to provide quick and easy manipulation of form data in a useful way.

At the most basic level, a calculation can consist of only a single FormCalc function. However, a single FormCalc function can make use of other FormCalc functions as parameters.

To attach a FormCalc function to an object

You can add a FormCalc function to any form design object that allows calculations and scripts, with the exception of the script object.

1. Make sure that you have the multiline version of the Script Editor showing on the LiveCycle Designer workspace.
2. Select a field on your form.
3. In the Show list, select the calculate event.
4. Click Functions  or F10 to display a list of FormCalc functions.
5. Select the desired function and press Enter.
6. Replace the default function syntax notation with your own set of values.
7. Click Enter Script Source Changes  to add the FormCalc calculation to your form.

Built-in function syntax

Each FormCalc function uses a specific syntax notation that you must follow in order for the function to execute correctly. The table below describes, very generally, the pieces of syntax notation.

Syntax Notation	Replacement Values
d	A valid date string (for example, 03/15/1996)
f	A valid date format string (for example, MM/DD/YYYY)
k	A valid locale identifier (for example, fr_FR)
n	A valid numeric value. Note that the range of valid values varies from function to function.
s	A valid unit of measurement (for example, "in" for inches).
v	A valid reference syntax.
n1, n2, n3	All values are required.
[[n [, k]]]	No values are required, but you can choose to specify just n, or both n and k.
n1 [, n2 ...]	n1 is required, but you can choose to specify an unlimited number of additional values.
d [, f [, k]]	d is required, but you can choose to also specify f or both f and k.

For more information and examples of valid syntax notation values, see the appropriate FormCalc functions in the *LiveCycle Designer Help*.

Creating basic calculations

About basic calculations

Simple expressions are the most basic instances of scripting. These expressions do not involve using FormCalc built-in functions and are never more than a single line in size. You must add simple expressions to the calculate event of a particular field or object in order for the value of the expression to output onto your form.

Examples of basic calculations

These are all examples of simple expressions:

```
2
"abc"
2 - 3 * 10 / 2 + 7
```

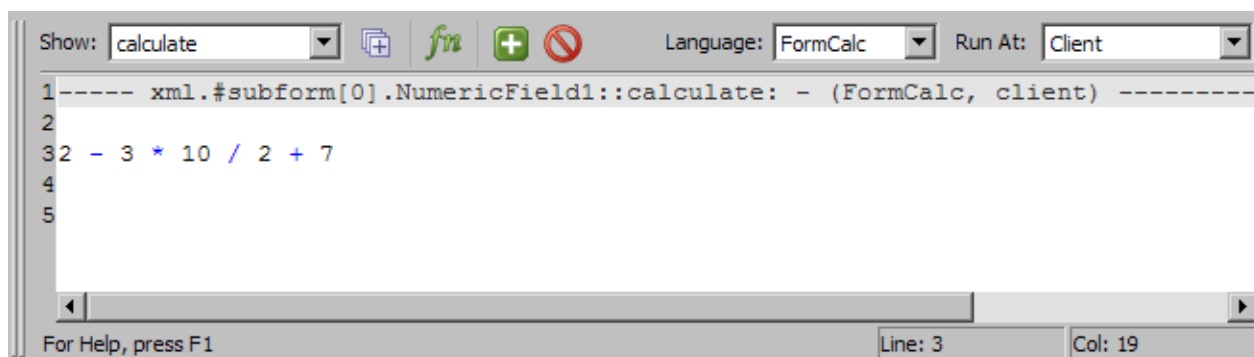
Each simple expression evaluates to a single value by following a traditional order of operations, even if that order is not always obvious from the expression syntax. For example, the following sets of expressions produce equivalent results:

Expression	Equivalent to	Result
"abc"	"abc"	abc
2 - 3 * 10 / 2 + 7	2 - (3 * 10 / 2) + 7	-6
(10 + 2) * (5 + 4)	(10 + 2) * (5 + 4)	108
0 and 1 or 2 > 1	(0 and 1) or (2 >1)	1 (true)
2 < 3 not 1 == 1	(2 < 3) not (1 == 1)	0 (false)

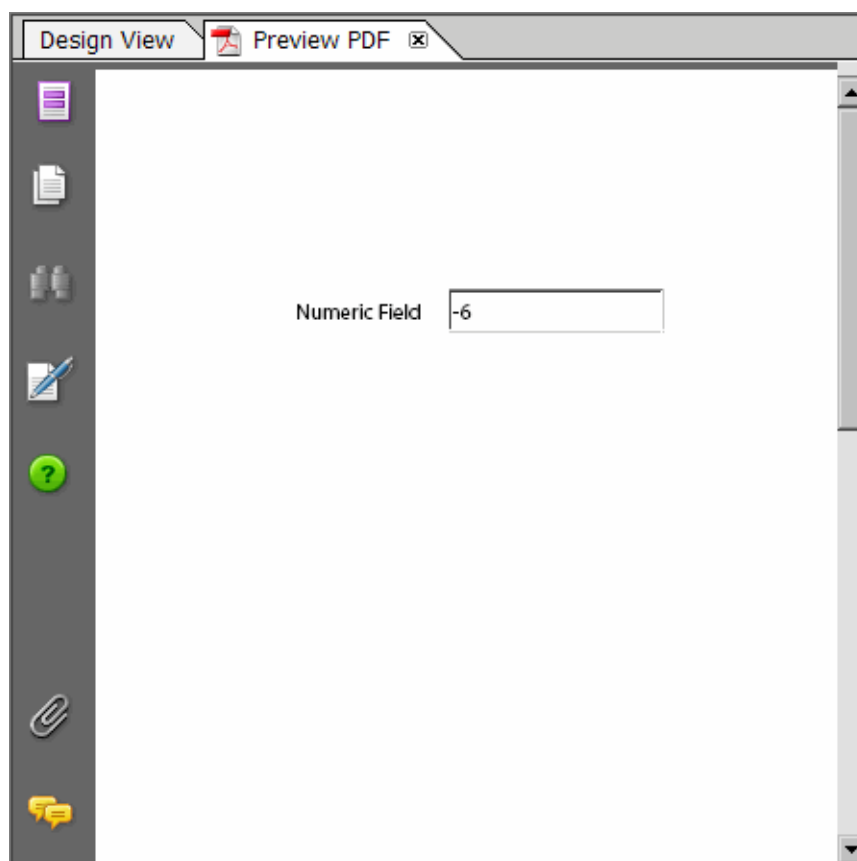
As implied in the previous table, all FormCalc operators carry a certain precedence when they appear within expressions. The following table illustrates this operator hierarchy.

Precedence	Operator
Highest	=
	(Unary) -, +, not
	*, /
	+, -
	<, <=, >, >=, lt, le, gt, ge
	==, <>, eq, ne
	&, and
Lowest	, or

All of the previous examples are valid, simple expressions that you can add to a form field or object that will accept calculations and scripts. For example, if you create a new form in LiveCycle Designer with a single numeric field, add the following calculation to the calculate event in the Script Editor.



Then, when you click the Preview tab to view the completed form, the value of the simple expression appears in the text field.



If the value does not appear in the preview, ensure that your simple expression appears in the calculate event of the form design object. You need to also ensure that you installed LiveCycle Designer and Acrobat correctly.

Using JavaScript

About using JavaScript

To allow form designers more flexibility and scripting power, LiveCycle Designer supports the use of JavaScript version 1.6 or earlier in all situations that support scripting.

Form developers familiar with JavaScript will be able to take their existing expertise and apply it directly to LiveCycle Designer. LiveCycle Designer provides a number of properties and methods that enhance JavaScript to allow you to access field and object values. These properties and methods combine with the LiveCycle Designer reference syntax to provide you with easy manipulation of form values and data.

Note: The Script Editor does not provide syntax error checking for scripts created using JavaScript. In addition, statement completion options do not appear for standard JavaScript objects or methods.

Creating scripts using JavaScript

Creating scripts in LiveCycle Designer using JavaScript is very similar to creating JavaScript in other applications. You can take advantage of previous knowledge of JavaScript concepts, reuse JavaScript functions using the LiveCycle Designer script object, and take advantage of JavaScript language functionality.

However, it is important to note that while previous JavaScript knowledge is transferable, to effectively use JavaScript on your form design you must understand how to construct LiveCycle Designer reference syntax. Specifically, you must be able to correctly use the XML Form Object Model reference syntax to access objects on your form design.


The following table outlines the key concepts for developing scripts in JavaScript for LiveCycle Designer, as well as where to find more information on each concept within the *LiveCycle Designer Help*.

Key concept	For more information see...
Creating references to object properties and values, including using the <code>resolveNode</code> method.	"Referencing object properties and values" on page 56 "To use statement completion to create calculations and scripts" on page 19
Using the host and event models to test and debug your form.	"Testing and debugging calculations and scripts" on page 23 "Referencing object properties and values" on page 56
Using a script object to reuse your existing JavaScript functions.	"About the script object" on page 65


In addition to the resources available in the *LiveCycle Designer Help*, the LiveCycle Developer Center contains extensive scripting resources and documentation. The developer center is located at www.adobe.com/devnet/livecycle/.

To attach a JavaScript script to an object

You can add a JavaScript script to any form design object that allows calculations and scripts, including the script object.

1. Make sure that you have the multiline version of the Script Editor showing on the LiveCycle Designer workspace.
2. Select a field on your form. For example, add a new text field to your form design.
3. In the Show list, select a valid event. For example, using the new text field, select the `docReady` event.
4. In the Run At list, select where you want the script to execute. For example, for the new text field, select Client.
5. Click Functions  or F10 to display a list of JavaScript functions, select the desired function, and press Enter. Replace the default function syntax notation with your own set of values. Alternatively, you can create your own script manually in the Script Source field of the Script Editor. For example, in the new text field, add the following JavaScript to the Script Source field:

```
this.border.fill.color.value = "255,0,0";
```

6. Click Enter Script Source Changes  to add the JavaScript script to your form.
7. Test the script by previewing the form using the Preview PDF tab. For example, for the new button object, when the form is displayed in the Preview PDF tab, the text field should appear red.

About variables

You can define form variables in LiveCycle Designer to store specific information in a central, accessible location. A *variable* typically acts as a placeholder for text that you might have to change in the future. For example, a variable can store the text of a message box title. When the text needs to change, all you have to do is open the affected form or template and update the text once through the variable definition. LiveCycle Designer automatically propagates the new text across all instances of the inserted variable.

Keep in mind that form variables are defined outside of the Script Editor, and are accessible by scripts on all objects on a form, as opposed to scripting variables that you create in a specific FormCalc or JavaScript script.

You can create, view, and delete variables without using scripting. However, you must use scripting to access the values stored by variables and manipulate them, or to apply the values to objects on your form.

Note: Form variable values reset each time you open a form.

Before you create a variable, decide the name of the variable and the text that it will contain. Variable definitions are saved with the form or template.


Naming variables

At run time, naming conflicts occur when the names of variables are identical to those used as XML Form Object Model properties, methods, or form design field names. These conflicts can cause scripts to return unexpected values; therefore, it is important to give each variable a unique name. Here a couple of examples:

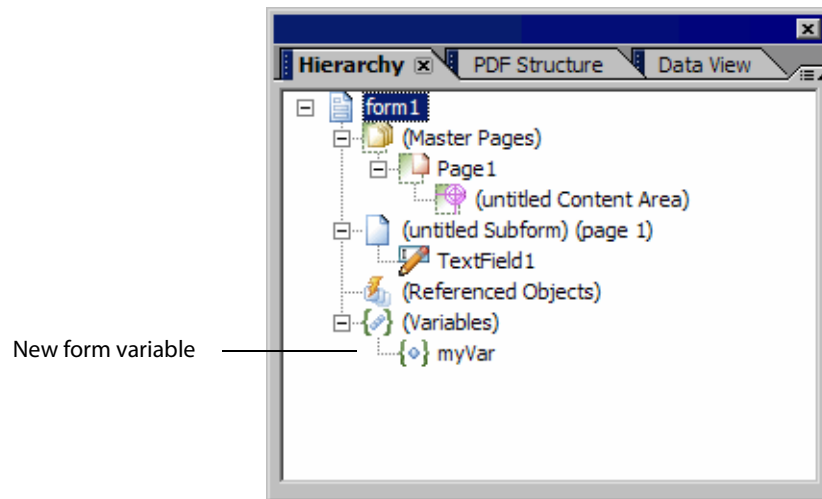
- Use the variable name `fieldWidth` and `fieldHeight` instead of `x` and `y`.
- Use the form design object name `clientName` instead of `name`.

Note: Variable names are case-sensitive and should not contain spaces.

To define a text variable

1. Select File > Form Properties.
2. In the Variables tab, click New (Insert) .
3. In the Variables list, type a unique name for the variable and press Enter. Variable names are case-sensitive and should not contain spaces.
4. Click once in the box to the right and type the text you want to associate with the variable.


The variable appears in the Hierarchy palette at the form level.



To view a text variable definition

1. Select File > Form Properties.
2. Click the Variables tab and select the variable from the Variables list. The associated text is displayed in the box to the right.

To delete a text variable

1. Select File > Form Properties.
2. In the Variables tab, select the variable and click Delete (Delete) .

Using variables in calculations and scripts

After you have created form variables, you only need to reference the variable name in your calculations and scripts in order to obtain the value of the variable.

Caution: When naming variables, you should avoid using names that are identical to the names of any XML Form Object Model properties, methods, or object names. For information about XML Form Object Model properties, methods, and objects, see the *Adobe XML Form Object Model Reference* at www.adobe.com/devnet/livecycle/designing_forms.html.

For example, create the following form variable definitions.

Variable name	Value
firstName	Tony
lastName	Blue
age	32

In FormCalc, you can access the variable values in the same manner that you access field and object values. In this example, the values are assigned to three separate fields:

```
TextField1 = firstName  
TextField2 = lastName  
NumericField1 = age
```

You can also use variables in FormCalc functions in the same way, as shown in this example:

```
Concat( "Dear ", firstName, lastName )
```

In JavaScript, you reference variable values by using the `.value` property instead of the `.rawValue` property that is used for field and object values, as shown in this example:

```
TextField1.rawValue = firstName.value;
```

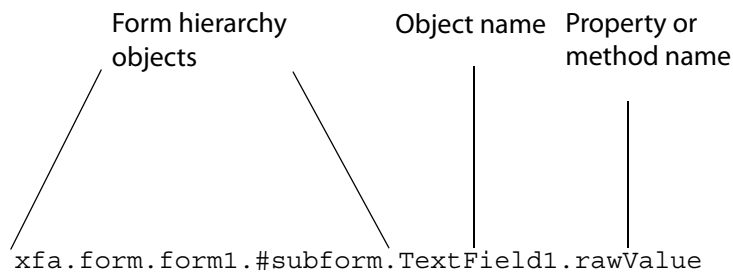
About referencing objects in calculations and scripts

Although both FormCalc calculations and JavaScript scripts have rules for structuring code, both rely on the same reference syntax when accessing form object properties and values. The XML Form Object Model provides a structured way to access object properties and values through a compound naming convention with each object, property, and method separated by dot (.) characters.

In general, each reference syntax has a similar structure divided into the following sections:

- The names of the parent objects in the form hierarchy that is used to navigate to a specific field or object. You can use the Hierarchy palette and Data View palette to determine the location of an object relative to other objects in the form and in any associated data.
- The name of the object you want to reference.
- The the name of the property or method you want to access. This section may also include XML Form Object Model objects that precede the property or method in the structure but that do not appear as objects in the Hierarchy palette.

For example, the following illustration shows the reference syntax for accessing the value of a text field on a form design that uses the default object-naming conventions:



Note: By default, the subform object that represents the first page of a new form is unnamed. In the reference syntax above, the unnamed subform is represented by `#subform`.

The reference syntax notation structure varies slightly, depending on the specific situation. For example, a fully qualified reference syntax works in any situation; however, in some cases, you can use a shortened reference syntax or a reference syntax shortcut to reduce the size of the syntax.

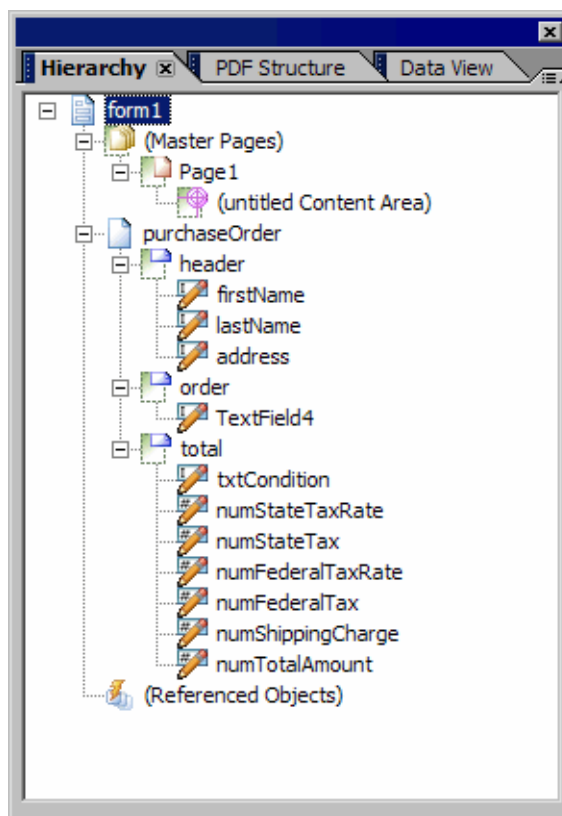
Referencing object properties and values

The reference syntax you use to access or modify object properties and values takes one of the following forms:

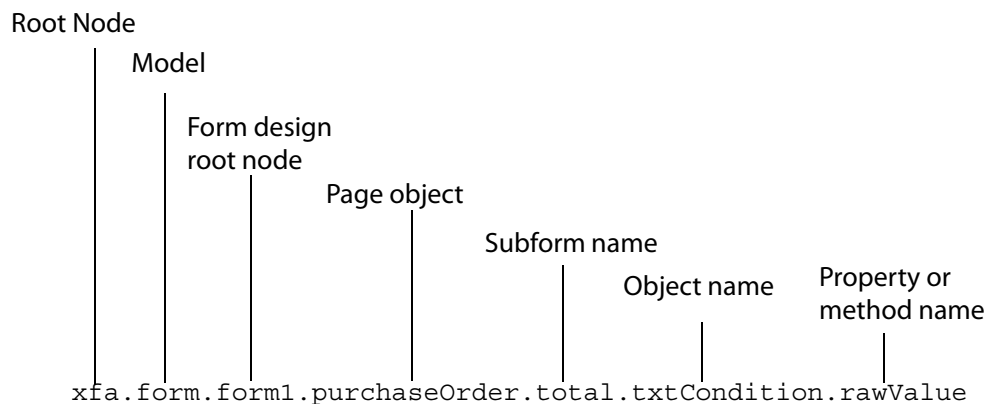
Fully qualified Reference syntax includes the full object hierarchy, beginning with the `xfa` root node. The fully qualified syntax accurately accesses the property or value of an object regardless of where the calculation or script that contains the reference syntax is located.

Abbreviated The reference syntax is shortened either because of the relative positioning of the calculation or script that contains the reference syntax and the object syntax references, or because shortcuts are used. In general, although an abbreviated reference syntax is faster to create, the disadvantage is that it works only as long as the objects remain in the same positions relative to each other.

For example, this illustration shows the hierarchy of a sample purchase order form.



This illustration shows a fully qualified reference syntax, for both FormCalc and JavaScript, to access the value of the `txtCondition` field. This reference syntax could be used as part of a calculation or script on any object on the form.



Note: Even though the reference syntax is common to both FormCalc and JavaScript, you must observe the conventions for each scripting language. For example, the reference syntax in the example above works as is for FormCalc; however, you would need to include a trailing semicolon (;) character for JavaScript.

If two objects exist in the same container, such as a subform, they are referred to as sharing the same context. Where objects exist in the same context, you can use an abbreviated reference syntax that includes only the name of the object followed by the property or method you want to access. For example, using the example from above, the following abbreviated reference syntax accesses the value of the `txtCondition` field from any of the fields in the `total` subform:

```
txtCondition.rawValue
```

If two objects exist in different containers, they do not share the same context. In this case, you can use an abbreviated reference syntax; however, the syntax must begin with the name of the highest level container object that the two objects do not have in common. For example, using the hierarchy above, the following abbreviated reference syntax accesses the value of the `address` field from the `txtCondition` field:

```
header.address.rawValue
```

Due to the way the XML Form Object Model is structured, some object properties and methods exist on child objects of the objects on the form. These child objects exist only as part of the XML Form Object Model and do not appear in the Hierarchy and Data View palettes. To access these properties and methods, you must include the child objects in the reference syntax. For example, the following reference syntax sets the tool tip text for the `txtCondition` field:

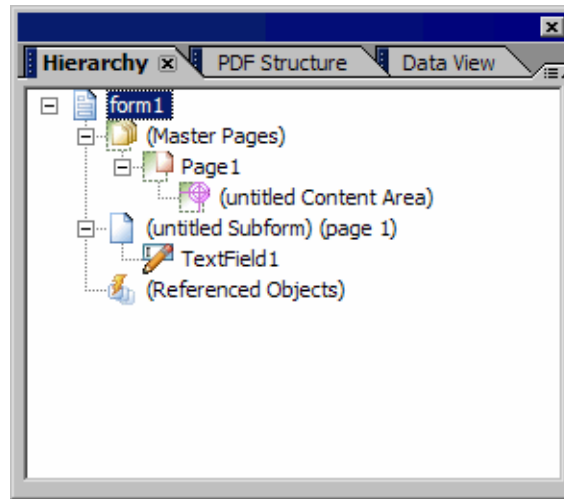
```
txtCondition.assist.toolTip.value = "Conditions of purchase." // FormCalc  
txtCondition.assist.toolTip.value = "Conditions of purchase."; // JavaScript
```

For more information about the XML Form Object model objects and their structure, see the *Adobe XML Form Object Model Reference* at www.adobe.com/devnet/livecycle/designing_forms.html.

Referencing unnamed and repeated objects

LiveCycle Designer supports the capability to create both unnamed objects and multiple objects with the same name. You can still create calculations and scripts to access and modify properties and values of unnamed objects by using the number sign (#) notation and object occurrence values using the square bracket ([]) notation. FormCalc correctly interprets the number sign (#) and square bracket ([]) characters; however, JavaScript does not. To access the value of a text field in a situation where the number sign (#) or square brackets ([]) occur, using JavaScript, you must use the `resolveNode` method in conjunction with either a fully qualified reference syntax or an abbreviated reference syntax.

For example, when you create a new blank form, by default, the name of the subform that represents the page of the form is an unnamed subform with an occurrence value of 0. The following illustration shows the form hierarchy on a new form with default object naming.



The untitled subform that represents the first page of the form has an occurrence number of 0. In this situation, both of the following reference syntaxes access the value of the text field in the form hierarchy above on a new form that uses default naming conditions:

```
xfa.form.form1.#subform.TextField1.rawValue  
xfa.form.form1.#subform[0].TextField1.rawValue
```

Note: By default, if you do not specify an occurrence value for an object, the reference syntax accesses the first occurrence of that object.

FormCalc recognizes the fully qualified reference syntax above and interprets it directly. To access the same value by using JavaScript, you must use one of these forms of the `resolveNode` scripting method:

```
xfa.resolveNode("xfa.form.form1.#subform.TextField1").rawValue;  
xfa.resolveNode("xfa.form.form1.#subform[0].TextField1").rawValue;
```

If you add a new page to your form, by default, the name of the subform that represents the new page is unnamed; however, the occurrence value for the new subform is set to 1. You can specify the new unnamed subform by using a similar reference syntax as above:

```
xfa.form.form1.#subform[1].TextField1.rawValue // FormCalc  
xfa.resolveNode("xfa.form.form1.#subform[1].TextField1").rawValue; // JavaScript
```

Note: The statement completion options available in the Script Editor include unnamed objects at the beginning of the list. Objects that have multiple occurrence values appear only once in the list, representing the first occurrence of the object. If you want to access an occurrence value other than the first occurrence, you must manually add the occurrence value to the reference syntax.

You can use the `resolveNode` method to reference objects within other reference syntax statements. This can help to reduce the amount of scripting you need to reference a particular object, property, or method. For example, you could simplify the reference syntax that points to a text field on the second page of your form to the following statement:

```
xfa.form.form1.resolveNode("#subform[1].TextField1").rawValue; // JavaScript
```

Referencing the current object

If you want to change properties or values of the current object using calculations or scripts attached to the object itself, both FormCalc and JavaScript use unique shortcuts to reduce the size of the reference syntax. FormCalc uses the number sign (\$) character to denote the current object, and JavaScript uses the keyword `this`.

For example, the following reference syntax returns the value of the current object:

```
$ // FormCalc
this.rawValue // JavaScript
```

Similarly, you can use the dollar sign (\$) shortcut and the keyword `this` to replace the name of the current object when accessing object properties in calculations and scripts. For example, the following reference syntax changes the tool tip text associated with the current object:

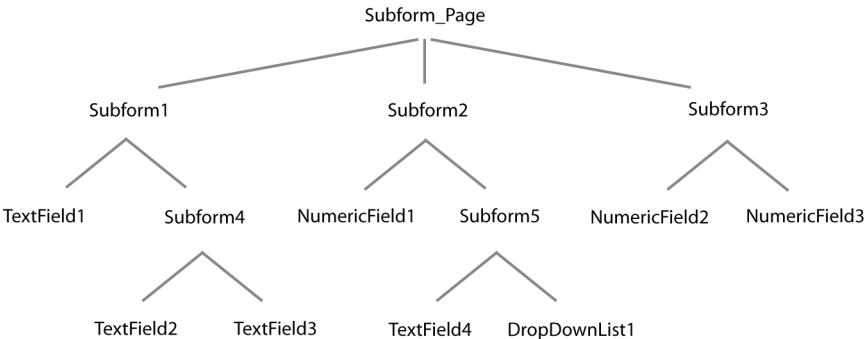
```
$.assist.toolTip.value = "This is some tool tip text." // FormCalc
this.assist.toolTip.value = "This is some tool tip text."; // JavaScript
```

FormCalc reference syntax shortcuts

To make accessing object properties and values easier, FormCalc includes shortcuts to reduce the effort required to create references. The following table outlines the reference syntax shortcuts for FormCalc.

Notation	Description
\$	Refers to the current field or object, as shown in this example: <code>\$ = "Tony Blue"</code> The above example sets the value of the current field or object to <code>Tony Blue</code> .
\$data	Represents the root of the data model <code>xfa.datasets.data</code> . For example, <code>\$data.purchaseOrder.total</code> is equivalent to <code>xfa.datasets.data.purchaseOrder.total</code>
\$event	Represents the current form object event. For example, <code>\$event.name</code> is equivalent to <code>xfa.event.name</code>
\$form	Represents the root of the form model <code>xfa.form</code> . For example, <code>\$form.purchaseOrder.tax</code> is equivalent to stating <code>xfa.form.purchaseOrder.tax</code>
\$host	Represents the host object. For example, <code>\$host.messageBox("Hello world")</code> is equivalent to <code>xfa.host.messageBox("Hello world")</code>

Notation	Description
\$layout	Represents the root of the layout model <code>xfa.layout</code> . For example, <code>\$layout.ready</code> is equivalent to stating <code>xfa.layout.ready</code>
\$record	Represents the current record of a collection of data, such as from an XML file. For example, <code>\$record.header.txtOrderedByCity</code> references the <code>txtOrderedByCity</code> node within the <code>header</code> node of the current XML data.
\$template	Represents the root of the template model <code>xfa.template</code> . For example, <code>\$template.purchaseOrder.item</code> is equivalent to <code>xfa.template.purchaseOrder.item</code>
!	Represents the root of the data model <code>xfa.datasets</code> . For example, <code>!data</code> is equivalent to <code>xfa.datasets.data</code>
*	Selects all form objects within a given container, such as a subform, regardless of name, or selects all objects that have a similar name. For example, the following expression selects all objects named <code>item</code> on a form: <code>xfa.form.form1.item[*]</code> Note: You can use the '*' (asterisk) syntax with JavaScript if it used with the <code>resolveNode</code> method. For more information about the <code>resolveNode</code> method, see the <i>LiveCycle Designer Help</i> or the <i>Adobe XML Form Object Model Reference</i> .

Notation	Description
..	<p>You can use two dots at any point in your reference syntax to search for objects that are a part of any subcontainer of the current container object, such as a subform. For example, the expression <code>Subform_Page..Subform2</code> means locate the node <code>Subform_Page</code> (as usual) and find a descendant of <code>Subform_Page</code> called <code>Subform2</code>.</p>  <pre> graph TD SP[Subform_Page] --> S1[Subform1] SP --> S2[Subform2] SP --> S3[Subform3] S1 --> T1[TextField1] S1 --> S4[Subform4] S2 --> NF1[NumericField1] S2 --> S5[Subform5] S3 --> NF2[NumericField2] S3 --> NF3[NumericField3] S4 --> T2[TextField2] S4 --> T3[TextField3] S5 --> T4[TextField4] S5 --> DDL1[DropDownList1] </pre> <p>Using the example tree above,</p> <p><code>Subform_Page..TextField2</code></p> <p>is equivalent to</p> <p><code>Subform_Page.Subform1[0].Subform3.TextField2[0]</code></p> <p>because <code>TextField2[0]</code> is in the first <code>Subform1</code> node that FormCalc encounters on its search. As a second example,</p> <p><code>Subform_Page..Subform3[*]</code></p> <p>returns all four instances of the <code>TextField2</code> object.</p> <p>Note: You can use the <code>..'</code> (double period) syntax with JavaScript if it used with the <code>resolveNode</code> method. For more information about the <code>resolveNode</code> method, see the <i>LiveCycle Designer Help</i> or the <i>Adobe XML Form Object Model Reference</i>.</p>
#	<p>The number sign (#) notation is used to denote one of the following items in a reference syntax:</p> <ul style="list-style-type: none"> • An unnamed object. For example, the following reference syntax accesses an unnamed subform: <pre>xfa.form.form1.#subform</pre> • Specify a property in a reference syntax if a property and an object have the same name. For example, the following reference syntax accesses the <code>name</code> property of a subform if the subform also contains a field named <code>name</code>: <pre>xfa.form.form1.#subform.#name</pre> <p>Note: You can use the <code>#</code> (number sign) syntax with JavaScript if it used with the <code>resolveNode</code> method. For more information about the <code>resolveNode</code> method, see the <i>LiveCycle Designer Help</i> or the <i>Adobe XML Form Object Model Reference</i>.</p>

Notation	Description
[]	<p>The square bracket ([]) notation denotes the occurrence value of an object. To construct an occurrence value reference, place square brackets ([]) after an object name, and enclose within the brackets one of the following values:</p> <ul style="list-style-type: none"> • [n], where <i>n</i> is an absolute occurrence index number beginning at 0. An occurrence number that is out of range does not return a value. For example, <pre>xfa.form.form1.#subform.Quantity[3]</pre> refers to the fourth occurrence of the <code>Quantity</code> object. • [+/- n], where <i>n</i> indicates an occurrence relative to the occurrence of the object making the reference. Positive values yield higher occurrence numbers, and negative values yield lower occurrence numbers. For example, <pre>xfa.form.form1.#subform.Quantity[+2]</pre> <p>This reference yields the occurrence of <code>Quantity</code> whose occurrence number is two more than the occurrence number of the container making the reference. For example, if this reference was attached to the <code>Quantity[2]</code> object, the reference would be the same as</p> <pre>xfa.template.Quantity[4]</pre> <p>If the computed index number is out of range, the reference returns an error.</p> <p>The most common use of this syntax is for locating the previous or next occurrence of a particular object. For example, every occurrence of the <code>Quantity</code> object (except the first) might use <code>Quantity[-1]</code> to get the value of the previous <code>Quantity</code> object.</p> • [*] indicates multiple occurrences of an object. The first named object is found, and objects of the same name that are siblings to the first are returned. Note that using this notation returns a collection of objects. For example, <pre>xfa.form.form1.#subform.Quantity[*]</pre> <p>This expression refers to all objects with a name of <code>Quantity</code> that are siblings to the first occurrence of <code>Quantity</code> found by the reference.</p> <p>Note: In language-specific forms for Arabic, Hebrew, Thai, and Vietnamese, the reference syntax is always on the right (even for right-to-left languages).</p> <p>LiveCycle Forms 7.1 does not support form designs that contain Arabic, Hebrew, Thai, and Vietnamese fonts.</p>

Notation	Description
[] (Continued)	<div data-bbox="550 273 1428 619"> <pre> graph TD Subform_Page --> Subform1 Subform_Page --> Subform2 Subform_Page --> Subform3 Subform1 --> TextField1 Subform1 --> Subform4 Subform2 --> NumericField1 Subform2 --> Subform5 Subform3 --> NumericField2 Subform3 --> NumericField3 Subform4 --> TextField2 Subform4 --> TextField3 Subform5 --> TextField4 Subform5 --> DropDownList1 </pre> </div> <p>Using the tree for reference, these expressions return the following objects:</p> <ul style="list-style-type: none"> • <code>Subform_Page.Subform1[*]</code> returns both Subform1 objects. • <code>Subform_Page.Subform1.Subform3.TextField2[*]</code> returns the two TextField2 objects. <code>Subform_Page.Subform1</code> resolves to the first Subform1 object on the left, and <code>TextField2[*]</code> evaluates relative to the Subform3 object. • <code>Subform_Page.Subform1[*].TextField1</code> returns both of the TextField1 instances. <code>Subform_Page.Subform1[*]</code> resolves to both Subform1 objects, and <code>TextField1</code> evaluates relative to the Subform1 objects. • <code>Subform_Page.Subform1[*].Subform3.TextField2[1]</code> returns the second and fourth TextField2 objects from the left. <code>Subform_Page.Subform1[*]</code> resolves to both Subform1 objects, and <code>TextField2[1]</code> evaluates relative to the Subform3 objects. • <code>Subform_Page.Subform1[*].Subform3[*]</code> returns both instances of the Subform3 object. • <code>Subform_Page.*</code> returns both Subform1 objects and the Subform2 object. • <code>Subform_Page.Subform2.*</code> returns the two instances of the NumericField2 object. <p>Note: You can use the '[' (square bracket) syntax with JavaScript if it used with the <code>resolveNode</code> method. For more information on the <code>resolveNode</code> method, see the <i>LiveCycle Designer Help</i> or the <i>Adobe XML Form Object Model Reference</i>.</p>

About the script object

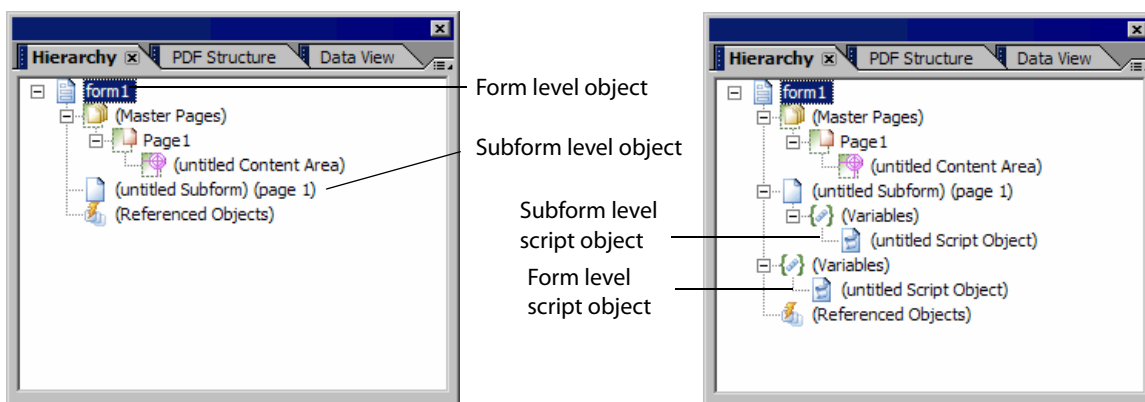
The *script object* is an object you can use to store JavaScript functions and values separately from any particular form object. Typically, you use the script object to create custom functions and methods that you want to use as part of JavaScript scripts in many locations on your form. This technique reduces the overall amount of scripting required to perform repetitive actions.

The script object only supports script written in JavaScript; however, there are no restrictions on the location where the scripts are executed, provided that the scripting language for the event that invokes the script object is set to JavaScript. Both Acrobat and LiveCycle Forms process scripting from a script object in the same manner, but both are also distinct. Only scripts set to run on the client can make use of script objects set to run on the client, and vice versa.

To create a script object

There are two parts to creating a script object. The first part involves adding the object to the form design, and the second part is writing the script you want to store in the script object.

1. Create a new form or open an existing form.
2. In the Hierarchy palette, right-click either a form-level object or a subform-level object and select Insert Script Object.



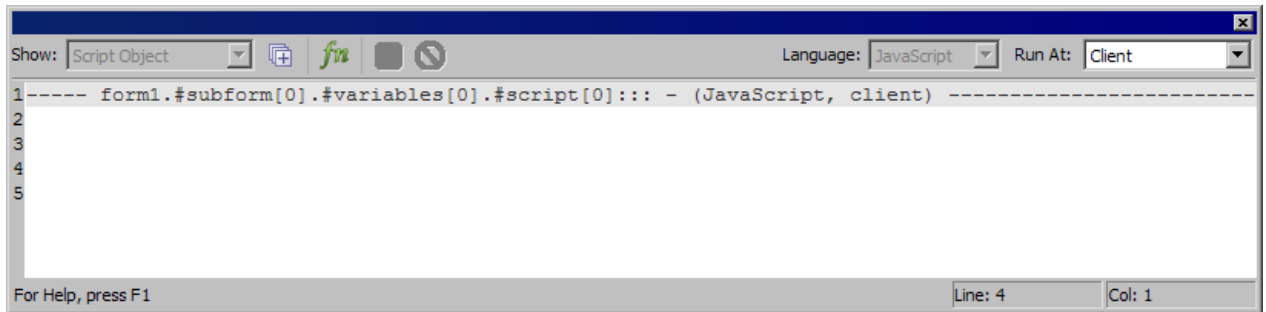
3. (Optional) Right-click the script object and select Rename Object.


To add script to a script object

After you have a script object on your form, you can add scripts using the Script Editor.

1. Select the script object in the Hierarchy palette.

The Script Editor is displayed with both a Script Object value in the Show list and a JavaScript value in the Language list. You cannot change either of these values.



2. Enter your script in the Script Source field.
3. When finished, click Enter Script Source Changes  to add the script to your form.

Example

For example, create a script object called `feedback` that contains the following function:

```
function emptyCheck(oField) {  
  
    if ((oField.rawValue == null) || (oField.rawValue == "")) {  
        xfa.host.messageBox("You must input a value for this field.", "Error  
Message", 3);  
    }  
}
```

To reference JavaScript functions stored in a script object

After you add scripts to a script object, you can reference the script object from any event that supports JavaScript scripts.

1. Select an object on your form and select an event from the Show list.
2. Create a reference to the script object and any functions within the script object. The following generic syntax assumes that the object where you are referencing the script object is at the same level as the script object in the form hierarchy or that the script object exists at the highest level of the form hierarchy.

```
script_object.function_name(parameter1, ...);
```

3. Apply the new script to the form object and test it by previewing the form using the Preview PDF tab.

Similar to referencing other objects on a form, you must provide a valid syntax when referencing the script object that includes where it exists within the form hierarchy. For more information about referencing objects in scripting, see ["Referencing object properties and values" on page 56](#).

Example

For example, using the script object example from ["To add script to a script object" on page 65](#), place the following JavaScript script on the `exit` event for a text field. Test the form using the Preview PDF tab.

```
feedback.emptyCheck(this);
```

About debugging calculations and scripts

LiveCycle Designer includes a number of features and strategies for debugging your calculations and scripts, depending on the scripting language you choose.

For both FormCalc and JavaScript, you can take advantage of certain LiveCycle Designer capabilities to debug and verify your scripts.

- Using the Warnings tab in the Report palette, you can view any errors or messages generated by LiveCycle Designer. This tab acts as a log of actions reported by LiveCycle Designer and updates when you preview a form design by using the Preview PDF tab. For more information about using the Warnings tab and the LiveCycle Designer workspace to debug your scripts, see [“Using the workspace to debug calculations and scripts” on page 14](#).
- Using the XML Form Object Model `messageBox` method to output information from an interactive form into a dialog box at run time.
- For certain types of debugging, you may want to output information, such as field values or messages, into a text field on your form design. For example, you can append new messages or values to the value of a text field to create a log that you can refer to when resolving issues.

If you use Acrobat Professional, you can use the JavaScript Debugger from Acrobat to test your form scripts. The JavaScript Debugger includes the JavaScript Console, which you can use to test portions of JavaScript code in the Preview PDF tab. The JavaScript Console provides an interactive and convenient interface for testing portions of JavaScript code and experimenting with object properties and methods. Because of its interactive nature, the JavaScript Console behaves as an editor that supports the execution of single lines or blocks of code.

For more information about the JavaScript Console and the JavaScript Debugger, see *Developing Acrobat Applications Using JavaScript* at partners.adobe.com/public/developer/acrobat/devcenter.html (English only).

Providing debugging feedback using the `messageBox` method

Regardless of the type of form you create, you can take advantage of the XML Form Object Model `messageBox` method to display messages or field values at run time. When initiated, the `messageBox` method displays a string value in a new client application dialog box. The string value can be a text message that you create for debugging purposes or the string value of fields or expressions.

For example, consider a scenario with a simple form design that contains a single numeric field (NumericField1) and a button (Button1). In this case, the following FormCalc calculation and JavaScript script each output a message displaying some text and the value currently displayed in the numeric field. By adding either the calculation or the script to the `click` event of the button object, you can interactively display the value of the numeric field in a new dialog box by clicking the button.

FormCalc

```
xfa.host.messageBox(Concat("The value of NumericField1 is: ",  
    NumericField1), "Debugging", 3)
```

JavaScript

```
xfa.host.messageBox("The value of NumericField1 is: " +  
    NumericField1.rawValue, "Debugging", 3);
```

Caution: The `messageBox` method returns an integer value representing the button that the form filler selects in the message box dialog. If you attach the `messageBox` method to the `calculate` event of a field object, and the `messagebox` method is the last line of the script, the field displays the return value of the `messageBox` method at run time.

In some situations, such as if you want to return a message box during a `calculate` event, you can take advantage of the `alert` method from the JavaScript Object Model from Acrobat. For example, the following script returns the value of a text field:

```
var oField = xfa.resolveNode("TextField1").rawValue;  
app.alert(oField);
```

For more information about the `alert` method and the JavaScript Object Model from Acrobat, see *Developing Acrobat Applications Using JavaScript* at partners.adobe.com/public/developer/acrobat/devcenter.html (English only).

To enable the JavaScript Debugger for LiveCycle Designer

To enable the JavaScript Debugger for LiveCycle Designer and execute code from the JavaScript Console, you must enable JavaScript and the JavaScript Debugger in Acrobat Professional. You need to enable the JavaScript Debugger to access the JavaScript Console because the console window is a component in the JavaScript Debugger interface.

Note: You can enable the JavaScript Debugger in Adobe Reader if you have LiveCycle Reader Extensions installed. To enable the JavaScript Debugger in Adobe Reader, you need to get the `debugger.js` file and then edit the Microsoft Windows Registry. For more information about enabling the JavaScript Debugger in Adobe Reader, see *Developing Acrobat Applications Using JavaScript* at partners.adobe.com/public/developer/acrobat/devcenter.html (English only).

1. Start LiveCycle Designer.
2. Start Acrobat Professional.
3. In Acrobat Professional, select **Edit > Preferences**.
4. Select **JavaScript** from the list on the left.
5. Select **Enable Acrobat JavaScript** if it is not already selected.
6. Under **JavaScript Debugger**, select **Enable JavaScript Debugger After Acrobat Is Restarted**.
7. Select **Enable Interactive Console**. This option lets you evaluate code that you write in the JavaScript Console.
8. Select **Show Console On Errors And Messages**. This option ensures that if you make mistakes, the JavaScript Console displays helpful information.
9. Click **OK** to close the Preferences dialog box.
10. Quit Acrobat Professional.

11. In LiveCycle Designer, click the Preview PDF tab.
12. Press Ctrl+J to open the JavaScript Debugger.

To prevent the JavaScript Debugger from disappearing in LiveCycle Designer

If you have the JavaScript Debugger from Acrobat active and it disappears when you click components in the LiveCycle Designer interface, you must stop the Acrobat.exe process in the Microsoft Windows Task Manager. The Acrobat.exe process continues to run after Acrobat is closed so that Acrobat is displayed faster if it is restarted. Stopping the Acrobat.exe process ends the association between the JavaScript Debugger and the Acrobat Professional session so that you can use the JavaScript Debugger in LiveCycle Designer.

1. In the Windows Task Manager, click the Processes tab.
2. In the Image Name column, right-click Acrobat.exe and select End Process.

Executing scripts in the JavaScript Console

Using the JavaScript Console from Acrobat, you can to evaluate single or multiple lines of code.

There are three ways to evaluate JavaScript code while using the interactive JavaScript Console:

- To evaluate a portion of a line of code, highlight the portion in the console window and press either Enter on the numeric keypad or Ctrl+Enter on the regular keyboard.
- To evaluate a single line of code, make sure the cursor is positioned in the appropriate line in the console window and press either Enter on the numeric keypad or Ctrl+Enter on the regular keyboard.
- To evaluate multiple lines of code, highlight those lines in the console window and press either Enter on the numeric keypad or Ctrl+Enter on the regular keyboard.

In all cases, the result of the most recently evaluated JavaScript script is displayed in the console window.

After evaluating each JavaScript script, the console window prints out `undefined`, which is the return value of the statement. Notice that the result of a statement is not the same as the value of an expression within the statement. The return value `undefined` does not mean that the value of script is undefined; it means that the return value of the JavaScript statement is undefined.

Click Clear (the trash can icon) at the lower right of the console to delete any contents that appear in the console window.

Providing debugging feedback using the JavaScript Console

If you are creating scripts using JavaScript, you can output messages to the JavaScript Console from Acrobat at run time by using the `console.println` method included with the JavaScript Object Model from Acrobat. When initiated, the `console.println` method displays a string value in the JavaScript Console. The string value can be a text message that you create for debugging purposes or the string value of fields or expressions.

For example, consider a scenario with a simple form design that contains a single numeric field (NumericField1) and a button (Button1). In this case, the following JavaScript script outputs a message displaying some text and the value currently displayed in the numeric field. By adding either the calculation or the script to the `click` event of the button object, you can interactively display the value of the numeric field in a new dialog box by clicking the button.

```
console.println("The value is: " + NumericField1.rawValue);
```

For more information about the `console.println` method and the JavaScript Object Model from Acrobat, see *Developing Acrobat Applications Using JavaScript* at partners.adobe.com/public/developer/acrobat/devcenter.html (English only).

About working with a host application

A host application is the application in which a form exists at any given time. For example, if you are using LiveCycle Forms to render a form in HTML format, then during the pre-rendering process the host application is LiveCycle Forms. Once you render a form and view it in a client application such as Acrobat, Adobe Reader, or an HTML browser, then the client application becomes the host application.

LiveCycle Designer includes a scripting model that provides scripting properties and methods for directly interfacing with a hosting application. For example, you can use the properties and methods in the host scripting model to provide PDF page navigation actions in Acrobat or Adobe Reader, or you can use the `importData` method to load data into your form.

You can reference the host script model syntax on any valid scripting event for form design objects using the following syntax for both FormCalc and JavaScript:

```
xfa.host.property_or_method
```

Host scripting model properties and methods

Using the host scripting model properties and methods, you can retrieve information and execute actions that are not otherwise accessible through calculations and scripts. For example, you can retrieve the name of the host application (such as Acrobat), or advance the current page on an interactive form. The following table lists the properties and methods that are available for the host scripting model.

Properties	Methods
appType	beep
calculationsEnabled	exportData
currentPage	gotoURL
language	importData
name	messageBox
numPages	pageDown
platform	pageUp
title	print
validationsEnabled	resetData
variation	response
version	setFocus

For more information about the host scripting model properties and methods, see the LiveCycle Developer Center located at www.adobe.com/devnet/livecycle/.

Comparing the host scripting model functionality

This table lists the LiveCycle Designer host scripting model properties and methods, and compares them to the equivalent expressions in the JavaScript Object Model in Acrobat.

For more information about the host scripting model properties and methods, see the *LiveCycle Designer Help* or the *Adobe XML Form Object Model Reference* available at the Adobe LiveCycle Designer Developer Center website: www.adobe.com/devnet/livecycle/designing_forms.html.

Host scripting model properties and methods	JavaScript Object Model from Acrobat equivalent
<code>xfa.host.appType</code>	<code>app.viewerType</code>
<code>xfa.host.beep([INTEGER param])</code>	<code>app.beep([nType])</code>
<code>xfa.host.currentPage</code>	<code>doc.pageNum</code>
<code>xfa.host.exportData([STRING param1 [, BOOLEAN param2]])</code>	<code>doc.exportXFADData(cPath [, bXDP])</code>
<code>xfa.host.gotoURL(STRING param1 [, BOOLEAN param2])</code>	<code>doc.getURL(cURL, [bAppend])</code> or <code>app.launchURL(URL);</code>
<code>xfa.host.importData([STRING param])</code>	<code>doc.importXFADData(cPath)</code>
<code>xfa.host.language</code>	<code>app.language</code>
<code>xfa.host.messageBox(STRING param1 [, STRING param2 [, INTEGER param3 [, INTEGER param4]]])</code>	<code>app.alert(cMsg [, nIcon [, nType [, cTitle]]])</code>
<code>xfa.host.name</code>	<code>none</code>
<code>xfa.host.numPages</code>	<code>doc.numPages</code>
<code>xfa.host.pageDown()</code>	<code>doc.pageNum++</code>
<code>xfa.host.pageUp()</code>	<code>doc.pageNum--</code>
<code>xfa.host.platform</code>	<code>app.platform</code>
<code>xfa.host.print(BOOLEAN param1, INTEGER param2, INTEGER param3, BOOLEAN param4, BOOLEAN param5, BOOLEAN param6, BOOLEAN param7, BOOLEAN param8)</code>	<code>doc.print([bUI [, nStart [, nEnd [, bSilent [, bShrinkToFit [, bPrintAsImage [, bReverse [, bAnnotations]]]]]]])</code>
<code>xfa.host.resetData([STRING param])</code>	<code>doc.resetForm([aFields])</code>
<code>xfa.host.response(STRING param1 [, STRING param2 [, STRING param3 [, BOOLEAN param4]]])</code>	<code>app.response(cQuestion [, cTitle [, cDefault [, bPassword]]])</code>
<code>xfa.host.setFocus(STRING param)</code>	<code>field.setFocus()</code> (Deprecated)
<code>xfa.host.title</code>	<code>doc.title</code>

Host scripting model properties and methods	JavaScript Object Model from Acrobat equivalent
<code>xfa.host.variation</code>	<code>app.viewerVariation</code>
<code>xfa.host.version</code>	<code>app.viewerVersion</code>

About the event model

The event model stores object event properties. These properties are useful if you want to access values that are otherwise out of the scope of the events listed in the Show list within the Script Editor.

The event model controls the changes in a form that occur before, during, and after actions take place. These actions include dynamic form events, such as the point when the data and form design are merged but before pagination is applied, and also interactive form events, such as when a user updates the value of a field.

Event model properties and methods

Using the event object properties and methods, you can retrieve information and execute actions that otherwise are not accessible through calculations and scripts. For example, you can retrieve the full value of a field that otherwise would have part of the data stripped out because it is too long or otherwise invalid. Retrieving the full value of a field is useful in situations where you have to do extensive error checking.

Properties	Methods
change	emit
className	reset
commitKey	
fullText	
keyDown	
modifier	
name	
newContentType	
newText	
prevContentType	
prevText	
reenter	
selEnd	
selStart	
shift	
soapFaultCode	
soapFaultString	
target	

For more information about the event scripting model properties and methods, see the Adobe LiveCycle Designer Developer Center located at www.adobe.com/devnet/lifecycle/designing_forms.html.

About moving from scripting in Acrobat to LiveCycle Designer

LiveCycle Designer includes extensive scripting capabilities, including support for the most common JavaScript objects from Acrobat. When you convert an Acrobat form to LiveCycle Designer, most JavaScript scripts continue to work without requiring changes. However, you will need to manually convert some JavaScript scripts from Acrobat to maintain the behavior of your Acrobat form.

When converting scripts on your Acrobat form, note that LiveCycle Designer scripting differs from scripting in Acrobat in several ways:

- **LiveCycle Designer workspace** In the LiveCycle Designer workspace, you can change object properties and behaviors on your form without requiring you to create scripts.
- **Scripting languages** LiveCycle Designer includes support for JavaScript as well as for FormCalc, which is a simple calculation language. FormCalc includes built-in functions that perform many useful operations that would otherwise require extensive scripting.
- **Referencing objects, properties, and methods** LiveCycle Designer forms are highly structured; therefore, to reference specific objects, properties, or methods, you must include the appropriate reference syntax in your script. You can use the statement completion options in the Script Editor to assist you in creating reference syntaxes.

It is possible to continue to use JavaScript objects, properties, and methods from Acrobat in LiveCycle Designer. However, you should consider JavaScript from Acrobat only for tasks that you cannot perform using the XML Form Object Model in LiveCycle Designer. For example, you can use JavaScript from Acrobat to add attachments, bookmarks, and annotations; search or spell check the form; create reports; or access and manipulate metadata. You cannot use JavaScript from Acrobat to perform actions such as setting field values, adding new fields to a form, or deleting pages from a form.

Note: You cannot use Acrobat to add JavaScript scripts to a LiveCycle Designer form, including Acrobat forms that you have converted using LiveCycle Designer. When you view a LiveCycle Designer form in Acrobat, all JavaScript tools are unavailable.

For more information about converting Acrobat scripting to LiveCycle Designer, see the article *Converting Acrobat JavaScript for Use in LiveCycle Designer Forms* in the LiveCycle Developer Center at www.adobe.com/devnet/lifecycle/.

Converting Acrobat forms that contain scripts

One of the first steps in converting a form from Acrobat to LiveCycle Designer is to determine how much of the Acrobat scripting is supported by LiveCycle Designer and how much you must convert.

In general, you should convert all Acrobat scripting to an equivalent in LiveCycle Designer. LiveCycle Designer scripting takes full advantage of the highly structured nature of LiveCycle Designer forms, as well as useful forms-specific functionality, to make designing and implementing your forms solution faster and easier.

The Acrobat scripting you should retain include those that deal with the form's environment and peripheral operations, such as adding attachments or multimedia, performing searches, or creating reports and handling document metadata.

For more information about converting Acrobat scripting to LiveCycle Designer, see the article *Converting Acrobat JavaScript for Use in LiveCycle Designer Forms* in the LiveCycle Designer Developer Center at www.adobe.com/devnet/livecycle/.

Using JavaScript objects from Acrobat in LiveCycle Designer

In LiveCycle Designer, you can script against certain JavaScript objects in Acrobat by using the Acrobat scripting syntax. As a result, you can use the properties and methods of those objects on your form. For example, to display a message in the JavaScript Console from Acrobat, you can add the following script to the event of a form design object in LiveCycle Designer:

```
console.println("This message appears in the JavaScript Console.");
```

You can also have the form send itself by email by adding the following script to the `click` event of a button:

```
var myDoc = event.target;  
myDoc.mailDoc(true);
```

Note: In LiveCycle Designer, you must ensure that the scripting language for the event is set to JavaScript so that the script will execute correctly at run time.

You can also use references to the JavaScript objects in Acrobat in your reference syntax. For example, the following script gets the signed state of a document signature field and takes an action based on the state:

```
// Proceed if the current field is not signed.  
var oState =  
    event.target.getField("form1[0].#subform[0].SignatureField1[0]")  
        .signatureValidate(); //Get the field's signed state.  
  
if (oState == 0) {  
    ...  
}
```

Note: This example uses a fully qualified reference syntax to reference the text. For more information about referencing form design objects, see ["Referencing object properties and values" on page 56](#).

When working with JavaScript from Acrobat in LiveCycle Designer, remember these points:

- In LiveCycle Designer, use `event.target` to access the `Doc` JavaScript object from Acrobat. In Acrobat, the `this` object is used to reference the `Doc` object; however, in LiveCycle Designer, the `this` object refers to the form design object to which the script is attached.

- The Script Editor has no statement completion for JavaScript objects from Acrobat. You should refer to the *JavaScript for Acrobat API Reference* in the Acrobat Family Developer Center at partners.adobe.com/public/developer/acrobat/devcenter.html.

For more information about converting Acrobat scripting to LiveCycle Designer, see the article *Converting Acrobat JavaScript for Use in LiveCycle Designer Forms* in the LiveCycle Developer Center at www.adobe.com/devnet/livecycle/.

JavaScript objects from Acrobat supported in LiveCycle Designer

The following table lists the availability of the most commonly used Acrobat objects, properties, and methods in LiveCycle Designer, and provides information on any equivalent functionality in LiveCycle Designer. Although the table contains the most commonly used Acrobat objects, properties and methods, some are not listed, such as multimedia objects, because they are rarely used for forms.

In cases where no equivalent LiveCycle Designer functionality is listed, no direct LiveCycle Designer property or method can reproduce the Acrobat behavior. However, you can still create custom functions or scripts to replicate the Acrobat capability.

JavaScript in Acrobat	LiveCycle Designer support	JavaScript-equivalent in LiveCycle Designer	Comments
Annot object properties and methods			
All properties and methods	Yes	None	Only forms with a fixed layout support the annotation layer.
app object properties			
calculate	No	None	LiveCycle Designer includes the <code>execCalculate</code> method which initiates the <code>calculate</code> event.
language	Yes	<code>xfa.host.language</code>	See the <code>language</code> property.
monitors	Yes	None	
platform	Yes	<code>xfa.host.platform</code>	See the <code>platform</code> property.
plugins	Yes	None	
toolbar	Yes	None	
viewerType	Yes	<code>xfa.host.appType</code>	See the <code>appType</code> property.
viewerVariation	Yes	<code>xfa.host.variation</code>	See the <code>variation</code> property.
viewerVersion	Yes	<code>xfa.host.version</code>	See the <code>version</code> property.
app object methods			

JavaScript in Acrobat	LiveCycle Designer support	JavaScript-equivalent in LiveCycle Designer	Comments
addItem	Yes	None	
addSubMenu	Yes	None	
addToolButton	Yes	None	
alert	Yes	<code>xfa.host.messageBox()</code>	See the <code>messageBox</code> method.
beep	Yes	<code>xfa.host.beep()</code>	See the <code>beep</code> method.
browseForDoc	Yes	None	
clearInterval	Yes	None	
clearTimeOut	Yes	None	
execDialog	Yes	None	
execMenuItem	Yes	None	Executes the specified menu command. Use this method in LiveCycle Designer for File menu commands.
getNthPluginName	Yes	None	
getPath	Yes	None	
goBack	Yes	None	
goForward	Yes	None	
hideMenuItem	Yes	None	
hideToolBarButton	Yes	None	
launchURL	Yes	None	LiveCycle Designer includes the <code>gotoURL</code> method that loads a specified URL into the client application, such as Acrobat or Adobe Reader.
listMenuItems	Yes	None	
listToolBarButtons	Yes	None	
mailGetAddrs	Yes	None	
mailMsg	Yes	None	
newDoc	Yes	None	This method can only be executed during batch, console, or menu events.

JavaScript in Acrobat	LiveCycle Designer support	JavaScript-equivalent in LiveCycle Designer	Comments
newFDF	No	None	
openDoc	Yes	None	
openFDF	No	None	
popUpMenuEx	Yes	None	
popUpMenu	Yes	None	
removeToolButton	Yes	None	
response	Yes	<code>xfa.host.response()</code>	See the response method.
setInterval	Yes	None	
setTimeout	Yes	None	
trustedFunction	Yes	None	
trustPropagatorFunction	Yes	None	This method is only available during batch, console, and application initialization.
Bookmark object properties and methods			
All properties and methods	Yes	None	
doc object properties			
author	Yes	None	
baseURL	Yes	None	
bookmarkRoot	Yes	None	
calculate	No	None	
dataObjects	Yes	None	
delay	No	None	
dirty	Yes	None	This LiveCycle Designer JavaScript script saves a copy of a form and tests whether the form has changed: <pre>var sOrigXML = xfa.data.saveXML; if (sOrigXML != xfa.data.saveXML) {...}</pre>
disclosed	Yes	None	

JavaScript in Acrobat	LiveCycle Designer support	JavaScript-equivalent in LiveCycle Designer	Comments
documentFileName	Yes	None	
dynamicXFAForm	Yes	None	
external	Yes	None	
filesize	Yes	None	
hidden	Yes	None	
icons	Yes	None	
keywords	Yes	None	
layout	Yes	None	
media	Yes	None	
metadata	Yes	<code>xfa.form.desc</code>	See the <code>desc</code> object.
modDate	Yes	None	
mouseX mouseY	Yes	None	
noautocomplete	Yes	None	
nocache	Yes	None	
numFields	Yes	<code>xfa.layout.pageContent()</code>	The <code>pageContent</code> method returns a list of all objects of a particular type. However, you must execute the method for body pages and master pages to scan the entire form.
numPages	Yes	<code>xfa.host.numPages</code> or <code>xfa.layout.absPageCount()</code> <code>xfa.layout.pageCount()</code>	The <code>numPages</code> property returns the page count for the rendered form in the client. See also the <code>absPageCount</code> and <code>pageCount</code> methods.
pageNum	Yes	<code>xfa.host.currentPage</code>	See the <code>currentPage</code> property.
pageNum--	Yes	<code>xfa.host.currentPage--</code> or <code>xfa.host.pageUp()</code>	See the <code>currentPage</code> property or the <code>pageUp</code> method.

JavaScript in Acrobat	LiveCycle Designer support	JavaScript-equivalent in LiveCycle Designer	Comments
pageNum++	Yes	xfa.host.currentPage++ or xfa.host.pageDown()	See the currentPage property or the pageDown method.
path	Yes	None	
securityHandler	Yes	None	
templates	No	None	Use subform objects in LiveCycle Designer, and use properties and methods to add, remove, move, and set subform instances.
title	Yes	xfa.host.title	See the title property.
doc object methods			
addAnnot	Yes	None	
addField	No	None	You must use forms that have a fixed layout in LiveCycle Designer, and then use the instanceManager object to add, remove, and set the number of instances of a particular object.
addIcon	Yes	None	
addLink	No	None	
addRecipientListCryptFilter	Yes	None	
addScript	Yes	None	
addThumbnails	No	None	
addWatermarkFromFile	Yes	None	
addWatermarkFromText	Yes	None	
addWeblinks	Yes	None	
appRightsSign	Yes	None	
appRightsValidate	Yes	None	
bringToFront	Yes	None	

JavaScript in Acrobat	LiveCycle Designer support	JavaScript-equivalent in LiveCycle Designer	Comments
calculateNow	No	<code>xfa.form.calculate(1);</code> or <code>execCalculate()</code>	<p>The <code>calculate</code> method forces a specific set of scripts on <code>calculate</code> events to initiate. Boolean value indicates whether <code>True</code> (default) - all calculation scripts initiate; or <code>False</code> - only pending calculation scripts initiate.</p> <p>Note: LiveCycle Designer <code>calculate</code> object controls whether a form filler can override a field's calculated value.</p> <p>Alternatively, you can use the <code>execCalculate</code> method for each object for which you want to force a recalculation.</p>
closeDoc	Yes	None	
createDataObject	Yes	None	
createTemplate	No	None	LiveCycle Designer forms do not have an equivalent to the concept of an Acrobat template. You must use subform objects in LiveCycle Designer.
deletePages	No	None	In LiveCycle Designer, you can use the <code>instanceManager</code> object to remove the subform object that represents a page of your form.
embedDocAsDataObject	Yes	None	
encryptForRecipients	Yes	None	
encryptUsingPolicy	Yes	None	
exportAsText	Yes	None	This method is only available in the JavaScript Console of the JavaScript Debugger in Acrobat or during batch processing.

JavaScript in Acrobat	LiveCycle Designer support	JavaScript-equivalent in LiveCycle Designer	Comments
exportAsFDF	No	<code>xfa.host.exportData()</code>	The <code>exportData</code> method exports an XML or XDP file instead of an FDF file.
exportAsXFDF	No	<code>xfa.host.exportData()</code>	The <code>exportData</code> method exports an XML or XDP file instead of an FDF file.
exportDataObject	Yes	None	
exportXFADData	No	<code>xfa.host.exportData()</code>	The <code>exportData</code> method exports an XML or XDP file instead of an FDF file.
extractPages	No	None	
flattenPages	Yes	None	
getAnnot	Yes	None	
getAnnots	Yes	None	
getDataObjectContents	Yes	None	
getField("FieldName")	Yes	<code>xfa.resolveNode("FieldName")</code>	The <code>resolveNode</code> method accesses the specified object in the source XML of the form.
getLegalWarnings	Yes	None	
getLinks	No	None	
getNthFieldName	Yes	You must loop through all objects with a similar class name until you reach the <code>nth</code> occurrence.	See the <code>className</code> property.
getNthTemplate	No	None	
getOCGs	Yes	None	
getOCGOrder	Yes	None	
getPageBox	Yes	None	
getPageLabel	Yes	None	
getPageNthWord	Yes	None	
getPageNthWordQuads	Yes	None	
getPageNumWords	Yes	None	
getPageRotation	Yes	None	
getPrintParams	Yes	None	

JavaScript in Acrobat	LiveCycle Designer support	JavaScript-equivalent in LiveCycle Designer	Comments
getTemplate	No	None	
getURL	Yes	<code>xfa.host.gotoURL("http://www.adobe.com");</code>	See the gotoURL method.
gotoNamedDest	No	None	
importAnFDF	No	None	
importAnXFDF	Yes	None	
importDataObject	Yes	None	
importIcon	Yes	None	
importTextData	Yes	None	
importXFADData	No	<code>xfa.host.importData("filename.xdp");</code>	See the importData method.
insertPages	No	None	
mailDoc	Yes	None	
mailForm	No	None	
movePage	No	None	
newPage	No	None	
openDataObject	Yes	None	
print	Yes	<code>xfa.host.print();</code>	See the print method.
removeDataObject	Yes	None	
removeField	No	None	
removeIcon	Yes	None	
removeLinks	No	None	
removeScript	Yes	None	
removeTemplate	No	None	
removeThumbnails	No	None	
removeWeblinks	Yes	None	
replacePages	No	None	

JavaScript in Acrobat	LiveCycle Designer support	JavaScript-equivalent in LiveCycle Designer	Comments
resetForm	No	<code>xfa.host.resetData()</code> or <code>xfa.event.reset()</code>	The <code>resetData</code> method resets all field values on a form to the default values. The <code>reset</code> method resets all properties within the event model.
saveAs	Yes	None	In LiveCycle Designer, the file must be saved at the application level. These scripts are examples of saving at the application level: <code>app.executeMenuItem("SaveAs");</code> or <code>var myDoc = event.target;</code> <code>myDoc.saveAs();</code>
spawnPageFromTemplate	No	None	
setAction	No	None	
setPageLabel	Yes	None	
setPageRotation	No	None	
setPageTabOrder	No	None	In LiveCycle Designer, select Edit > Tab Order to set the tab order.
setScript	No	None	
submitForm	Yes	Use one of the submit button objects in LiveCycle Designer.	
event object properties			
change	Yes	<code>xfa.event.change</code>	See the <code>change</code> property.
targetName	Yes	<code>xfa.event.target</code>	See the <code>target</code> property.
field object properties			
comb	No	None	

JavaScript in Acrobat	LiveCycle Designer support	JavaScript-equivalent in LiveCycle Designer	Comments
<code>charLimit</code>	No	<code>this.value.#text.maxChars</code>	In forms that have a fixed layout, character limit can be set in the LiveCycle Designer workspace. You can set fields on forms whose layout expands to accommodate all data.
<code>display = display.noView</code>	No	See “Changing the presence of a form design object” on page 108.	You can also set the presence property in the LiveCycle Designer workspace. Note: You cannot use the <code>prePrint</code> event to change the presence of an object prior to printing.
<code>display = display.noPrint</code>	No	See “Changing the presence of a form design object” on page 108.	You can also set the presence property in the LiveCycle Designer workspace. Note: You cannot use the <code>prePrint</code> event to change the presence of an object prior to printing.
<code>defaultValue</code>	No	None	Set the default field value in the LiveCycle Designer workspace.
<code>exportValues</code>	No	None	Set the export value in the LiveCycle Designer workspace.
<code>fillColor</code>	No	<code>xfa.form.Form1.NumericField1.fillColor</code>	See the <code>fillColor</code> property.
<code>hidden</code>	No	<code>this.presence = "invisible" this.presence = "visible"</code>	You can also set the presence property in the LiveCycle Designer workspace.
<code>multiline</code>	No	<code>this.ui.textEdit.multiLine = "1";</code>	See the <code>multiLine</code> property.

JavaScript in Acrobat	LiveCycle Designer support	JavaScript-equivalent in LiveCycle Designer	Comments
password	No	None	LiveCycle Designer contains a Password Field that you can use for passwords on a form.
page	No	None	Not applicable for LiveCycle Designer forms.
print	No	<code>this.relevant = "-print";</code>	See the <code>relevant</code> property.
radiosInUnison	No	None	Grouped radio buttons in LiveCycle Designer are mutually exclusive by default.
rect	Yes	<p>You can get the height and width of a LiveCycle Designer form field by using the following reference syntax:</p> <pre>this.h; this.w;</pre> <p>Alternatively, you can retrieve the x and y coordinates of an object using the following reference syntax:</p> <pre>this.x; this.y;</pre>	See the <code>h</code> , <code>w</code> , <code>x</code> , and <code>y</code> properties.
required	No	<pre>this.mandatory = "error";</pre> <p>or</p> <pre>this.validate.nullTest = "error";</pre>	See the <code>mandatory</code> and <code>nullTest</code> properties.
textColor	No	<code>this.fontColor</code>	See the <code>fontColor</code> property.
textSize	No	<code>this.font.size</code>	See the <code>size</code> property.
textFont	No	<code>this.font.typeface</code>	See the <code>typeface</code> property.

JavaScript in Acrobat	LiveCycle Designer support	JavaScript-equivalent in LiveCycle Designer	Comments
value	No	this.rawValue	See the rawValue property. Note: LiveCycle Designer fields have a value property; it is not the equivalent of the Acrobat value property.
field object methods			
clearItems	No	DropDownList1.clearItems ();	The clearItems method only applies to Drop-down List and List Box objects in LiveCycle Designer.
deleteItemAt	No	None	
getItemAt	No	None	
insertItemAt	No	DropDownList1.addItem (.....)	See the addItem method.
isBoxChecked	No	if (CheckBox1.rawValue == 1)	See the rawValue property.
isDefaultChecked	No	None	
setAction	No	None	Not applicable for LiveCycle Designer forms.
setFocus	Yes	xfa.host.setFocus ("TextField1.somExpression")	The setFocus method requires that the specified object have a unique name with respect to other objects on your form.
setItems	No	None	
setLock	Yes	None	
signatureGetModifications	Yes	None	
signatureGetSeedValue	Yes	None	
signatureInfo	Yes	None	
signatureSetSeedValue	Yes	None	
signatureSign	Yes	None	
signatureValidate	Yes	None	
search object method			

JavaScript in Acrobat	LiveCycle Designer support	JavaScript-equivalent in LiveCycle Designer	Comments
<code>search.query("<your text>");</code>	Yes	None	The “.” (double period) FormCalc shortcut syntax allows you to search for objects within the XML Form Object Model.
SOAP object method			
All properties and methods	Yes	None	

This chapter will contain examples of using JavaScript scripts to perform specific tasks or outcomes. Initially, several samples will be included based on content that is posted to the LiveCycle Designer Developer Center, with the potential for more samples to be added in subsequent releases of this guide.

The samples that we plan to include for the first version of this guide are:

- [Changing the background colors of fields, fillable areas, and subforms](#)
- [Hiding and showing objects](#)
- [Changing the visual properties of an object on the client](#)
- [Getting the current or previous value of a drop-down list](#)
- [Adjusting the height of a field at run time](#)
- [Setting a field as required at run time](#)
- [Calculating the field sums](#)
- [Highlighting fields in response to form filler interaction](#)
- [Resetting the values of the current subform](#)
- [Changing the presence of a form design object](#)
- [Using the properties of the instance manager to control subforms](#)
- [Using the methods of the instance manager to control subforms](#)
- [Using the instance manager to control subforms at run time](#)

About the scripting examples

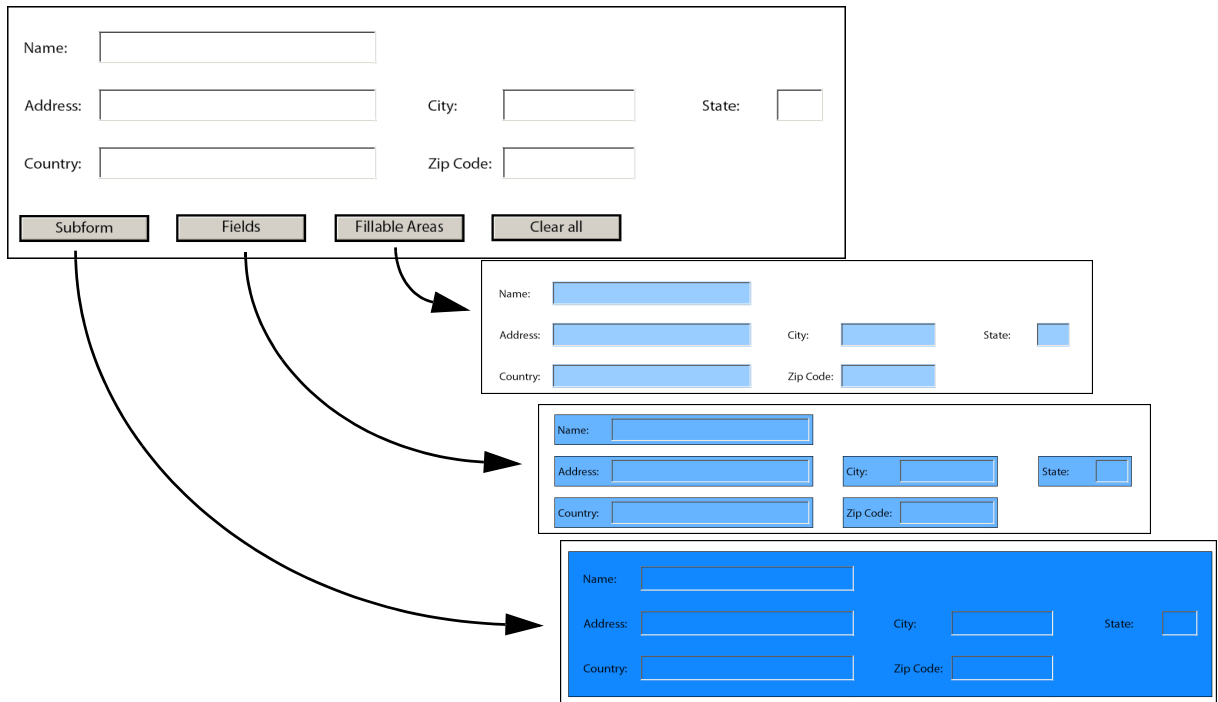
The scripting examples demonstrate quick and simple techniques that you can apply to your own work.

For more examples and ideas, visit the LiveCycle Developer Center at www.adobe.com/devnet/lifecycle.

Changing the background colors of fields, fillable areas, and subforms

This example demonstrates how to change the background color of subforms, fields, and fillable areas on a form in response to form filler interaction at run time.

In this example, clicking a button changes the background color of an associated object.



Note: To manipulate the background color of objects at run time, you must save your form as an Acrobat Dynamic XML Form file.

To see this scripting example and others, visit the LiveCycle Developer Center at www.adobe.com/devnet/lifecycle.

► Scripting the subform and text field background colors

You set the subform and the text field background colors by using the `fillColor` method. For example, the following line is the script for the subform:

```
Subform1.fillColor = "17,136,255";
```

The following lines make up the script for the background color of the text fields:

```
Subform1.Name.fillColor = "102,179,255";  
Subform1.Address.fillColor = "102,179,255";  
Subform1.City.fillColor = "102,179,255";  
Subform1.State.fillColor = "102,179,255";  
Subform1.ZipCode.fillColor = "102,179,255";  
Subform1.Country.fillColor = "102,179,255";
```

► Scripting the fillable area background color

When setting the background color or the fillable area for each text field, your scripts must access properties that require a reference syntax expression that includes the number sign (#). Because JavaScript does not interpret the number sign (#) properly in reference syntax expressions, the script uses the `resolveNode` method to resolve the expression.

```
xfa.resolveNode("Subform1.Name.ui.#textEdit.border.fill.color").value =  
"153,204,255";
```

```
xfa.resolveNode("Subform1.Address.ui.#textEdit.border.fill.color").value =  
"153,204,255";  
xfa.resolveNode("Subform1.City.ui.#textEdit.border.fill.color").value =  
"153,204,255";  
xfa.resolveNode("Subform1.State.ui.#textEdit.border.fill.color").value =  
"153,204,255";  
xfa.resolveNode("Subform1.ZipCode.ui.#textEdit.border.fill.color").value =  
"153,204,255";  
xfa.resolveNode("Subform1.Country.ui.#textEdit.border.fill.color").value =  
"153,204,255";
```

► Scripting the Clear All button

The script for the Clear All button uses the `remerge` method to remerge the form design and form data. In this case, the method effectively restores the fields, fillable areas, and subforms to their original state.

```
xfa.form.remerge();
```

Hiding and showing objects

This example demonstrates how to hide buttons when printing a form, as well as how to hide and show objects by changing the presence values at run time.

In this example, all form objects are showing in the form.

Form Objects

A screenshot of a form titled "Form Objects". It contains several input fields and three buttons. The fields are labeled "Name:", "Address:", "City:", "State:", "Zip Code:", and "Country:". The "Name" field contains "Jim Stall", "Address" contains "84 Brownstone St.", and "City" contains "Sunnydale". The "State", "Zip Code", and "Country" fields are empty. Below the fields are three buttons: "Submit", "Submit by Email", and "Print Form".

Presence Values

A screenshot of a "Presence Values" dialog box. It has two columns of dropdown menus. The first column has labels "Subform:", "Text Fields:", and "Buttons:". The second column has labels "Values:". Each label is followed by a dropdown menu. The "Subform" dropdown is set to "Subform1". The "Text Fields" dropdown is set to "Visible". The "Buttons" dropdown is set to "Visible". At the bottom of the dialog is a "Reset" button.

The form filler can use the drop-down lists in the Presence Values area to show or hide objects. In the following diagram, the Address field is hidden and the form layout has adjusted accordingly. The Print Form button is also invisible.

Form Objects

Name:

City:

State:

Zip Code:

Country:

Presence Values

Subform:	Values:
<input type="text" value="Subform1"/>	<input type="text" value="Visible"/>
Text Fields:	Values:
<input type="text" value="Address"/>	<input type="text" value="Hidden (Exclude from Layout)"/>
Buttons:	Values:
<input type="text" value="Print Form"/>	<input type="text" value="Invisible"/>

Note: To hide and show objects at run time, you must save your form as an Acrobat Dynamic PDF Form file.

To see this scripting example and others, visit the LiveCycle Developer Center
www.adobe.com/devnet/lifecycle.

► Scripting the presence values for subforms

The script for the subform presence values uses a switch statement to handle the three presence options that a form filler can apply to the subform object:

```
switch(xfa.event.newText) {  
    case 'Invisible':  
        Subform1.presence = "invisible";  
        break;  
    case 'Hidden (Exclude from Layout)':  
        Subform1.presence = "hidden";  
        break;  
    default:  
        Subform1.presence = "visible";  
        break;  
}
```

► Scripting the presence values for text fields

The script for the text fields presence values requires two variables. The first variable stores the number of objects contained in Subform1:

```
var nSubLength = Subform1.nodes.length;
```

The second variable stores the name of the text field that the form filler selects in the Text Fields drop-down list:

```
var sSelectField = fieldList.rawValue;
```

The following script uses the `replace` method to remove all of the spaces from the name of the field stored in the `sSelectField` variable, which allows the value of the drop-down list to match the name of the object in the Hierarchy palette:

```
sSelectField = sSelectField.replace(' ', '');
```

This script uses a `For` loop to cycle through all of the objects contained in Subform1:

```
for (var nCount = 0; nCount < nSubLength; nCount++) {
```

If the current object in Subform1 is of type `field` and the current object has the same name as the object that the form filler selected, the following switch cases are performed:

```
if ((Subform1.nodes.item(nCount).className == "field") &  
(Subform1.nodes.item(nCount).name == sSelectField)) {
```

The following script uses a switch statement to handle the three presence values that a form filler can apply to text field objects:

```
switch(xfa.event.newText) {  
    case 'Invisible':  
        Subform1.nodes.item(nCount).presence = "invisible";  
        break;  
    case 'Hidden (Exclude from Layout)':  
        Subform1.nodes.item(nCount).presence = "hidden";  
        break;  
    default:  
        Subform1.nodes.item(nCount).presence = "visible";  
        break;  
}  
}  
}
```

► Scripting the presence values for buttons

The script for the buttons presence values requires two variables. This variable stores the number of objects contained in Subform1:

```
var nSubLength = Subform1.nodes.length;
```

This variable stores the name of the button that the form filler selects in the Buttons drop-down list:

```
var sSelectButton = buttonList.rawValue;
```

The following script uses the `replace` method to remove all of the spaces from the name of the button stored in the `sSelectField` variable, which allows the value of the drop-down list to match the name of the object in the Hierarchy palette:

```
sSelectButton = sSelectButton.replace(/\s/g, '');
```

This script uses a `For` loop to cycle through all of the objects contained in `Subform1`:

```
for (var nCount = 0; nCount < nSubLength; nCount++) {
```

If the current object in `Subform1` is of type `field` and the current object has the same name as the object that the form filler selected, perform the following switch cases:

```
if ((Subform1.nodes.item(nCount).className == "field") &  
Subform1.nodes.item(nCount).name == sSelectButton)) {
```

This script uses a `switch` statement to handle the five presence values that a form filler can apply to button objects.

Note: The relevant property indicates whether an object should appear when the form is printed.

```
switch(xfa.event.newText) {  
  case 'Invisible':  
    Subform1.nodes.item(nCount).presence = "invisible";  
    break;  
  case 'Hidden (Exclude from Layout)':  
    Subform1.nodes.item(nCount).presence = "hidden";  
    break;  
  case 'Visible (but Don\'t Print)':  
    Subform1.nodes.item(nCount).presence = "visible";  
    Subform1.nodes.item(nCount).relevant = "-print";  
    break;  
  case 'Invisible (but Print Anyway)':  
    Subform1.nodes.item(nCount).presence = "invisible";  
    Subform1.nodes.item(nCount).relevant = "+print";  
    break;  
  default:  
    Subform1.nodes.item(nCount).presence = "visible";  
    break;  
}  
}  
}
```

► Scripting for resetting the drop-down lists

Use the `resetData` method to reset all of the drop-down lists to their default values:

```
xfa.host.resetData();
```

Use the `remerge` method to remerge the form design and form data. In this case, the method effectively returns the objects in the Form Objects area to their original states:

```
xfa.form.remerge();
```

Changing the visual properties of an object on the client

The example demonstrates how to manipulate the visual properties of an object; in this case, a text field. For example, selecting the Make the Field Wider check box expands the fillable area of the text field to four inches.

The screenshot shows the LiveCycle Designer interface. At the top, there's a text field with the caption 'Caption:' and the value 'This is the default value.' Below it, the 'Manipulating the object' panel is visible. This panel contains a list of checkboxes for manipulating the object's appearance. The 'Make the field wider' checkbox is checked, and an arrow points from the text field to this checkbox. Other checkboxes include 'Move the field', 'Make the field taller', 'Change the border color of the object', 'Change the fill color of the fillable area', 'Expand to fit the width of the value', and 'Make the field disappear'. To the right of this panel, there's another section for 'Modifying the text field value' and 'Other' properties, each with their own set of checkboxes. A 'Clear all' button is located at the bottom right of the properties panel.

Note: To alter the visual properties of objects on the client, you must save your form as an Acrobat Dynamic PDF Form file.

In this example, the check boxes do not have unique object names; therefore, LiveCycle Designer assigns an instance value to reference the object. The check box script uses an `if-else` statement to give the effect of selecting and deselecting.

To see this scripting example and others, visit the LiveCycle Developer Center www.adobe.com/devnet/livecycle.

► Scripting for the Move the Field check box

When the check box is selected, the field is moved according to the x and y settings. When the check box is deselected, the field is returned to its original location.

```
if (CheckBox1.rawValue == true) {  
    TextField.x = "3.0in";  
    TextField.y = "3.5in";  
}
```

```
else {  
    TextField.x = "1in";  
    TextField.y = "3in";  
}
```

► **Scripting for the Make the Field Wider check box**

When the check box is selected, the field changes to 4 inches. When the check box is deselected, the field width changes to 2.5 inches.

```
if (CheckBox2.rawValue == true)  
    TextField.w = "4in";  
else  
    TextField.w = "2.5in";
```

► **Scripting for the Make the Field Taller check box**

When the check box is selected, the field height changes to 1.5 inches. When the check box is deselected, the field height changes to .5 inches.

```
if (CheckBox3.rawValue == true)  
    TextField.h = "1.5in";  
else  
    TextField.h = "0.5in";
```

► **Scripting for the Change the Border Color of the Object check box**

When the check box is selected, the field border changes to red. When the check box is deselected, the field border changes to white.

```
if (CheckBox4.rawValue == true)  
    TextField.border.edge.color.value = "255,0,0";  
else  
    TextField.border.edge.color.value = "255,255,255";
```

► **Scripting for the Change the Fill Color of the Fillable Area check box**

When the check box is selected, the fillable area of the text field changes to green. When the check box is deselected, the fillable area of the text field changes to white.

```
if (CheckBox5.rawValue == true) {  
    xfa.resolveNode("TextField.ui.#textEdit.border.fill.color").value =  
    "0,255,0";  
}  
else {  
    xfa.resolveNode("TextField.ui.#textEdit.border.fill.color").value =  
    "255,255,255";  
}
```

► **Scripting for the Expand to Fit the Width of the Value check box**

When the check box is selected, the fillable area of the text field adjusts to accommodate the value. When the check box is deselected, the fillable area of the text field does not adjust.

```
if (CheckBox6.rawValue == true)  
    TextField.minW = "0.25in";  
else  
    TextField.maxW = "2.5in";
```

► **Scripting for the Make the Field Disappear check box**

When the check box is selected, the field is hidden. When the check box is deselected, the field is visible.

```
if (CheckBox7.rawValue == true)
    TextField.presence = "hidden";
else
    TextField.presence = "visible";
```

► **Scripting for the Change the Font of the Value check box**

When the check box is selected, the font of the value changes to Courier New. When the check box is deselected, the font of the value changes to Myriad Pro.

```
if (CheckBox8.rawValue == true)
    TextField.font.typeface = "Courier New";
else
    TextField.font.typeface = "Myriad Pro";
```

► **Scripting for the Change the Size of the Font check box**

When the check box is selected, the font size changes to 14 pt. When the check box is deselected, the font size changes to 10 pt.

```
if (CheckBox9.rawValue == true)
    TextField.font.size = "14pt";
else
    TextField.font.size = "10pt";
```

► **Scripting for the Align Text Field Value Vertically check box**

When the check box is selected, the text field value is aligned to the top. When the check box is deselected, the text field value is aligned to the middle.

```
if (CheckBox10.rawValue == true)
    TextField.para.vAlign = "top";
else
    TextField.para.vAlign = "middle";
```

► **Scripting for the Align Text Field Value Horizontally check box**

When the check box is selected, the text field value is aligned to the center. When the check box is deselected, the text field value is aligned to the left.

```
if (CheckBox11.rawValue == true)
    TextField.para.hAlign = "center";
else
    TextField.para.hAlign = "left";
```

► **Scripting for the Display a Set Value check box**

When the check box is selected, the value that is defined by using a script appears in the text field. When the check box is deselected, the default value (which is also defined by using a script) appears in the text field.

```
if (CheckBox12.rawValue == true)
    TextField.rawValue = "This is a value set using a script.";
else
    TextField.rawValue = "This is a default value.";
```

► Scripting for the Change the Caption Text check box

When the check box is selected, the alternate caption text that is defined by using a script appears as the caption. When the check box is deselected, the default caption (which is also defined by using a script) appears in the text field.

```
if (CheckBox13.rawValue == true)
    xfa.resolveNode("TextField.caption.value.#text").value = "Alternate
Caption: ";
else
    xfa.resolveNode("TextField.caption.value.#text").value = "Caption:";
```

► Scripting for the Change Field Border from 3D to Solid check box

When the check box is selected, the field border changes to a solid box. When the check box is deselected, the field border changes to 3D.

```
if (CheckBox14.rawValue == true)
    xfa.resolveNode("TextField.ui.#textEdit.border.edge").stroke = "solid";
else
    xfa.resolveNode("TextField.ui.#textEdit.border.edge").stroke = "lowered";
```

► Scripting for the Clear All Check Boxes button

Use the `resetData` method to reset all of the check boxes to their default value (Off).

```
xfa.host.resetData();
```

Use the `remerge` method to remerge the form design and form data. In this case, the method effectively returns the text field to its original state.

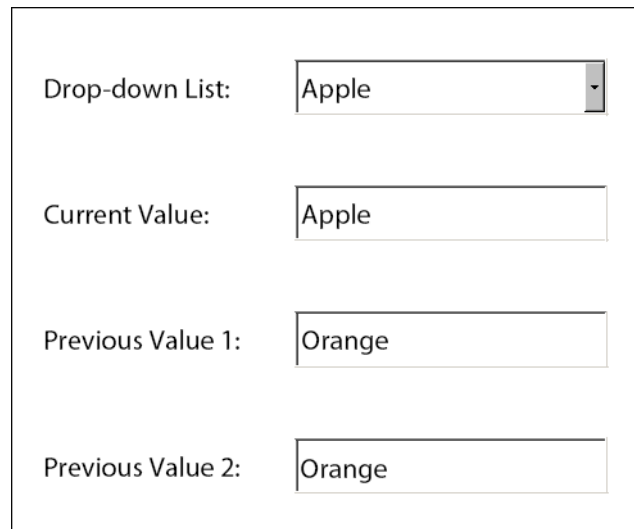
```
xfa.form.remerge();
```

Getting the current or previous value of a drop-down list

This example demonstrates how to obtain the current value of a drop-down list as well as the different ways to access the previous value of a drop-down list on a form. In addition to the actual scripts that set the current and previous values, it is important to note that the scripts are located on the `change` event for the drop-down list.

In the following example, when a form filler selects a value from the drop-down list, the selected value appears in the Current Value field. Then, when the form filler selects another value from the drop-down

list, the new value appears in the Current Value List and the previous value appears in the Previous Value 1 field.



Drop-down List:	Apple
Current Value:	Apple
Previous Value 1:	Orange
Previous Value 2:	Orange

Note: Each of the methods for obtaining the previous value of a drop-down list uses a different script. The Previous Value 1 text field is populated by a direct reference to the `rawValue` property of the drop-down list, whereas the Previous Value 2 text field is populated using the `prevText` property. For consistent results, it is recommended that you access the previous value by using the `prevText` property.

To see this scripting example and others, visit the LiveCycle Developer Center at www.adobe.com/devnet/livecycle.

► Scripting for populating the Current Value text field

Populate the value of the Current Value text field by using the `newText` property:

```
CurrentValue.rawValue = xfa.event.newText;
```

► Scripting for populating the Previous Value 1 text field

Populate the value of the Previous Value 1 text field by referencing the `rawValue` of the drop-down list:

```
PreviousValue1.rawValue = DropDownList.rawValue;
```

► Scripting for populating the Previous Value 2 text field

Populate the value of the Previous Value 2 text field by using the `prevText` property:

```
PreviousValue2.rawValue = xfa.event.prevText;
```

Adjusting the height of a field at run time

The example demonstrates how to expand a field to match the height of the content in another field.

In this example, when the form filler types multiple lines in TextField1 and then clicks the Expand button, the height of TextField2 increases to match the height of TextField1.

A diagram showing a form layout. On the left is a text field labeled "TextField1". To its right is another text field labeled "TextField2". Above "TextField2" is a button labeled "Expand". The "Expand" button is positioned such that its height is determined by the content height of "TextField1".

To see this scripting example and others, visit the LiveCycle Developer Center at www.adobe.com/devnet/lifecycle.

► Scripting for the Expand button

The following script is for the Expand button:

```
var newHeight = xfa.layout.h(TextField1, "in");  
TextField2.h = newHeight + "in";
```

Setting a field as required at run time

This example demonstrates how to make a field required at run time.

In this example, when the Set as Required button is clicked, if the form filler attempts to submit a form without typing some text in TextField1, an error message appears.

A diagram showing a form layout. At the top is a text field labeled "TextField1". Below it is a button labeled "Set as Required". Below that button is another button labeled "email".

To see this scripting example and others, visit the LiveCycle Developer Center www.adobe.com/devnet/lifecycle.

► Scripting for the Set as Required button

The following script is for the Set as Required button:

```
TextField1.validate.nullTest = "error";
```

You can also use one of these two scripts:

```
TextField1.mandatory = "error"  
TextField1.mandatoryMessage = "this field is mandatory!"
```

Calculating the field sums

This example demonstrates how to calculate the sums of fields located at different levels of the form hierarchy when the form filler opens the form in a client application, such as Acrobat Professional, Adobe Reader, or HTML client.

NumericField1	3
NumericField1	3
NumericField1	3
Sum	9

To see this scripting example and others, visit the LiveCycle Developer Center at www.adobe.com/devnet/lifecycle.

► Scripting for calculating the sum of repeating fields in a form

To calculate the sum of repeating fields in a form, you add a `calculate` event to the Sum field:

```
var fields = xfa.resolveNodes("NumericField1[*]");

var total = 0;
for (var i=0; i <= fields.length-1; i++) {
    total = total + fields.item(i).rawValue;
}

this.rawValue = total;
```

► Scripting for calculating the sum of repeating fields

Similarly, to calculate the sum of repeating fields, you add a `calculate` event to the Sum field:

```
var fields = xfa.resolveNodes("detail[*].NumericField1");

var total = 0;
for (var i=0; i <= fields.length-1; i++) {
    total = total + fields.item(i).rawValue;
}

this.rawValue = total;
```

► Scripting to calculate the sum of the fields on the page

To calculate the sum of the fields on the page, you add a `calculate` event to the Sum field:

```
var fields = xfa.layout.pageContent(0, "field", 0);

var total = 0;
for (var i=0; i <= fields.length-1; i++) {
    if (fields.item(i).name == "NumericField1") {
        total = total + fields.item(i).rawValue;
    }
}
```

```
}  
  
this.rawValue = total;
```

Highlighting fields in response to form filler interaction

This example demonstrates how to highlight the current field that a form filler is working with, highlight fields that a form filler is required to fill, and use message boxes to provide feedback to the form filler.

In this example, an asterisk (*) appears to the right of the required fields. When a field is selected, the field border changes to blue. If the form filler clicks the Verify Data button without having filled the required fields, a message appears and the field changes to red. If all the required fields are filled, a confirmation message appears when the form filler clicks the Verify Data button.

Name: *

Address: *

City: * State: *

Country: * Zip Code: *

Telephone:

Email:

To see this scripting example and others, visit the LiveCycle Developer Center at www.adobe.com/devnet/lifecycle.

► Scripting for adding a blue border around a selected field

To add the blue border around the selected field, add the following scripts to each text field:

For example, add an `enter` event to the Name field:

```
Name.border.edge.color.value = "0,0,255";
```

For example, add an `exit` event to the Name field:

```
Name.border.edge.color.value = "255,255,255";
```

For example, add a `mouseenter` event to the Name field:

```
Name.border.edge.color.value = "0,0,255";
```

For example, add a `mouseleave` event to the Name field:

```
Name.border.edge.color.value = "255,255,255";
```

► Scripting for the Verify Data button

The following script, which is created for the Verify Data button, performs a series of checks to verify that the required fields contain data. In this case, each field is individually checked to verify that the value of the field is non-null or an empty string. If the value of the field is null or an empty string, an alert message appears indicating that data must be input into the field and the background color of the fillable area is changed to red.

Use this variable to indicate whether a field does not contain data:

```
var iVar = 0;

if ((Name.rawValue == null) || (Name.rawValue == "")) {
    xfa.host.messageBox("Please enter a value in the Name field.");
```

This script changes the color of the fillable area of the text field:

```
    xfa.resolveNode("Name.ui.#textEdit.border.edge").stroke = "solid";
    xfa.resolveNode("Name.ui.#textEdit.border.fill.color").value =
"255,100,50";

    // Set the variable to indicate that this field does not contain data.
    iVar = 1;
}
else {
    // Reset the fillable area of the text field.
    xfa.resolveNode("Name.ui.#textEdit.border.edge").stroke = "lowered";
    xfa.resolveNode("Name.ui.#textEdit.border.fill.color").value =
"255,255,255";
}

if ((Address.rawValue == null) || (Address.rawValue == "")) {
    xfa.host.messageBox("Please enter a value in the Address field.");
```

This script changes the color of the fillable area of the text field:

```
    xfa.resolveNode("Address.ui.#textEdit.border.edge").stroke = "solid";
    xfa.resolveNode("Address.ui.#textEdit.border.fill.color").value =
"255,100,50";
```

This script sets the variable to indicate that this field does not contain data:

```
    iVar = 1;
}
else {
```

This script resets the fillable area of the text field:

```
    xfa.resolveNode("Address.ui.#textEdit.border.edge").stroke = "lowered";
    xfa.resolveNode("Address.ui.#textEdit.border.fill.color").value =
"255,255,255";
}

if ((City.rawValue == null) || (City.rawValue == "")) {
    xfa.host.messageBox("Please enter a value in the City field.");
```

This script changes the color of the fillable area of the text field:

```
xfa.resolveNode("City.ui.#textEdit.border.edge").stroke = "solid";
xfa.resolveNode("City.ui.#textEdit.border.fill.color").value =
"255,100,50";
```

This script sets the variable to indicate that this field does not contain data:

```
iVar = 1;
}
else {
```

This script resets the fillable area of the text field:

```
xfa.resolveNode("City.ui.#textEdit.border.edge").stroke = "lowered";
xfa.resolveNode("City.ui.#textEdit.border.fill.color").value =
"255,255,255";
}

if ((State.rawValue == null) || (State.rawValue == "")) {
    xfa.host.messageBox("Please enter a value in the State field.");
}
```

This script changes the color of the fillable area of the text field:

```
xfa.resolveNode("State.ui.#textEdit.border.edge").stroke = "solid";
xfa.resolveNode("State.ui.#textEdit.border.fill.color").value =
"255,100,50";
```

This script sets the variable to indicate that this field does not contain data:

```
iVar = 1;
}
else {
```

This script resets the fillable area of the text field:

```
xfa.resolveNode("State.ui.#textEdit.border.edge").stroke = "lowered";
xfa.resolveNode("State.ui.#textEdit.border.fill.color").value =
"255,255,255";
}

if ((ZipCode.rawValue == null) || (ZipCode.rawValue == "")) {
    xfa.host.messageBox("Please enter a value in the Zip Code field.");
}
```

This script changes the color of the fillable area of the text field:

```
xfa.resolveNode("ZipCode.ui.#textEdit.border.edge").stroke = "solid";
xfa.resolveNode("ZipCode.ui.#textEdit.border.fill.color").value =
"255,100,50";
```

This script sets the variable to indicate that this field does not contain data:

```
iVar = 1;
}
else {
```

This script resets the fillable area of the text field:

```
xfa.resolveNode("ZipCode.ui.#textEdit.border.edge").stroke = "lowered";
```

```
xfa.resolveNode("ZipCode.ui.#textEdit.border.fill.color").value =  
"255,255,255";  
}  
  
if ((Country.rawValue == null) || (Country.rawValue == "")) {  
    xfa.host.messageBox("Please enter a value in the Country field.");  
}
```

This script changes the color of the fillable area of the text field:

```
xfa.resolveNode("Country.ui.#textEdit.border.edge").stroke = "solid";  
xfa.resolveNode("Country.ui.#textEdit.border.fill.color").value =  
"255,100,50";
```

This script sets the variable to indicate that this field does not contain data.

```
    iVar = 1;  
}  
else {
```

This script resets the fillable area of the text field.

```
    xfa.resolveNode("Country.ui.#textEdit.border.edge").stroke = "lowered";  
    xfa.resolveNode("Country.ui.#textEdit.border.fill.color").value =  
    "255,255,255";  
}
```

If all of the required fields contain data, the `iVar` variable is set to zero, and a confirmation message appears:

```
if (iVar == 0) {  
    xfa.host.messageBox("Thank you for inputting your information.");  
}
```

Resetting the values of the current subform

This example demonstrates how to reset the values of a specific set of fields, not the whole form. To do this, reset only the fields in the required subform object.

In this example, the form filler can click the Clear button to reset the field values.

1				Clear
2				Clear
3				Clear

To see this scripting example and others, visit the LiveCycle Developer Center
www.adobe.com/devnet/livecycle.

► Scripting for the values that appear in the left column

Type this script for the values appearing in the left column:

```
this.rawValue = this.parent.index + 1;
```

To reset the default values add a `click` event to the Clear button. You need a dynamic reference syntax expression because the detail is a repeating subform and must be reflected in the reference syntax expression. In this situation, it is easier to build the `resetData` parameters separately.

```
var f1 = this.parent.somExpression + ".TextField2" + ",";
var f2 = f1 + this.parent.somExpression + ".DropDownList1" + ",";
var f3 = f2 + this.parent.somExpression + ".NumericField1";

// ...and pass the variable as a parameter.
xfa.host.resetData(f3);
```

Changing the presence of a form design object

LiveCycle Designer provides the following presence settings for the different objects on a form through various tabs in the Object palette. The Invisible and Hidden (Exclude from Layout) settings are unavailable for groups, content areas, master pages, page set, and subform set objects.

Note: To change the presence setting of an object by using scripts, you must change the value of two underlying XML Form Object Model properties: `presence` and `relevant`.

The following table lists the presence settings and the corresponding reference syntax.

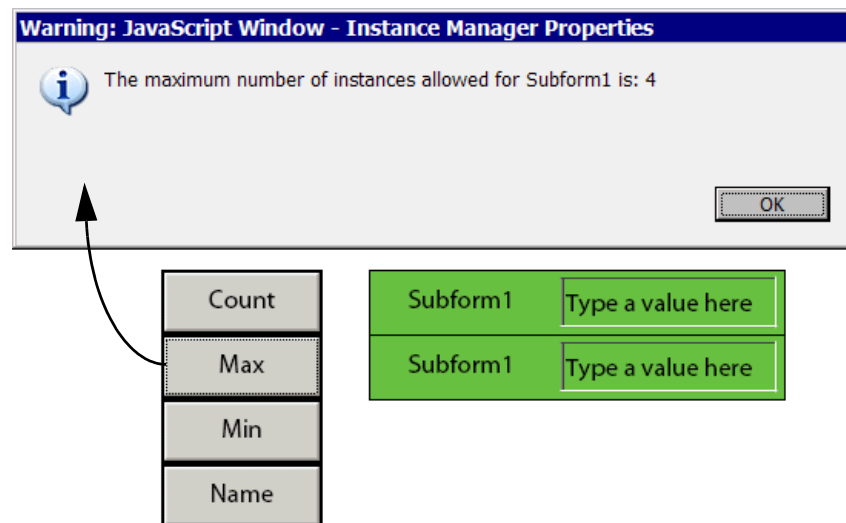
Presence setting	Reference syntax
Visible	<p>FormCalc</p> <pre>ObjectName.presence = "visible"</pre> <p>JavaScript</p> <pre>ObjectName.presence = "visible";</pre>
Visible (Screen Only)	<p>FormCalc</p> <pre>ObjectName.presence = "visible" ObjectName.relevant = "-print"</pre> <p>JavaScript</p> <pre>ObjectName.presence = "visible"; ObjectName.relevant = "-print";</pre>
Visible (Print Only)	<p>FormCalc</p> <pre>ObjectName.presence = "visible" ObjectName.relevant = "+print"</pre> <p>JavaScript</p> <pre>ObjectName.presence = "visible"; ObjectName.relevant = "+print";</pre>

Presence setting	Reference syntax
Invisible	FormCalc <code>ObjectName.presence = "invisible"</code> JavaScript <code>ObjectName.presence = "invisible";</code>
Hidden (Exclude from Layout)	FormCalc <code>ObjectName.presence = "hidden"</code> JavaScript <code>ObjectName.presence = "hidden";</code>

Using the properties of the instance manager to control subforms

This example demonstrates how to use the properties of the instance manager (which is part of the XML Form Object Model) to retrieve information about subforms at run time.

In the following form, the four buttons provide information about Subform1 by using the instance manager's scripting properties. For example, when the form filler clicks the Max button, a message describing the allowed maximum number of supported Subform1 instances appears.



► Scripting for the message box to output the value of the count property

The following script uses the `messageBox` method to output the value of the `count` property:

```
xfa.host.messageBox("The current number of Subform1 instances on the
form is:" + properties.Subform1.instanceManager.count, "Instance Manager
Properties",3);
```

You can also write this script by using the underscore (`_`) notation to reference the `count` property of the instance manager, as shown here:

```
xfa.host.messageBox("The current number of Subform1 instances on the form
is: " + properties._Subform1.count, "Instance Manager Properties", 3);
```

The underscore (_) notation is especially important if no subform instances currently exist on the form.

► Scripting for the message box to output the value of the max property

The following script uses the `messageBox` method to output the value of the `max` property:

```
xfa.host.messageBox("The maximum number of instances allowed for Subform1  
is: " + properties.Subform1.instanceManager.max, "Instance Manager  
Properties", 3);
```

You can also write this script by using the underscore (_) notation to reference the `max` property of the instance manager, as shown here:

```
xfa.host.messageBox("The maximum number of instances allowed for Subform1  
is: " + properties._Subform1.max, "Instance Manager Properties", 3);
```

► Scripting for the message box to output the value of the min property

The following script uses the `messageBox` method to output the value of the `min` property:

```
xfa.host.messageBox("The minimum number of instances allowed for Subform1  
is: " + properties.Subform1.instanceManager.min, "Instance Manager  
Properties", 3);
```

You can also write this script by using the underscore (_) notation to reference the `min` property of the instance manager, as shown here:

```
xfa.host.messageBox("The minimum number of instances allowed for Subform1  
is: " + properties._Subform1.min, "Instance Manager Properties", 3);
```

► Scripting for the message box to output the name of the subform property

The following script uses the `messageBox` method to output the name of the subform property:

```
xfa.host.messageBox("The name of the subform using the instance manager name  
property is: " + properties.Subform1.instanceManager.name +  
".\n\nNote: This value is different than the value returned by the name  
property for the Subform1 object." , "Instance Manager Properties", 3);
```

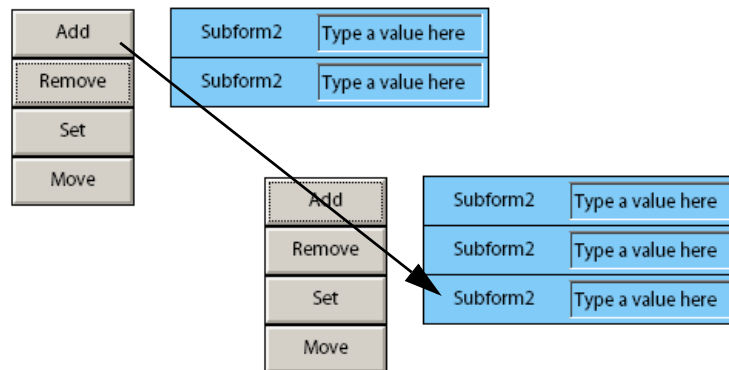
You can also write this script by using the underscore (_) notation to reference the `name` property of the instance manager, as shown here:

```
xfa.host.messageBox("The name of the subform using the instance manager name  
property is: " + properties._Subform1.name +  
".\n\nNote: This value is different than the value returned by the name  
property for the Subform1 object." , "Instance Manager Properties", 3);
```

Using the methods of the instance manager to control subforms

This example demonstrates how to use the methods of the instance manager (which is part of the XML Form Object Model) to perform operations on subform objects at run time. For example, you can add remove instances of a particular subform, table, or table row.

In the following form, the form filler uses the four buttons to use the various instance manager scripting methods. For example, when the form filler clicks the Add button a new Subform2 instance is added to the form.



Note: The Move button reorders the first two Subform2 instances, and the Set button displays the maximum number of Subform2 instances. In both cases, you may need to add or remove subforms, or make changes to the data in the text fields to see the changes applied to the Subform2 instances.

► Scripting to determine whether you added the maximum number of subforms to a form

The following script determines whether the supported maximum number of Subform2 instances exist on the form. If the maximum number exists, the script displays a message. Otherwise, a new Subform2 instance is added to the form.

```
if (methods.Subform2.instanceManager.count ==
methods.Subform2.instanceManager.max) {
    xfa.host.messageBox("You have reached the maximum number of items allowed.",
    "Instance Manager Methods", 1);
}
else {
    methods.Subform2.instanceManager.addInstance(1);
    xfa.form.recalculate(1);
}
```

You can also write this script by using the underscore (_) notation to reference the properties and methods of the instance manager, as shown here:

```
if (methods._Subform2.count == methods._Subform1.max) {
    xfa.host.messageBox("You have reached the maximum number of items
    allowed.", "Instance Manager Methods", 1);
}
else {
    methods._Subform2.addInstance(1);
    xfa.form.recalculate(1);
}
```

► Scripting to determine whether there are more subforms to remove on the form

The following script determines whether any Subform2 instances exist on the form. If none exist, the script displays a message indicating that no instances exist. If instances do exist, the script removes the first instance from the form.

```
if (methods.Subform2.instanceManager.count == 0) {
```

```
        xfa.host.messageBox("There are no subform instances to remove.",  
        "Instance Manager Methods", 1);  
    }  
    else {  
        methods.Subform2.instanceManager.removeInstance(0);  
        xfa.form.calculate(1);  
    }  
}
```

You can also write this script by using the underscore (_) notation to reference the properties and methods of the instance manager, as shown here:

```
if (methods._Subform2.count == 0) {  
    xfa.host.messageBox("There are no subform instances to remove.",  
    "Instance Manager Methods", 1);  
}  
else {  
    methods._Subform2.removeInstance(0);  
    xfa.form.calculate(1);  
}
```

► Scripting to force four subform instances to appear on the form

The following script forces four Subform2 instances to appear on the form regardless of how many instances currently exist:

```
methods.Subform2.instanceManager.setInstances(4);
```

You can also write this script by using the underscore (_) notation to reference the properties and methods of the instance manager, as shown here:

```
methods._Subform2.setInstances(4);
```

► Scripting to force the first and second subforms to switch locations on the form

The following script forces the first and second Subform2 instances to switch locations on the form.

```
methods.Subform2.instanceManager.moveInstance(0,1);
```

You can also write this script by using the underscore (_) notation to reference the properties and methods of the instance manager, as shown here.

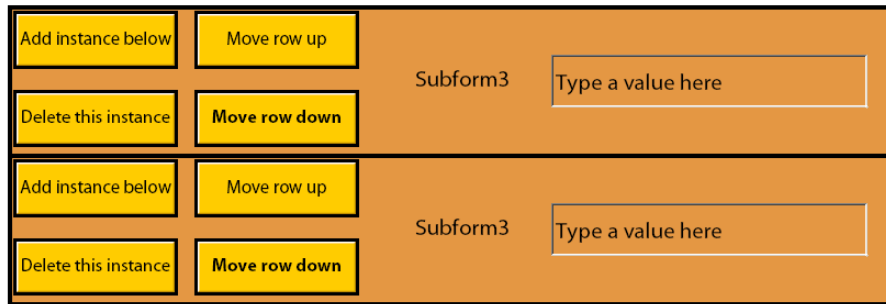
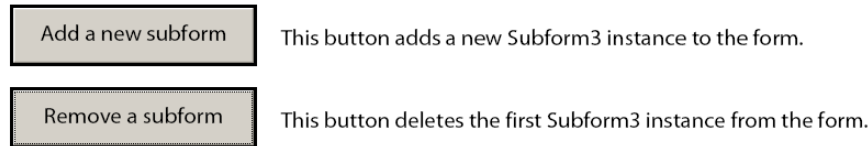
```
methods._Subform2.moveInstance(0,1);
```

Using the instance manager to control subforms at run time

This example demonstrates how to use properties and methods of the instance manager to retrieve information about subforms and perform operations on subform objects at run time.

In this example, the form filler uses the buttons to perform various actions using instances of Subform3. For example, when the form filler clicks the Add Row Below button a new Subform3 instance is added below the current instance.

Note: You may need to add or remove subforms, or make changes to the data in the text field, to see the changes applied to the instances of Subform3.



Tip: If no instances of a particular subform exist on your form, you must use the underscore (_) notation provided with each example below. For more information about using the underscore (_) notation, see the *LiveCycle Designer Help*.

► Scripting the Add a New Subform button

The following script determines whether the supported maximum number of Subform3 instances exist on the form. If the maximum number exist, the script displays a message. Otherwise, a new Subform3 instance is added to the form.

```
if (advanced.Subform3.instanceManager.count ==
    advanced.Subform3.instanceManager.max) {
    xfa.host.messageBox("You have reached the maximum number of items
        allowed.", "Combining Instance Manager Concepts", 1);
}
else {
    advanced.Subform3.instanceManager.addInstance(1);
    xfa.form.recalculate(1);
}
```

You can also write this script by using the underscore (_) notation to reference the properties and methods of the instance manager, as shown here:

```
if (advanced._Subform3.count == advanced._Subform3.max) {
    xfa.host.messageBox("You have reached the maximum number of items
        allowed.", "Combining Instance Manager Concepts", 1);
}
else {
    advanced._Subform3.addInstance(1);
    xfa.form.recalculate(1);
}
```

► Scripting the Remove a Subform button

The following script determines whether any Subform3 instances exist on the form. If none exist, the script displays a message indicating that no instances exist. If instances exist, the script removes the first instance from the form.

```
if (advanced.Subform3.instanceManager.count == 0) {  
    xfa.host.messageBox("There are no subform instances to remove.",  
        "Combining Instance Manager Concepts", 1);  
}  
else {  
    advanced.Subform3.instanceManager.removeInstance(0);  
}
```

You can also write this script by using the underscore (_) notation to reference the properties and methods of the instance manager, as shown here:

```
if (advanced._Subform3.count == 0) {  
    xfa.host.messageBox("There are no subform instances to remove.",  
        "Combining Instance Manager Concepts", 1);  
}  
else {  
    advanced._Subform3.removeInstance(0);  
}
```

► Scripting the Add Instance Below button

The following if-else statement prevents the script from proceeding if the form currently contains the maximum number of Subform3 instances:

```
if (Subform3.instanceManager.count < Subform3.instanceManager.occurences) {  
    //oNewInstance stores an instance of Subform3 created by the addInstance()  
    //method.  
    var oNewInstance = Subform3.instanceManager.addInstance(1);  
    //nIndexFrom and nIndexTo store the before and after index values to use with  
    //the moveInstance() method.  
    var nIndexFrom = oNewInstance.index;  
    var nIndexTo = Subform3.index + 1;
```

In this case, when the script references the value for nIndexFrom, the new instance of Subform3 is added to the form in the position specified in the moveInstance method:

```
    Subform3.instanceManager.moveInstance(nIndexFrom, nIndexTo);  
}  
else {  
    xfa.host.messageBox("You have reached the maximum number of items  
        allowed.", "Combining Instance Manager Concepts", 1);  
}
```

You can also write this script by using the underscore (_) notation to reference the properties and methods of the instance manager, as shown here:

```
if (_Subform3.count < _Subform3.occurences) {  
    var oNewInstance = _Subform3.addInstance(1);  
    var nIndexFrom = oNewInstance.index;  
    var nIndexTo = Subform3.index + 1;  
    _Subform3.moveInstance(nIndexFrom, nIndexTo);  
}
```

```
else {  
    xfa.host.messageBox("You have reached the maximum number of items allowed.",  
        "Combining Instance Manager Concepts", 1);  
}
```

► Scripting the Delete This Instance button

The following if-else statement prevents the script from proceeding if the form currently contains the minimum number of Subform3 instances.

```
if (Subform3.instanceManager.count > Subform3.instanceManager.occurences.min) {
```

This script uses the `removeInstance` method to remove an instance of Subform3.

Note: This script uses the value `parent.parent.index` to indicate the Subform3 instance to remove. The `parent` reference indicates the container of the object using the reference. In this case, using the reference `parent.index` would indicate the untitled subform that contains the Add Instance Below, Delete This Instance, Move Row Up, and Move Row Down buttons.

```
    Subform3.instanceManager.removeInstance(parent.parent.index);  
}  
else {  
    xfa.host.messageBox("You have reached the minimum number of items  
        allowed.", "Combining Instance Manager Concepts", 1);  
}
```

You can also write this script by using the underscore (`_`) notation to reference the properties and methods of the instance manager, as shown here:

```
if (_Subform3.count > _Subform3.occurences.min) {  
    Subform3.removeInstance(Subform3.index);  
}  
else {  
    xfa.host.messageBox("You have reached the minimum number of items allowed.",  
        "Combining Instance Manager Concepts", 1);  
}
```

► Scripting the Move Row Up button

The following if-else statement prevents the script from proceeding if the instance of Subform3 appears as the first instance in the list:

```
if (Subform3.index != 0) {  
    //nIndexFrom and nIndexTo store the before and after index values to use with  
    //the moveInstance method.  
    var nIndexFrom = Subform3.index;  
    var nIndexTo = Subform3.index - 1;  
    Subform3.instanceManager.moveInstance(nIndexFrom, nIndexTo);  
}  
else {  
    xfa.host.messageBox("The current item cannot be moved because it is the  
        first instance in the list.", "Combining Instance Manager Concepts", 1);  
}
```

You can also write this script by using the underscore (`_`) notation to reference the properties and methods of the instance manager, as shown here:

```
if (Subform3.index != 0) {
```

```
    var nIndexFrom = Subform3.index;  
    var nIndexTo = Subform3.index - 1;  
    Subform3.moveInstance(nIndexFrom, nIndexTo);  
  }  
  else {  
    xfa.host.messageBox("The current item can't be moved since it already is the  
first instance in the list.", "Combining Instance Manager Concepts", 1);  
  }
```

► Scripting the Move Row Down button

This variable stores the index value of the instance of Subform3:

```
var nIndex = Subform3.index;
```

The following if-else statement prevents the script from proceeding if the instance of Subform3 appears as the last instance in the list:

```
if ((nIndex + 1) < Subform3.instanceManager.count) {  
  // nIndexFrom and nIndexTo store the before and after index values to use with  
the moveInstance() method.  
  var nIndexFrom = nIndex;  
  var nIndexTo = nIndex + 1;  
  
  Subform3.instanceManager.moveInstance(nIndexFrom, nIndexTo);  
}  
else {  
  xfa.host.messageBox("The current item cannot be moved because it is the last  
instance in the list.", "Combining Instance Manager Concepts", 1);  
}
```

You can also write this script by using the underscore (_) notation to reference the properties and methods of the instance manager, as shown here:

```
var nIndex = Subform3.index;  
if ((nIndex + 1) < Subform3.instanceManager.count) {  
  var nIndexFrom = nIndex;  
  var nIndexTo = nIndex + 1;  
  _Subform3.moveInstance(nIndexFrom, nIndexTo);  
}  
else {  
  xfa.host.messageBox("The current item can't be moved since it already is the  
last instance in the list.", "Combining Instance Manager Concepts", 1);  
}
```

Index

- A**
- adding
 - FormCalc functions to objects 48
 - JavaScript scripts to objects 52
 - scripts to script objects 65
 - Adobe Acrobat
 - converting forms containing scripts 77
 - JavaScript object model 77
 - Adobe LiveCycle Designer
 - converting Acrobat forms to 77
 - scripting, moving from Acrobat 76
 - application events 27
 - Arabic characters, displaying 13
 - array referencing 63
- B**
- basic calculations 49
 - built-in functions, FormCalc 47
- C**
- calculate event 28
 - calculations
 - about 9
 - about creating 16
 - adding object reference syntax to 20
 - attaching to objects 16
 - creating 18, 19
 - current object, referencing 60
 - debugging 14, 23, 67
 - event model properties and methods 74
 - referencing objects in 56, 58, 60
 - running 20, 23
 - testing 23
 - variables 54
 - change event 29
 - click event 29
 - colors, background, changing 91
 - comparing FormCalc and JavaScript 41
 - creating
 - calculations and scripts 18, 19
 - script objects 65
- D**
- debugging
 - about calculations and scripts 67
 - calculations and scripts 14, 23
 - enabling the JavaScript Debugger 68
 - executing JavaScript scripts 69
 - using the JavaScript Console 70
 - using the messageBox method 67
 - default
 - processing application, setting 13
 - scripting language, setting 12, 13
 - defining text variables 53
 - displaying
 - international characters 13
 - Script Editor 12
 - docClose event 30
 - docReady event 31
 - drop-down lists, value of, obtaining 100
- E**
- enter event 31
 - event model
 - about 74
 - scripting properties and methods 74
 - events
 - about 20, 24
 - application events 27
 - calculate (processing event) 28
 - change (interactive event) 29
 - click (interactive event) 29
 - docClose (application event) 30
 - docReady (application event) 31
 - enter (interactive event) 31
 - exit (interactive event) 32
 - form:ready (processing event) 32
 - full (interactive event) 33
 - indexChange (processing event) 33
 - initialize (processing event) 34
 - interactive 25
 - layout:ready (processing event) 34
 - mouseDown (interactive event) 35
 - mouseenter (interactive event) 35
 - mouseleave (interactive events) 36
 - mouseup (interactive event) 36
 - postPrint (application event) 37
 - postSave (application event) 37
 - preOpen (interactive event) 37
 - prePrint (application event) 38
 - preSave (application event) 38
 - preSubmit (application event) 39
 - processing events 24
 - scripting, viewing 21
 - scripts associated with 20
 - types of 24
 - validate (processing event) 40
 - executing
 - events 24
 - scripts in JavaScript Console 69
 - exit event 32
 - expressions, Acrobat-equivalent 72
-

F

- fields
 - at run time, setting as required 102
 - background color, changing 91
 - height, adjusting at run time 102
 - highlighting 104
 - sums, calculating 103
- form designs
 - object naming conventions 16
 - objects, changing presence of 108
- form:ready event 32
- FormCalc
 - about 17, 47
 - basic calculations 49
 - built-in functions 47
 - function syntax 48
 - functions 41
 - functions, adding to objects 48
 - reference syntax shortcuts 60
 - referencing objects 56, 58, 60
- forms, converting from Acrobat to LiveCycle Designer 77
- full event 33
- function syntax, FormCalc 48

H

- Hebrew characters, displaying 13
- hiding objects 93
- host application 71

I

- indexChange event 33
- initialize event 34
- initiating events 24
- instance manager
 - controlling subforms at run time 112
 - methods, controlling subforms using 110
 - properties, controlling subforms using 109
- interactive events 25

J

- JavaScript
 - about 17, 51
 - adding scripts to objects 52
 - enabling 68
 - functions 41
 - objects supported from Acrobat 78
 - referencing objects 56, 58, 60
 - using objects from Acrobat 77
- JavaScript Console
 - about 67
 - debugging 70
 - executing scripts in 69
- JavaScript Debugger
 - about 67
 - enabling for LiveCycle Designer 68
 - prevent from disappearing 69

L

- layout:ready event 34

M

- messageBox method 67
- mouseDown event 35
- mouseenter event 35
- mouseExit event 36
- mouseUp event 36
- multiline view, Script Editor 11, 12

N

- naming
 - form design objects and variables 16
 - variables 53

O

- objects
 - attaching calculations and scripts to 16
 - calculation and script support 10
 - making visible and invisible 108
 - naming conventions 16
 - referencing in calculations and scripts 56, 58, 60
 - showing and hiding 93
 - viewing attached scripts 21
 - visual properties, changing 97
- order of events 24

P

- postPrint event 37
- postSave event 37
- preOpen event 37
- prePrint event 38
- preSave event 38
- presence property 108
- preSubmit event 39
- processing application, default 13
- processing events 24
- properties, visual, changing for objects 97

R

- reference syntax
 - about 56
 - adding to calculations and scripts 20
 - referencing object properties and values 56
 - referencing objects 58, 60
 - shortcuts for FormCalc 60
- referencing script objects 66

S

- Script Editor
 - about 11
 - displaying 12
 - finding and replacing text in 18
 - switching between single-line and multiline view 12
 - viewing scripts and events in 21

- script objects 65, 66
- scripting
 - about examples 91
 - against JavaScript object model in Acrobat 77
 - event model properties and methods 74
 - moving from Adobe Acrobat 76
- scripting language, setting 12, 13, 17
- scripting methods
 - Acrobat-equivalent 72
 - host scripting model 71
 - to control subforms 110
- scripting properties
 - Acrobat-equivalent 72
 - host scripting model 71
 - to control subforms 109
- scripts
 - about 9
 - about creating 16
 - about debugging and testing 67
 - adding object reference syntax to 20
 - associating with events 20
 - attached to objects, viewing 21
 - attaching to objects 16
 - creating 18, 19
 - current object, referencing 60
 - debugging 14, 23
 - executing in the JavaScript Console 69
 - referencing objects in 56, 58, 60
 - running 20, 23
 - testing 23
 - variables 54
- sequence of events 24
- setting
 - default processing application 13
 - default scripting language 12, 13
- shortcuts, reference syntax 60
- showing
 - Script Editor 12
 - scripting events and scripts 21
- showing objects 93

- simple expressions 49
- single-line view, Script Editor 11, 12
- subforms
 - background color, changing 91
 - controlling at run time 109, 110, 112
 - value, resetting 107

T

- testing
 - calculations and scripts 23
 - JavaScript scripts 67
- text variables
 - See also* variables
 - defining 53
 - deleting 54
 - viewing definition 54
- text, finding and replacing in XML Source tab and Script Editor 18
- Thai characters, displaying 13
- triggering events 24

U

- unnamed objects, referencing in calculations and scripts 58

V

- validate event 40
- variables
 - See also* text variables
 - about 53
 - in calculations and scripts 54
 - naming 53
 - naming conventions 16
- Vietnamese characters, displaying 13

X

- XML Form Object Model
 - about 56
 - instance manager 109
- XML Source tab, finding and replacing text in 18