ADOBE® INDESIGN® CS6

ADOBE INDESIGN CS6 SCRIPTING GUIDE: JAVASCRIPT

Adobe

© 2012 Adobe Systems Incorporated. All rights reserved.

Adobe® InDesign® CS6 Scripting Guide: JavaScript

Document Update Status (for entire document; see each chapter for chapter-specific update status)		
CS6	Updated	Throughout document, changed CS5 to CS6 and version 7.0 to 8.0.

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 in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Creative Suite, and InDesign are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Mac OS is a trademark of Apple Computer, Incorporated, registered in the United States and other countries. All other trademarks are the property of their respective owners.

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

1 Introduction 10

How to Use the Scripts in this Document 10

About the Structure of the Scripts 11

Automatic jsxbin Conversion 11

Other JavaScript development options 11

For More Information 12

2 Scripting Features 13

Script Preferences 13

Getting the Current Script 14

Script Versioning 14

- Targeting 15
- Compilation 15
- Interpretation 15

Using the doScript Method 16

- Sending parameters to doScript 16
- Returning values from doScript 16

Controlling Undo with doScript 18

Working with Script Labels 18

Running Scripts at Startup 20

Session and Main Script Execution 20

3 Documents 21

Basic Document Operations 22

- Creating a new document 22
- Opening a document 22
- Saving a document 23
- Closing a document 24

Basic Page Layout 24

- Defining page size and document length 24
- Defining bleed and slug areas 25
- Setting page margins and columns 26
- Changing the appearance of the pasteboard 28
- Guides and grids 28
- Changing measurement units and ruler 30
- Defining and applying document presets 31
- Setting up master spreads 33
- Adding XMP metadata 35
- Creating a document template 35
- Creating watermarks 39

Adjusting Page Sizes and Layout 41
Selecting pages 41
Resizing and reframing pages 41
Transforming pages 42
Master page overlay 43

Making an Adaptive Layout 44
Making a flexible layout 44
Adding guides for a guide-based layout 45
Setting constraints for an object-based layout 46
Creating alternative layouts 46

Collecting and Dropping Content 47

Printing a Document 47
Printing using page ranges 48
Setting print preferences 48
Printing with printer presets 51

Exporting a Document as PDF 51
Exporting to PDF 51
Setting PDF export options 52
Exporting to grayscale PDF 53
Exporting a range of pages to PDF 54
Exporting individual pages to PDF 54
Exporting PDF with interactive features 55
Exporting as a PDF form 55

Exporting Pages as EPS 58
Exporting all pages to EPS 58
Exporting a range of pages to EPS 58
Exporting as EPS with file naming 58

Exporting to EPub 59
Exporting the current document 60
Setting EPub export options 60

4 Working with Layers 62

Understanding the Layer Object Model 62

Scripting Layers 63
Creating layers 63
Referring to layers 63
Deleting layers 65
Moving layers 65
Duplicating layers 65
Merging layers 65
Assigning page items to layers 66
Setting layer properties 66

5 Text and Type 68

Entering and Importing Text 68
Creating a text frame 68
Adding text 69
Stories and text frames 69
Replacing text 70

Inserting special characters 70
Placing Text and Setting Text-Import Preferences 71
Exporting Text and Setting Text-Export Preferences 74
Understanding Text Objects 78
Working with text selections 80
Moving and copying text 80
Text objects and iteration 82
Working with Text Frames 83
Linking text frames 83
Unlinking text frames 84
Removing a frame from a story 84
Splitting all frames in a story 86
Creating an anchored frame 86
Fitting text frames to content 87
Formatting Text 88
Setting text defaults 88
Working with fonts 91
Applying a font 91
Changing text properties 92
Changing text color 94
Creating and applying styles 95
Deleting a style 97
Importing paragraph and character styles 97
Finding and Changing Text 98
About find/change preferences 98
Finding and changing text 99
Finding and changing text formatting 99
Using grep 100
Using glyph search 102
Working with Tables 102
Adding Path Text 106
Using Autocorrect 106
Adding Footnotes 106
Spanning Columns 107
Setting Text Preferences 107
Working with Linked Stories 108
Creating linked stories 108
Changing linked story options 109
Frequently asked questions 109

6 Working with Page Items 110

Creating Page Items 110
Page-item geometry 112
Grouping Page Items 113
Duplicating and Moving Page Items 113
Creating compound paths 114

Using Pathfinder operations . . . 115
Converting page-item shapes . . . 116
Arranging page items . . . 116

Transforming Page Items . . . 116
Using the transform method . . . 116
Working with transformation matrices . . . 117
Coordinate spaces . . . 119
Transformation origin . . . 120
Resolving locations . . . 121
Transforming points . . . 122
Transforming again . . . 123

Resize and Reframe . . . 123

Working with Articles . . . 124
Retrieving an article list . . . 124
Adding new articles . . . 124
Removing an article . . . 124
Reordering articles . . . 124
Adding new members to an article . . . 125
Reordering members of an article . . . 125
Removing members from an article . . . 126

7 User Interfaces . . . 127

Dialog Overview . . . 127

Your First InDesign Dialog . . . 129

Adding a User Interface to "Hello World" . . . 129

Creating a More Complex User Interface . . . 130

Working with ScriptUI . . . 132
Creating a progress bar with ScriptUI . . . 133

8 Events . . . 134

Understanding the Event Scripting Model . . . 134
About event properties and event propagation . . . 134

Working with Event Listeners . . . 135

Sample afterNew Event Listener . . . 137

Sample beforePrint Event Listener . . . 139

Sample Selection Event Listeners . . . 140

Sample onIdle Event Listener . . . 141

9 Menus . . . 144

Understanding the Menu Model . . . 144
Localization and menu names . . . 146

Running a Menu Action from a Script . . . 147

Adding Menus and Menu Items . . . 147

Menus and Events . . . 148

Working with scriptMenuActions 149
A More Complex Menu-scripting Example 150

10 Working with Preflight 156

Exploring Preflight Profiles 156
Listing preflight profiles 156
Listing preflight rules 157
Listing preflight data objects 157
Importing a Preflight Profile 158
Creating a Preflight Profile 159
Adding Rules 161
Processing a Profile 162
Custom Rules 163
Available Rules 163
ADBE_BlankPages 164
ADBE_BleedSlug 165
ADBE_BleedTrimHazard 166
ADBE_Colorspace 166
ADBE_CrossReferences 166
ADBE_FontUsage 166
ADBE_ImageColorManagement 167
ADBE_ImageResolution 167
ADBE_PageCount 168
ADBE_PageSizeOrientation 168
ADBE_ScaledGraphics 168
ADBE_ScaledType 168
ADBE_SmallText 168
ADBE_SpotColorSetup 169
ADBE_StrokeRequirements 169
ADBE_TextOverrides 169
ADBE_TransparencyBlending 169

11 Creating Dynamic Documents 170

Importing Movies and Sounds 170
Creating Buttons 171
Creating Multistate Objects 173
Working with Animation 175
Basic animation 175
TimingSettings 176
Animating transformations 179
Motion presets 179
Design options 180
Key frames 180
Adding Page Transitions 181

12 XML 182

Overview 182

The Best Approach to Scripting XML in InDesign 182

Scripting XML Elements 183
- Setting XML preferences 183
- Setting XML import preferences 183
- Importing XML 184
- Creating an XML tag 184
- Loading XML tags 185
- Saving XML tags 185
- Creating an XML element 185
- Moving an XML element 185
- Deleting an XML element 185
- Duplicating an XML element 185
- Removing items from the XML structure 186
- Creating an XML comment 186
- Creating an XML processing instruction 186
- Working with XML attributes 186
- Working with XML stories 187
- Exporting XML 187

Adding XML Elements to a Layout 188
- Associating XML elements with page items and text 188
- Marking up existing layouts 189
- Applying styles to XML elements 191
- Working with XML tables 192

13 XML Rules 195

Overview 195
- Why use XML rules? 196
- XML-rules programming model 196

XML Rules Examples 202
- Setting up a sample document 202
- Getting started with XML rules 203
- Changing the XML structure using XML rules 207
- Duplicating XML elements with XML rules 208
- XML rules and XML attributes 209
- Applying multiple matching rules 211
- Finding XML elements 212
- Extracting XML elements with XML rules 215
- Applying formatting with XML rules 216
- Creating page items with XML rules 219

Creating Tables using XML Rules 220

Scripting the XML-rules Processor Object 221

14 Track Changes 223

Tracking Changes 223
- Navigating tracked changes 223
- Accepting and reject tracked changes 224

Information about tracked changes 224
Preferences for Tracking Changes 225

1 Introduction

Chapter Update Status		
CS6	Minor update	Updated "Other JavaScript development options" on page 11 to remove reference to CS5.

This document shows how to do the following:

- Work with the Adobe® InDesign® scripting environment.
- Use advanced scripting features.
- Perform basic document tasks like setting up master spreads, printing, and exporting.
- Work with page items (rectangles, ellipses, graphic lines, polygons, text frames, and groups).
- Work with text and type in an InDesign document, including finding and changing text.
- Create dialog boxes and other user-interface items.
- Customize and add menus and create menu actions.
- Respond to user-interface events.
- Work with XML, from creating XML elements and importing XML to adding XML elements to a layout.
- Apply XML rules, a new scripting feature that makes working with XML in InDesign faster and easier.

We assume that you have already read the *Adobe InDesign Scripting Tutorial* and know how to create, install, and run scripts. If you need to know how to connect with your scripting environment or view the InDesign scripting object model from your script editor, that information can be found in the *Adobe InDesign Scripting Tutorial*.

How to Use the Scripts in this Document

For the most part, the scripts shown in this document are not complete scripts. They are only fragments of scripts, and are intended to show only the specific part of a script relevant to the point being discussed in the text. You can copy the script lines shown in this document and paste them into your script editor, but you should not expect them to run without further editing. Note, in addition, that scripts copied out of this document may contain line breaks and other characters (due to the document layout) that will prevent them from executing properly.

A zip archive of all of the scripts shown in this document is available at the InDesign scripting home page, at: http://www.adobe.com/products/indesign/scripting/index.html. After you have downloaded and expanded the archive, move the folders corresponding to the scripting language(s) of your choice into the Scripts Panel folder inside the Scripts folder in your InDesign folder. At that point, you can run the scripts from the Scripts panel inside InDesign.

About the Structure of the Scripts

The script examples are all written using a common template that includes the functions "main," "mySetup," "mySnippet," and "myTeardown." We did this to simplify automated testing and publication—there is no reason for you to construct your scripts this way. Most of the time, the part of the script you will be interested in will be inside the "mySnippet" function.

Automatic jsxbin Conversion

The jsxbin format (Binary JavaScript) enables distributing a script without exposing its source code. The ExtendScript ToolKit offers an Export to Binary feature that instantly converts a js(x) file into jsxbin. The jsxbin export process can be done automatically without user interaction.

To do this:

1. Write a compiler script file that compiles .jsx files into a .jsxbin file.
   - ▷ The script file must start with this target directive:

     ```
     #target estoolkit#dbg
     ```

   - ▷ Use the `app.compile()` method and pass the content of the script file as a String parameter.
   - ▷ In the script, store the result in a new file with the extension .jsxbin.
2. Launch the ExtendScript ToolKit with the command-line parameter -cmd [path to compiler script file]:

   Windows:

   ```
   "ExtendScript Toolkit.exe" -cmd compiler.jsx
   ```

   Mac:

   ```
   "ExtendScript Toolkit.app/Contents/MacOS/ExtendScript Toolkit" \
            -cmd compiler.jsx
   ```

Here is a sample compiler script (compiler.jsx) with the source script's file path hardcoded.

```
#target estoolkit#dbg
var fileIn = File("/c/jsxbin_test/source.jsx");
fileIn.open("r");
var s = fileIn.read();
fileIn.close();
var t = app.compile(s);
var fileOut = File( fileIn.absoluteURI + "bin" ) ;
fileOut.open("w");
fileOut.write(t);
fileOut.close();
```

Other JavaScript development options

You can use the ExtendScript Toolkit to create JavaScript scripts explicitly for InDesign, or you can use the Creative Suite Extension Builder (CS Extension Builder) to develop *CS extensions* in ActionScript. CS extensions are Flash-based (SWF) and can potentially work in a variety of Creative Suite applications.

Applications have an extensibility infrastructure that allows developers to extend the capabilities of the applications; the infrastructure is based on Flash/Flex technology, and each CS extension is delivered as a compiled Flash (SWF) file. CS includes the Extension Manager to enable installation of CS extensions.

An example of a CS extension that ships with the point products is Adobe Kuler. Kuler has a consistent user interface across the different suite applications, but has different logic in each, adapted to the host application.

The user interface for an extension is written in ActionScript, using the Flex framework. A C5 extension is typically accessed through its own menu item in the application's Extensions menu. CS Extension Builder allows you to design the user interface interactively using the Design view of FlashBuilder. It also allows you to develop all of the application logic for your CS extension in ActionScript; you can develop and debug your extension in the familiar FlashBuilder environment.

To develop your application logic, we recommend using the Creative Suite ActionScript Wrapper Library (`CSAWLib`), which exposes the scripting DOM of each host application as an ActionScript library. This is tightly integrated with the CS Extension Builder environment, which includes wizards to help you build your extension's basic structure, and run and debug your code against suite applications such as Adobe InDesign, Photoshop and Illustrator.

The methods, properties, and behavior of the scripting DOM is as described in the *JavaScript Scripting Reference* for the host application. For details of how to use CS Extension Builder and the wrapper libraries, see the Creative Suite SDK documentation, which is accessible from within the Flash Builder or Eclipse Help system when you have installed CS Extension Builder.

For More Information

For more information on InDesign scripting, you also can visit the InDesign Scripting User to User forum, at http://www.adobeforums.com. In the forum, script writers can ask questions, post answers, and share their newest scripts. The forum contains hundreds of sample scripts.

2 Scripting Features

Chapter Update Status		
CS6	Updated	"Script Versioning" on page 14 and its three subsections have been updated, corrected, and clarified.

This chapter covers scripting techniques related to InDesign's scripting environment. Almost every other object in the InDesign scripting model controls a feature that can change a document or the application defaults. By contrast, the features in this chapter control how scripts operate.

This document discusses the following:

- The `scriptPreferences` object and its properties.
- Getting a reference to the executing script.
- Running scripts in prior versions of the scripting object model.
- Using the `doScript` method to run scripts.
- Working with script labels.
- Running scripts at InDesign start-up.
- Controlling the ExtendScript engine in which scripts execute.

We assume that you have already read *Adobe InDesign Scripting Tutorial* and know how to write, install, and run InDesign scripts in the scripting language of your choice.

Script Preferences

The `scriptPreferences` object provides objects and properties related to the way InDesign runs scripts. The following table provides more detail on each property of the scriptPreferences object:

Property	Description
`enableRedraw`	Turns screen redraw on or off while a script is running from the Scripts panel.
`scriptsFolder`	The path to the scripts folder.
`scriptsList`	A list of the available scripts. This property is an array of arrays, in the following form: `[[fileName, filePath], ...]` Where *`fileName`* is the name of the script file and *`filePath`* is the full path to the script. You can use this feature to check for the existence of a script in the installed set of scripts.

Property	Description
`userInteractionLevel`	This property controls the alerts and dialogs InDesign presents to the user. When you set this property to `UserInteractionLevels.neverInteract`, InDesign does not display any alerts or dialogs. Set it to `UserInteractionLevels.interactWithAlerts` to enable alerts but disable dialogs. Set it to `interactWithAll` to restore the normal display of alerts and dialogs. The ability to turn off alert displays is very useful when you are opening documents via script; often, InDesign displays an alert for missing fonts or linked graphics files. To avoid this alert, set the user-interaction level to `UserInteractionLevels.neverInteract` before opening the document, then restore user interaction (set the property to `interactWithAll`) before completing script execution.
`version`	The version of the scripting environment in use. For more information, see "Script Versioning" on page 14. Note this property is *not* the same as the version of the application.

Getting the Current Script

You can get a reference to the current script using the `activeScript` property of the application object. You can use this property to help you locate files and folders relative to the script, as shown in the following example (from the ActiveScript tutorial script):

```
var myScript = app.activeScript;
alert("The current script is: " + myScript);
var myParentFolder = File(myScript).parent;
alert("The folder containing the active script is: " + myParentFolder);
```

When you debug scripts using a script editor, the `activeScript` property returns an error. Only scripts run from the Scripts palette appear in the `activeScript` property.

When you debug scripts from the ExtendScript Toolkit, using the `activeScript` property returns an error. To avoid this error and create a way of debugging scripts that use the `activeScript` property, use the following error handler (from the GetScriptPath tutorial script):

```
function myGetScriptPath() {
    try{
        return app.activeScript;
    }
    catch(myError){
        return File(myError.fileName);
    }
}
```

Script Versioning

InDesign can run scripts using earlier versions of the InDesign scripting object model. To run an older script in a newer version of InDesign, you must consider the following:

- **Targeting —** Scripts must be targeted to the version of InDesign in which they are being run (i.e., the current version). The mechanics of targeting are language specific as described in "Targeting" on page 15.

- **Compilation —** This involves mapping the names in the script to the underlying script IDs, which are what InDesign understands. The mechanics of compilation are language specific as described in "Compilation" on page 15.
- **Interpretation —** This involves matching the IDs to the appropriate request handler within InDesign so that InDesign correctly interprets a script written for an earlier version of the scripting object model. To do this, either explicitly set the application's script preferences to the old object model within the script (as shown in "Interpretation" on page 15) or run the script from a folder in the Scripts panel folder as follows:

Folder	For InDesign version of scripts
`Version 8.0 Scripts`	CS6
`Version 7.0 Scripts`	CS5 and CS5.5
`Version 6.0 Scripts`	CS4
`Version 5.0 Scripts`	CS3
`Version 2.0 Scripts`	CS2

Targeting

A script must always target the version of InDesign under which it is running (the current version), either explicitly or implicitly. Targeting is implicit when the script is launched from the Scripts panel.

Otherwise, if the script is launched externally (from the ESTK), explicit targeting for JavaScripts is done using the `target` directive:

```
//target CS6
#target "InDesign-8.0"
//target the latest version of InDesign
#target "InDesign"
```

Compilation

JavaScripts are not precompiled. For compilation, InDesign uses the same version of the DOM that is set for interpretation.

Interpretation

The InDesign application object contains a `scriptPreferences` object, which allows a script to get/set the version of the scripting object model to use for interpreting scripts. The version defaults to the current version of the application and persists.

For example, to change the version of the scripting object model to CS5 (7.0):

```
//Set to 7.0 scripting object model
app.scriptPreferences.version = 7.0;
```

Using the doScript Method

The `doScript` method gives a script a way to execute another script. The script can be a string of valid scripting code or a file on disk. The script can be in the same scripting language as the current script or another scripting language. The available languages vary by platform: on Mac OS®, you can run AppleScript or JavaScript; on Windows®, VBScript or JavaScript.

The `doScript` method has many possible uses:

- Running a script in another language that provides a feature missing in your main scripting language. For example, VBScript lacks the ability to display a file or folder browser, which JavaScript has. AppleScript can be very slow to compute trigonometric functions (sine and cosine), but JavaScript performs these calculations rapidly. JavaScript does not have a way to query Microsoft® Excel for the contents of a specific spreadsheet cell, but both AppleScript and VBScript have this capability. In all these examples, the `doScript` method can execute a snippet of scripting code in another language, to overcome a limitation of the language used for the body of the script.
- Creating a script "on the fly." Your script can create a script (as a string) during its execution, which it can then execute using the `doScript` method. This is a great way to create a custom dialog or panel based on the contents of the selection or the attributes of objects the script creates.
- Embedding scripts in objects. Scripts can use the `doScript` method to run scripts that were saved as strings in the `label` property of objects. Using this technique, an object can contain a script that controls its layout properties or updates its content according to certain parameters. Scripts also can be embedded in XML elements as an attribute of the element or as the contents of an element. See "Running Scripts at Startup" on page 20.

Sending parameters to doScript

To send a parameter to a script executed by `doScript`, use the following form (from the DoScriptParameters tutorial script):

```
var myParameters = ["Hello from DoScript", "Your message here."];
var myJavaScript = "alert(\"First argument: \" + arguments[0] + \"\\rSecond argument:
\" + arguments[1]);";
app.doScript(myJavaScript, ScriptLanguage.javascript, myParameters);
if(File.fs == "Windows"){
    var myVBScript = "msgbox arguments(1), vbOKOnly, \"First argument: \" &
    arguments(0)";
    app.doScript(myVBScript, ScriptLanguage.visualBasic, myParameters);
}
else{
    var myAppleScript = "tell application \"Adobe InDesign CS6\\rdisplay
    dialog(\"First argument\" & item 1 of arguments & return & \"Second argument: \"
    & item 2 of arguments & return & end tell";
    app.doScript(myAppleScript, ScriptLanguage.applescriptLanguage, myParameters);
}
```

Returning values from doScript

The following script fragment shows how to return a value from a script executed by `doScript`. This example uses a JavaScript that is executed as a string, but the same method works for script files. This example returns a single value, but you can return multiple values by returning an array (for the complete script, refer to the DoScriptReturnValues script).

```
//To send parameters to a script run using app.doScript(), the doScript
//statement must not appear inside a function. If it does, the parameters
//will not be passed to the script.
var myDocument = app.documents.add();
var myPage = myDocument.pages.item(0);
var myTextFrame = myPage.textFrames.add();
myTextFrame.geometricBounds = ["72pt", "72pt", "288pt", "288pt"];
myTextFrame.contents = "Example text frame.";
var myDestinationPage = myDocument.pages.add(LocationOptions.after, myPage);
var myPageIndex = myDestinationPage.name;
var myID = myTextFrame.id;
var myJavaScript = "var myDestinationPage = arguments[1];\r" ;
myJavaScript += "myID = arguments[0];\r";
myJavaScript += "var myX = arguments[2];\r";
myJavaScript += "var myY = arguments[3]\r;"
myJavaScript += "var myPageItem =
app.documents.item(0).pages.item(0).pageItems.itemByID(myID);\r";
myJavaScript +=
"myPageItem.duplicate(app.documents.item(0).pages.item(myDestinationPage));\r"
//Create an array for the parameters we want to pass to the JavaScript.
var myArguments = [myID, myPageIndex, 0, 0];
var myDuplicate = app.doScript(myJavaScript, ScriptLanguage.javascript, myArguments);
//myDuplicate now contains a reference to the duplicated text frame.
//Change the text in the duplicated text frame.
myDuplicate.contents = "Duplicated text frame.";
```

Another way to get values from another script is to use the `scriptArgs` (short for "script arguments") object of the application. The following script fragment shows how to do this (for the complete script, see DoScriptScriptArgs):

```
var nameA = "ScriptArgumentA";
var nameB = "ScriptArgumentB";
var nAc = nameA + ": ";
var nBc = nameB + ": ";
//Create a string to be run as a JavaScript.
var p1 = "app.scriptArgs.setValue(\"" + nameA + "\", ";
var p2 = "\"This is the first script argument value.\");\r";
var p3 = "app.scriptArgs.setValue(\"" + nameB + "\", ";
var p4 = "\"This is the second script argument value.\")";
var p5, p6; //Used later.
var myJavaScript = p1 + p2 + p3 + p4;
var myScriptArgumentA = app.scriptArgs.getValue(nameA);
var myScriptArgumentB = app.scriptArgs.getValue(nameB);
alert(nameA + ": " + myScriptArgumentA + "\r" + nameB + ": " + myScriptArgumentB);
if(File.fs == "Windows") {
    //Create a string to be run as a VBScript.
    p1 = "Set myInDesign = CreateObject(\"InDesign.Application.CS6\")\r";
    p2 = "myInDesign.ScriptArgs.SetValue \"" + nameA + "\", ";
    p3 = "\"This is the first script argument value.\"\r";
    p4 = "myInDesign.ScriptArgs.SetValue \"" + nameB + "\", ";
    p5 = "\"This is the second script argument value.\"";
    var myVBScript = p1 + p2 + p3 + p4 + p5;
    app.doScript(myVBScript, ScriptLanguage.visualBasic);
} else {
```

```
    //Create a string to be run as a AppleScript.
    p1 = "tell application \"Adobe InDesign CS6\"\r";
    p2 = "tell script args" +"\r";
    p3 = "set value name\"" + nameA +"\" value ";
    p4 = "\"This is the firest AppleScript script argument value.\"" +"\r";
    p5 = "set value name\"" + nameB +"\" value ";
    p6 = "\"This is the second AppleScript script argument value.\"" +"\r";
    p7 = "end tell" + "\r"
    p8 = "end tell" + "\r"
    var myAppleScript = p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8;
    app.doScript(myAppleScript, ScriptLanguage.applescriptLanguage);
}
var myScriptArgumentA = app.scriptArgs.getValue(nameA);
var myScriptArgumentB = app.scriptArgs.getValue(nameB);
alert(nAc + myScriptArgumentA + nBc + myScriptArgumentB);
```

Controlling Undo with doScript

InDesign gives you the ability to undo almost every action, but this comes at a price: for almost every action you make, InDesign writes to disk. For normal work you using the tools presented by the user interface, this does not present any problem. For scripts, which can perform thousands of actions in the time a human being can blink, the constant disk access can be a serious drag on performance.

The `doScript` method offers a way around this performance bottleneck by providing two parameters that control the way that scripts are executed relative to InDesign's Undo behavior. These parameters are shown in the following examples:

```
//Given a script "myJavaScript" and an array of parameters "myParameters"...
app.doScript(myJavaScript, ScriptLanguage.javascript, myParameters,
UndoModes.fastEntireScript, "Script Action");
//UndoModes can be:
//UndoModes.autoUnto: Add no events to the Undo queue.
//UndoModes.entireScript: Put a single event in the Undo queue.
//UndoModes.fastEntireScript: Put a single event in the Undo queue.
//UndoModes.scriptRequest: Undo each script action as a separate event.
//The last parameter is the text that appears in the Undo menu item.
```

Working with Script Labels

Many objects in InDesign scripting have a `label` property, including page items (rectangles, ovals, groups, polygons, text frames, and graphic lines), table cells, documents, stories, and pages. This property can store a very large amount of text.

The label of page items can be viewed, entered, or edited using the Script Label panel (choose Window > Utilities > Script Label to display this panel), shown below. You also can add a label to an object using scripting, and you can read the script label via scripting. For many objects, like stories, pages, and paragraph styles, you cannot set or view the label using the Script Label panel.

The `label` property can contain any form of text data, such as tab- or comma-delimited text, HTML, or XML. Because scripts also are text, they can be stored in the `label` property.

Page items can be referred to by their `label`, just like named items (such as paragraph styles, colors, or layers) can be referred to by their `name`. The following script fragment demonstrates this special case of the label property (for the complete script, see ScriptLabel):

```
var myDocument = app.documents.add();
var myPage = myDocument.pages.item(0);
var myX1, myX2, myY1, myY2, myRectangle;
var myPageWidth = myDocument.documentPreferences.pageWidth;
var myPageHeight = myDocument.documentPreferences.pageHeight;
//<fragment>
//Create 10 random page items.
for(var i = 0; i < 10; i++)
{
    myX1 = myGetRandom(0, myPageWidth, false);
    myY1 = myGetRandom(0, myPageHeight, false);
    myX2 = myGetRandom(0, myPageWidth, false);
    myY2 = myGetRandom(0, myPageHeight, false);
    myRectangle = myPage.rectangles.add({geometricBounds:[myY1, myX1, myY2, myX2]});
    if(myGetRandom(0, 1, true))
    {
        myRectangle.label = "myScriptLabel";
    }
}
var count = 0;
for(var i = 0; i < myPage.pageItems.length; i++)
{
    if(myPage.pageItems.item(i).label == "myScriptLabel")
    {
        count++;
    }
}
alert("Found " + count + " page items with the label.");
//This function gets a random number in the range myStart to myEnd.
function myGetRandom(myStart, myEnd, myInteger)
{
    var myRandom;
    var myRange = myEnd - myStart;
    if(myInteger == true)
    {
        myRandom = myStart = Math.round(Math.random());
    }
    else
    {
        myRandom = myStart + Math.floor(Math.random()*myRange);
    }
    return myRandom;
}
```

In addition, all objects that support the `label` property also support custom labels. A script can set a custom label using the `insertLabel` method, and extract the custom label using the `extractLabel` method, as shown in the following script fragment (from the CustomLabel tutorial script):

```
var myDocument = app.documents.add();
myDocument.viewPreferences.horizontalMeasurementUnits = MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
var myPage = myDocument.pages.item(0);
var myRectangle = myPage.rectangles.add({geometricBounds:[72, 72, 144, 144]});
//Insert a custom label using insertLabel. The first parameter is the
//name of the label, the second is the text to add to the label.
myRectangle.insertLabel("CustomLabel", "This is some text stored in a custom label.");
//Extract the text from the label and display it in an alert.
var myString = myRectangle.extractLabel("CustomLabel");
alert("Custom label contained: " + myString);
```

Running Scripts at Startup

To run a script when InDesign starts, put the script in the Startup Scripts folder in the Scripts folder (for more information, see "Installing Scripts" in *Adobe InDesign Scripting Tutorial*).

NOTE: Scripts run in the session ExtendScript engine when InDesign starts can create objects and functions that will be available to other scripts for the duration of the session. For more information, see "Session and Main Script Execution" on page 20.

Session and Main Script Execution

InDesign has two ways to run a JavaScript, `session` and `main`. These names correspond to the ExtendScript "engine" used to run the script.

By default, when you run an InDesign JavaScript, the script is interpreted and executed by the "main" ExtendScript engine, which is destroyed when the script completes execution. Script objects created by the script do not persist.

Scripts run in the session engine can create objects that persist until you close InDesign. You can refer to these objects from other scripts run in the `session` engine. To set the `session` engine as the target of an InDesign JavaScript, add the following line to the start of your script.

```
#targetengine "session"
```

You can create your own persistent ExtendScript interpretation and execution environment. To do this, use the `#targetenging` statement and provide your own ExtendScript engine name, as shown in the following script fragment:

```
#targetengine "adobe"
```

3 Documents

Chapter Update Status		
CS6	Changes	• Updated list of what this chapter shows you how to do. • Added section "Making an Adaptive Layout" on page 44. • Added section "Collecting and Dropping Content" on page 47. • Added subsection "Exporting to grayscale PDF" on page 53. • Added subsection "Exporting as a PDF form" on page 55. • In "Adjusting Page Sizes and Layout" on page 41, "Resizing and reframing pages" on page 41, and "Transforming pages" on page 42, removed outdated references to features being new in CS5.

The work you do in InDesign revolves around documents—creating them, saving them, printing or exporting them, and populating them with page items, colors, styles, and text. Almost every document-related task can be automated using InDesign scripting.

This chapter shows you how to do the following

- Perform basic document-management tasks, including:
  - Creating a new document.
  - Opening a document.
  - Saving a document.
  - Closing a document.
- Perform basic page-layout operations, including:
  - Setting the page size and document length.
  - Defining bleed and slug areas.
  - Specifying page columns and margins.
- Change the appearance of the pasteboard.
- Use guides and grids.
- Change measurement units and ruler origin.
- Define and apply document presets.
- Set up master pages (master spreads)
- Set text-formatting defaults.
- Add XMP metadata (information about a file).
- Create a document template.
- Create watermarks.

- Apply different sizes to different pages (multiple page sizes).
- Apply flexible layout formats to the same pages for use in different devices or documents with different sizes or orientation.
- Print a document.
- Export a document as Adobe PDF.
- Export pages of a document as EPS.
- Export a document as an ePub file.

We assume that you have already read *Adobe InDesign Scripting Tutorial* and know how to create, install, and run a script.

Basic Document Operations

Opening, closing, and saving documents are some of the most basic document tasks. This section shows how to do them using scripting.

Creating a new document

The following script shows how to make a new document using scripting. (For the complete script, see MakeDocument.)

```
var myDocument = app.documents.add();
```

To create a document using a document preset, the `add` method includes an optional parameter you can use to specify a document preset, as shown in the following script. (For the complete script, see MakeDocumentWithPreset.)

```
//Creates a new document using the specified document preset.
//Replace "myDocumentPreset" in the following line with the name
//of the document preset you want to use.
var myDocument = app.documents.add(true,
app.documentPresets.item("myDocumentPreset"));
```

You can create a document without displaying it in a window, as shown in the following script fragment (from the MakeDocumentWithParameters tutorial script):

```
//Creates a new document without showing the document window.
//The first parameter (showingWindow) controls the visibility of the
//document. Hidden documents are not minimized, and will not appear until
//you add a new window to the document.
var myDocument = app.documents.add(false);
//To show the window:
var myWindow = myDocument.windows.add();
```

Some script operations are much faster when the document window is hidden.

Opening a document

The following script shows how to open an existing document. (For the complete script, see OpenDocument.)

```
app.open(File("/c/myTestDocument.indd"));
```

You can choose to prevent the document from displaying (that is, hide it) by setting the `showing window` parameter of the `open` method to false (the default is true). You might want to do this to improve performance of a script. To show a hidden document, create a new window, as shown in the following script fragment (from the OpenDocumentInBackground tutorial script):

```
//Opens an existing document in the background, then shows the document.
//You'll have to fill in your own file path.
var myDocument = app.open(File("/c/myTestDocument.indd"), false);
//At this point, you could do things with the document without showing the
//document window. In some cases, scripts will run faster when the document
//window is not visible.
//When you want to show the hidden document, create a new window.
var myLayoutWindow = myDocument.windows.add();
```

Saving a document

In the InDesign user interface, you save a file by choosing File > Save, and you save a file to another file name by choosing File > Save As. In InDesign scripting, the `save` method can do either operation, as shown in the following script fragment (from the SaveDocument tutorial script):

```
//If the active document has been changed since it was last saved, save it.
if(app.activeDocument.modified == true){
    app.activeDocument.save();
}
```

The `save` method has two optional parameters: The first (`to`) specifies the file to save to; the second (`stationery`) can be set to true to save the document as a template, as shown in the following script fragment (from the SaveDocumentAs tutorial script):

```
//If the active document has not been saved (ever), save it.
if(app.activeDocument.saved == false){
    //If you do not provide a file name, InDesign displays the Save dialog box.
    app.activeDocument.save(new File("/c/myTestDocument.indd"));
}
```

You can save a document as a template, as shown in the following script fragment (from the SaveAsTemplate tutorial script):

```
//Save the active document as a template.
var myFileName;
if(app.activeDocument.saved == true){
    //Convert the file name to a string.
    myFileName = app.activeDocument.fullName + "";
    //If the file name contains the extension ".indd", change it to ".indt".
    if(myFileName.indexOf(".indd")!=-1){
        var myRegularExpression = /.indd/gi
        myFileName = myFileName.replace(myRegularExpression, ".indt");
    }
}
//If the document has not been saved, then give it a default file name/file path.
else{
    myFileName = "/c/myTestDocument.indt";
}

app.activeDocument.save(File(myFileName), true);
```

Closing a document

The `close` method closes a document, as shown in the following script fragment (from the CloseDocument tutorial script):

```
app.activeDocument.close();
//Note that you could also use:
//app.documents.item(0).close();
```

The `close` method can take up to two optional parameters, as shown in the following script fragment (from the CloseWithParameters tutorial script):

```
//Use SaveOptions.yes to save the document,SaveOptions.no to close the
//document without saving, or SaveOptions.ask to display a prompt. If
//you use SaveOptions.yes, you'll need to provide a reference to a file
//to save to in the second parameter (SavingIn).
//Note that the file path is provided using the JavaScript URI form
//rather than the platform-specific form.
//
//If the file has not been saved, display a prompt.
if(app.activeDocument.saved != true){
    app.activeDocument.close(SaveOptions.ask);
    //Or, to save to a specific file name:
    //var myFile = File("/c/myTestDocument.indd");
    //app.activeDocument.close(SaveOptions.yes, myFile);
}
else{
    //If the file has already been saved, save it.
    app.activeDocument.close(SaveOptions.yes);
}
```

You can close all open documents without saving them, as shown in the following script fragment (from the CloseAll tutorial script):

```
for(myCounter = app.documents.length; myCounter > 0; myCounter--){
    app.documents.item(myCounter-1).close(SaveOptions.no);
}
```

Basic Page Layout

Each document has a default page size, assigned number of pages, bleed and slug working areas, and columns and margins to define the area into which material is placed. Again, all these parameters are accessible to scripting, as shown in the examples in this section.

Defining page size and document length

When you create a new document using the InDesign user interface, you can specify the default page size, number of pages, page orientation, and whether the document uses facing pages. To create a document using InDesign scripting, use the `documents.add` method, which does not specify these settings. After creating a document, you can use the `documentPreferences` object to control the settings, as shown in the following script fragment (from the DocumentPreferences tutorial script):

```
var myDocument = app.documents.add();
with(myDocument.documentPreferences){
    pageHeight = "800pt";
    pageWidth = "600pt";
    pageOrientation = PageOrientation.landscape;
    pagesPerDocument = 16;
}
```

NOTE: The `app` object also has a `documentPreferences` object. You can set the application defaults for page height, page width, and other properties by changing the properties of this object. You can also set individual page sizes; see "Adjusting Page Sizes and Layout".

Defining bleed and slug areas

Within InDesign, a *bleed* or a *slug* is an area outside the page margins that can be printed or included in an exported PDF. Typically, these areas are used for objects that extend beyond the page edges (bleed) and job/document information (slug). The two areas can be printed and exported independently; for example, you might want to omit slug information for the final printing of a document. The following script shows how to set up the bleed and slug for a new document. (For the complete script, see BleedAndSlug.)

```
myDocument = app.documents.add();
//The bleed and slug properties belong to the documentPreferences object.
with(myDocument.documentPreferences){
    //Bleed
    documentBleedBottomOffset = "3p";
    documentBleedTopOffset = "3p";
    documentBleedInsideOrLeftOffset = "3p";
    documentBleedOutsideOrRightOffset = "3p";
    //Slug
    slugBottomOffset = "18p";
    slugTopOffset = "3p";
    slugInsideOrLeftOffset = "3p";
    slugRightOrOutsideOffset = "3p";
}
```

Alternately, if all the bleed distances are equal, as in the preceding example, you can use the `documentBleedUniformSize` property, as shown in the following script fragment (from the UniformBleed tutorial script):

```
//Create a new document.
myDocument = app.documents.add();
//The bleed properties belong to the documentPreferences object.
with(myDocument.documentPreferences){
    //Bleed
    documentBleedUniformSize = true;
    documentBleedTopOffset = "3p";
}
```

If all the slug distances are equal, you can use the `documentSlugUniformSize` property, as shown in the following script fragment (from the UniformSlug tutorial script):

```
//Create a new document.
myDocument = app.documents.add();
//The slug properties belong to the documentPreferences object.
with(myDocument.documentPreferences){
    //Slug:
    documentSlugUniformSize = true;
    slugTopOffset = "3p";
}
```

In addition to setting the bleed and slug widths and heights, you can control the color used to draw the guides defining the bleed and slug. This property is not in the `documentPreferences` object; instead, it is in the `pasteboardPreferences` object, as shown in the following script fragment (from the BleedSlugGuideColors tutorial script):

```
with(app.activeDocument.pasteboardPreferences){
    //Any of InDesign's guides can use the UIColors constants...
    bleedGuideColor = UIColors.cuteTeal;
    slugGuideColor = UIColors.charcoal;
    //...or you can specify an array of RGB values (with values from 0 to 255)
    //bleedGuideColor = [0, 198, 192];
    //slugGuideColor = [192, 192, 192];
}
```

Setting page margins and columns

Each page in a document can have its own margin and column settings. With InDesign scripting, these properties are part of the `marginPreferences` object for each page. This following sample script creates a new document, then sets the margins and columns for all pages in the master spread. (For the complete script, see PageMargins.)

```
myDocument = app.documents.add();
with (myDocument.pages.item(0).marginPreferences){
    columnCount = 3;
    //columnGutter can be a number or a measurement string.
    columnGutter = "1p";
    bottom = "6p"
    //When document.documentPreferences.facingPages == true,
    //"left" means inside; "right" means outside.
    left = "6p"
    right = "4p"
    top = "4p"
}
```

To set the page margins for an individual page, use the margin preferences for that page, as shown in the following script fragment (from the PageMarginsForOnePage tutorial script):

```
myDocument = app.documents.add();
with (myDocument.pages.item(0).marginPreferences){
    columnCount = 3;
    //columnGutter can be a number or a measurement string.
    columnGutter = "1p";
    bottom = "6p"
    //When document.documentPreferences.facingPages == true,
    //"left" means inside; "right" means outside.
    left = "6p"
    right = "4p"
    top = "4p"
}
```

InDesign does not allow you to create a page that is smaller than the sum of the relevant margins; that is, the width of the page must be greater than the sum of the left and right page margins, and the height of the page must be greater than the sum of the top and bottom margins. If you are creating very small pages (for example, for individual newspaper advertisements) using the InDesign user interface, you can easily set the correct margin sizes as you create the document, by entering new values in the document default page Margin fields in the New Document dialog box.

From scripting, however, the solution is not as clear: when you create a document, it uses the *application's* default-margin preferences. These margins are applied to all pages of the document, including master pages. Setting the document margin preferences affects only new pages and has no effect on existing pages. If you try to set the page height and page width to values smaller than the sum of the corresponding margins on any existing pages, InDesign does not change the page size.

There are two solutions. The first is to set the margins of the existing pages before you try to change the page size, as shown in the following script fragment (from the PageMarginsForSmallPages tutorial script):

```
var myDocument = app.documents.add();
myDocument.marginPreferences.top = 0;
myDocument.marginPreferences.left = 0;
myDocument.marginPreferences.bottom = 0;
myDocument.marginPreferences.right = 0;
//The following assumes that your default document contains a single page.
myDocument.pages.item(0).marginPreferences.top = 0;
myDocument.pages.item(0).marginPreferences.left = 0;
myDocument.pages.item(0).marginPreferences.bottom = 0;
myDocument.pages.item(0).marginPreferences.right = 0;
//The following assumes that your default master spread contains two pages.
myDocument.masterSpreads.item(0).pages.item(0).marginPreferences.top = 0;
myDocument.masterSpreads.item(0).pages.item(0).marginPreferences.left = 0;
myDocument.masterSpreads.item(0).pages.item(0).marginPreferences.bottom = 0;
myDocument.masterSpreads.item(0).pages.item(0).marginPreferences.right = 0;
myDocument.masterSpreads.item(0).pages.item(1).marginPreferences.top = 0;
myDocument.masterSpreads.item(0).pages.item(1).marginPreferences.left = 0;
myDocument.masterSpreads.item(0).pages.item(1).marginPreferences.bottom = 0;
myDocument.masterSpreads.item(0).pages.item(1).marginPreferences.right = 0;
myDocument.documentPreferences.pageHeight = "1p";
myDocument.documentPreferences.pageWidth = "6p";
```

Alternately, you can change the application's default-margin preferences before you create the document, as shown in the following script fragment (from the ApplicationPageMargins tutorial script):

```
with (app.marginPreferences){
    //Save the current application default margin preferences.
    var myY1 = top;
    var myX1 = left;
    var myY2 = bottom;
    var myX2 = right;
    //Set the application default margin preferences.
    top = 0;
    left = 0;
    bottom = 0;
    right = 0;
}
//Create a new example document to demonstrate the change.
var myDocument = app.documents.add();
myDocument.documentPreferences.pageHeight = "1p";
myDocument.documentPreferences.pageWidth = "6p";
//Reset the application default margin preferences to their former state.
with (app.marginPreferences){
    top = myY1;
    left = myX1 ;
    bottom = myY2;
    right = myX2;
}
```

Changing the appearance of the pasteboard

The *pasteboard* is the area that surrounds InDesign pages and spreads. You can use it for temporary storage of page items or for job-tracking information. You can change the size of the pasteboard and its color using scripting. The `previewBackgroundColor` property sets the color of the pasteboard in Preview mode, as shown in the following script fragment (from the PasteboardPreferences tutorial script):

```
myDocument = app.documents.add();
with(myDocument.pasteboardPreferences){
    //You can use either a number or a measurement string
    //to set the space above/below.
    minimumSpaceAboveAndBelow = "12p";
    //You can set the preview background color to any of
    //the predefined UIColor enumerations...
    previewBackgroundColor = UIColors.gray;
    //...or you can specify an array of RGB values
    //(with values from 0 to 255)
    //previewBackgroundColor = [192, 192, 192];
}
```

Guides and grids

Guides and grids make it easy to position objects on your document pages. These are very useful items to add when you are creating templates for others to use.

Defining guides

Guides in InDesign give you an easy way to position objects on the pages of your document. The following script fragment shows how to use guides. (For the complete script, see Guides.)

```
var myDocument = app.documents.add();
var myPageWidth = myDocument.documentPreferences.pageWidth;
var myPageHeight = myDocument.documentPreferences.pageHeight;
with(myDocument.pages.item(0)){
    //Place guides at the margins of the page.
    guides.add(undefined, {orientation:HorizontalOrVertical.vertical, <lb>
    location:marginPreferences.left});
    guides.add(undefined, {orientation:HorizontalOrVertical.vertical,  <lb>
    location:(myPageWidth - marginPreferences.right)});
    guides.add(undefined, {orientation:HorizontalOrVertical.horizontal,  <lb>
    location:marginPreferences.top});
    guides.add(undefined, {orientation:HorizontalOrVertical.horizontal,  <lb>
    location:(myPageHeight - marginPreferences.bottom)});
    //Place a guide at the vertical center of the page.
    guides.add(undefined, {orientation:HorizontalOrVertical.vertical,  <lb>
    location:(myPageWidth/2)});
    //Place a guide at the horizontal center of the page.
    guides.add(undefined, {orientation:HorizontalOrVertical.horizontal,  <lb>
    location:(myPageHeight/2)});
}
```

Horizontal guides can be limited to a given page or extend across all pages in a spread. From InDesign scripting, you can control this using the `fitToPage` property. This property is ignored by vertical guides.

You can use scripting to change the layer, color, and visibility of guides, just as you can from the user interface, as shown in the following script fragment (from the GuideOptions tutorial script):

```
var myDocument = app.documents.add();
var myPageWidth = myDocument.documentPreferences.pageWidth;
var myPageHeight = myDocument.documentPreferences.pageHeight;
with(myDocument.pages.item(0)){
    //Place guides at the margins of the page.
    guides.add(undefined, {orientation:HorizontalOrVertical.vertical, <lb>
    location:marginPreferences.left});
    guides.add(undefined, {orientation:HorizontalOrVertical.vertical,  <lb>
    location:(myPageWidth - marginPreferences.right)});
    guides.add(undefined, {orientation:HorizontalOrVertical.horizontal,  <lb>
    location:marginPreferences.top});
    guides.add(undefined, {orientation:HorizontalOrVertical.horizontal,  <lb>
    location:(myPageHeight - marginPreferences.bottom)});
    //Place a guide at the vertical center of the page.
    guides.add(undefined, {orientation:HorizontalOrVertical.vertical,  <lb>
    location:(myPageWidth/2)});
    //Place a guide at the horizontal center of the page.
    guides.add(undefined, {orientation:HorizontalOrVertical.horizontal,  <lb>
    location:(myPageHeight/2)});
}
```

You also can create guides using the `createGuides` method on spreads and master spreads, as shown in the following script fragment (from the CreateGuides tutorial script):

```
var myDocument = app.documents.add();
with (myDocument.spreads.item(0)){
    //Parameters (all optional): row count, column count, row gutter,
    //column gutter,guide color, fit margins, remove existing, layer.
    //Note that the createGuides method does not take an RGB array
    //for the guide color parameter.
    createGuides(4, 4, "1p", "1p", UIColors.gray, true, true,
myDocument.layers.item(0));
}
```

Setting grid preferences

To control the properties of the document and baseline grid, you set the properties of the `gridPreferences` object, as shown in the following script fragment (from the DocumentAndBaselineGrid tutorial script):

```
var myDocument = app.documents.add();
//Set the document measurement units to points.
myDocument.viewPreferences.horizontalMeasurementUnits = MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
//Set up grid preferences.
with(myDocument.gridPreferences){
    baselineStart = 56;
    baselineDivision = 14;
    baselineShown = true;
    horizontalGridlineDivision = 14;
    horizontalGridSubdivision = 5
    verticalGridlineDivision = 14;
    verticalGridSubdivision = 5
    documentGridShown = true;
}
```

Snapping to guides and grids

All snap settings for a document's grids and guides are in the properties of the `guidePreferences` and `gridPreferences` objects. The following script fragment shows how to set guide and grid snap properties. (For the complete script, see GuideGridPreferences.)

```
var myDocument = app.activeDocument;
with(myDocument.guidePreferences){
    guidesInBack = true;
    guidesLocked = false;
    guidesShown = true;
    guidesSnapTo = true;
}
with(myDocument.gridPreferences){
    documentGridShown = false;
    documentGridSnapTo = true;
    //Objects "snap" to the baseline grid when
    //guidePreferences.guideSnapTo is set to true.
    baselineGridShown = true;
}
```

Changing measurement units and ruler

Thus far, the sample scripts used *measurement strings*, strings that force InDesign to use a specific measurement unit (for example, "8.5i" for 8.5 inches). They do this because you might be using a different measurement system when you run the script.

To specify the measurement system used in a script, use the document's `viewPreferences` object, as shown in the following script fragment (from the ViewPreferences tutorial script):

```
var myDocument = app.activeDocument;
with(myDocument.viewPreferences){
    //Measurement unit choices are:
    //* MeasurementUnits.agates
    //* MeasurementUnits.picas
    //* MeasurementUnits.points
    //* MeasurementUnits.inches
    //* MeasurementUnits.inchesDecimal
    //* MeasurementUnits.millimeters
    //* MeasurementUnits.centimeters
    //* MeasurementUnits.ciceros
    //Set horizontal and vertical measurement units to points.
    horizontalMeasurementUnits = MeasurementUnits.points;
    verticalMeasurementUnits = MeasurementUnits.points;
}
```

If you are writing a script that needs to use a specific measurement system, you can change the measurement units at the beginning of the script, then restore the original measurement units at the end of the script. This is shown in the following script fragment (from the ResetMeasurementUnits tutorial script):

```
var myDocument = app.activeDocument
with (myDocument.viewPreferences){
    var myOldXUnits = horizontalMeasurementUnits;
    var myOldYUnits = verticalMeasurementUnits;
    horizontalMeasurementUnits = MeasurementUnits.points;
    verticalMeasurementUnits = MeasurementUnits.points;
}
//At this point, you can perform any series of script actions
//that depend on the measurement units you've set. At the end of
//the script, reset the measurement units to their original state.
with (myDocument.viewPreferences){
    try{
        horizontalMeasurementUnits = myOldXUnits;
        verticalMeasurementUnits = myOldYUnits;
    }
    catch(myError){
        alert("Could not reset custom measurement units.");
    }
}
```

Defining and applying document presets

InDesign document presets enable you to store and apply common document set-up information (page size, page margins, columns, and bleed and slug areas). When you create a new document, you can base the document on a document preset.

Creating a preset by copying values

To create a document preset using an existing document's settings as an example, open a document that has the document set-up properties you want to use in the document preset, then run the following script (from the DocumentPresetByExample tutorial script):

```
var myDocumentPreset;
if(app.documents.length > 0){
    var myDocument = app.activeDocument;
    //If the document preset "myDocumentPreset" does not already
    //exist, create it.
    myDocumentPreset = app.documentPresets.item("myDocumentPreset");
    try {
        var myPresetName = myDocumentPreset.name;
    }
    catch (myError){
        myDocumentPreset = app.documentPresets.add({name:"myDocumentPreset"});
    }
    //Set the application default measurement units to match the document
    //measurement units.
    app.viewPreferences.horizontalMeasurementUnits =
    myDocument.viewPreferences.horizontalMeasurementUnits;
    app.viewPreferences.verticalMeasurementUnits =
    myDocument.viewPreferences.verticalMeasurementUnits;
    //Fill in the properties of the document preset with the corresponding
    //properties of the active document.
    with(myDocumentPreset){
        //Note that the following gets the page margins
        //from the margin preferences of the document; to get the margin
        //preferences from the active page,replace "app.activeDocument" with
        //"app.activeWindow.activePage" in the following line (assuming the
        //active window is a layout window).
        var myMarginPreferences = app.activeDocument.marginPreferences;
        left = myMarginPreferences.left;
        right = myMarginPreferences.right;
        top = myMarginPreferences.top;
        bottom = myMarginPreferences.bottom;
        columnCount = myMarginPreferences.columnCount;
        columnGutter = myMarginPreferences.columnGutter;
        documentBleedBottom =
        app.activeDocument.documentPreferences.documentBleedBottomOffset;
        documentBleedTop =
        app.activeDocument.documentPreferences.documentBleedTopOffset;
        documentBleedLeft =
        app.activeDocument.documentPreferences.documentBleedInsideOrLeftOffset;
        documentBleedRight = app.activeDocument.documentPreferences.
        documentBleedOutsideOrRightOffset;
        facingPages = app.activeDocument.documentPreferences.facingPages;
        pageHeight = app.activeDocument.documentPreferences.pageHeight;
        pageWidth = app.activeDocument.documentPreferences.pageWidth;
        pageOrientation =
        app.activeDocument.documentPreferences.pageOrientation;
        pagesPerDocument =
        app.activeDocument.documentPreferences.pagesPerDocument;
        slugBottomOffset =
        app.activeDocument.documentPreferences.slugBottomOffset;
        slugTopOffset = app.activeDocument.documentPreferences.slugTopOffset;
        slugInsideOrLeftOffset =
        app.activeDocument.documentPreferences.slugInsideOrLeftOffset;
        slugRightOrOutsideOffset =
        app.activeDocument.documentPreferences.slugRightOrOutsideOffset;
    }
}
```

Creating a document preset

To create a document preset using explicit values, run the following script (from the DocumentPreset tutorial script):

```
var myDocumentPreset;
//If the document preset "myDocumentPreset" does not already exist, create it.
myDocumentPreset = app.documentPresets.item("myDocumentPreset");
try {
    var myPresetName = myDocumentPreset.name;
}
catch (myError){
    myDocumentPreset = app.documentPresets.add({name:"myDocumentPreset"});
}
//Fill in the properties of the document preset.
with(myDocumentPreset){
    pageHeight = "9i";
    pageWidth = "7i";
    left = "4p";
    right = "6p";
    top = "4p";
    bottom = "9p";
    columnCount = 1;
    documentBleedBottom = "3p";
    documentBleedTop = "3p";
    documentBleedLeft = "3p";
    documentBleedRight = "3p";
    facingPages = true;
    pageOrientation = PageOrientation.portrait;
    pagesPerDocument = 1;
    slugBottomOffset = "18p";
    slugTopOffset = "3p";
    slugInsideOrLeftOffset = "3p";
    slugRightOrOutsideOffset = "3p";
}
```

Setting up master spreads

After setting up the basic document page size, slug, and bleed, you probably will want to define the document's master spreads. The following script shows how to do that. (For the complete script, see MasterSpread.)

```
myDocument = app.documents.add();
//Set up the document.
with(myDocument.documentPreferences){
    pageHeight = "11i"
    pageWidth = "8.5i"
    facingPages = true;
    pageOrientation = PageOrientation.portrait;
}
//Set the document's ruler origin to page origin. This is very important
//--if you don't do this, getting objects to the correct position on the
//page is much more difficult.
myDocument.viewPreferences.rulerOrigin = RulerOrigin.pageOrigin;
with(myDocument.masterSpreads.item(0)){
    //Set up the left page (verso).
    with(pages.item(0)){
        with(marginPreferences){
```

```
            columnCount = 3;
            columnGutter = "1p";
            bottom = "6p"
            //"left" means inside; "right" means outside.
            left = "6p"
            right = "4p"
            top = "4p"
        }
        //Add a simple footer with a section number and page number.
        with(textFrames.add()){
            geometricBounds = ["61p", "4p", "62p", "45p"];
            insertionPoints.item(0).contents = SpecialCharacters.sectionMarker;
            insertionPoints.item(0).contents = SpecialCharacters.emSpace;
            insertionPoints.item(0).contents = SpecialCharacters.autoPageNumber;
            paragraphs.item(0).justification = Justification.leftAlign;
        }
    }
    //Set up the right page (recto).
    with(pages.item(1)){
        with(marginPreferences){
            columnCount = 3;
            columnGutter = "1p";
            bottom = "6p"
            //"left" means inside; "right" means outside.
            left = "6p"
            right = "4p"
            top = "4p"
        }
        //Add a simple footer with a section number and page number.
        with(textFrames.add()){
            geometricBounds = ["61p", "6p", "62p", "47p"];
            insertionPoints.item(0).contents = SpecialCharacters.autoPageNumber;
            insertionPoints.item(0).contents = SpecialCharacters.emSpace;
            insertionPoints.item(0).contents = SpecialCharacters.sectionMarker;
            paragraphs.item(0).justification = Justification.rightAlign;
        }
    }
}
```

To apply a master spread to a document page, use the `appliedMaster` property of the document page, as shown in the following script fragment (from the ApplyMaster tutorial script):

```
//Assumes that the active document has a master page named "B-Master"
//and at least three pages--page 3 is pages.item(2) because JavaScript arrays are
zero-based.
app.activeDocument.pages.item(2).appliedMaster =
app.activeDocument.masterSpreads.item("B-Master");
```

Use the same property to apply a master spread to a master spread page, as shown in the following script fragment (from the ApplyMasterToMaster tutorial script):

```
//Assumes that the active document has master spread named "B-Master"
//that is not the same as the first master spread in the document.
app.activeDocument.masterSpreads.item(0).pages.item(0).appliedMaster =
app.activeDocument.masterSpreads.item("B-Master");
```

Adding XMP metadata

Metadata is information that describes the content, origin, or other attributes of a file. In the InDesign user interface, you enter, edit, and view metadata using the File Info dialog (choose File > File Info). This metadata includes the document's creation and modification dates, author, copyright status, and other information. All this information is stored using XMP (Adobe Extensible Metadata Platform), an open standard for embedding metadata in a document.

To learn more about XMP, see the XMP specification at http://partners.adobe.com/asn/developer/pdf/MetadataFramework.pdf.

You also can add XMP information to a document using InDesign scripting. All XMP properties for a document are in the document's `metadataPreferences` object. The example below fills in the standard XMP data for a document.

This example also shows that XMP information is extensible. If you need to attach metadata to a document and the data does not fall into a category provided by the metadata preferences object, you can create your own metadata container (`email`, in this example). (For the complete script, see MetadataExample.)

```
var myDocument = app.documents.add();
with (myDocument.metadataPreferences){
    author = "Adobe";
    copyrightInfoURL = "http://www.adobe.com";
    copyrightNotice = "This document is copyrighted.";
    copyrightStatus = CopyrightStatus.yes;
    description = "Example of xmp metadata scripting in InDesign CS";
    documentTitle = "XMP Example";
    jobName = "XMP_Example_2003";
    keywords = ["animal", "mineral", "vegetable"];
    //The metadata preferences object also includes the read-only
    //creator, format, creationDate, modificationDate, and serverURL
    //properties that are automatically entered and maintained by InDesign.
    //Create a custom XMP container, "email"
    var myNewContainer = createContainerItem("http://ns.adobe.com/xap/1.0/", "email");
    setProperty("http://ns.adobe.com/xap/1.0/", "email/*[1]", "someone@adobe.com");
}
```

Creating a document template

This example creates a new document, defines slug and bleed areas, adds information to the document's XMP metadata, sets up master pages, adds page footers, and adds job information to a table in the slug area. (For the complete script, see DocumentTemplate.)

```
//Set the application default margin preferences.
with (app.marginPreferences){
    //Save the current application default margin preferences.
    var myY1 = top;
    var myX1 = left;
    var myY2 = bottom;
    var myX2 = right;
    //Set the application default margin preferences.
    //Document baseline grid will be based on 14 points, and
    //all margins are set in increments of 14 points.
    top = 14 * 4 + "pt";
    left = 14 * 4 + "pt";
    bottom = "74pt";
    right = 14 * 5 + "pt";
}
//Make a new document.
var myDocument = app.documents.add();
myDocument.documentPreferences.pageWidth = "7i";
myDocument.documentPreferences.pageHeight = "9i";
myDocument.documentPreferences.pageOrientation = PageOrientation.portrait;
//At this point, we can reset the application default margins to their original state.
with (app.marginPreferences){
    top = myY1;
    left = myX1;
    bottom = myY2;
    right = myX2;
}
//Set up the bleed and slug areas.
with(myDocument.documentPreferences){
    //Bleed
    documentBleedBottomOffset = "3p";
    documentBleedTopOffset = "3p";
    documentBleedInsideOrLeftOffset = "3p";
    documentBleedOutsideOrRightOffset = "3p";
    //Slug
    slugBottomOffset = "18p";
    slugTopOffset = "3p";
    slugInsideOrLeftOffset = "3p";
    slugRightOrOutsideOffset = "3p";
}
//Create a color.
try{
    myDocument.colors.item("PageNumberRed").name;
}
catch (myError){
    myDocument.colors.add({name:"PageNumberRed", model:ColorModel.process,
colorValue:[20, 100, 80, 10]});
}
//Next, set up some default styles.
//Create up a character style for the page numbers.
try{
    myDocument.characterStyles.item("page_number").name;
}
catch (myError){
    myDocument.characterStyles.add({name:"page_number"});
}
myDocument.characterStyles.item("page_number").fillColor =
myDocument.colors.item("PageNumberRed");
//Create up a pair of paragraph styles for the page footer text.
//These styles have only basic formatting.
```

```
try{
    myDocument.paragraphStyles.item("footer_left").name;
}
catch (myError){
    myDocument.paragraphStyles.add({name:"footer_left", pointSize:11, leading:14});
}
//Create up a pair of paragraph styles for the page footer text.
try{
    myDocument.paragraphStyles.item("footer_right").name;
}
catch (myError){
    myDocument.paragraphStyles.add({name:"footer_right",
basedOn:myDocument.paragraphStyles.item("footer_left"),
justification:Justification.rightAlign, pointSize:11, leading:14});
}
//Create a layer for guides.
try{
    myDocument.layers.item("GuideLayer").name;
}
catch (myError){
    myDocument.layers.add({name:"GuideLayer"});
}
//Create a layer for the footer items.
try{
    myDocument.layers.item("Footer").name;
}
catch (myError){
    myDocument.layers.add({name:"Footer"});
}
//Create a layer for the slug items.
try{
    myDocument.layers.item("Slug").name;
}
catch (myError){
    myDocument.layers.add({name:"Slug"});
}
//Create a layer for the body text.
try{
    myDocument.layers.item("BodyText").name;
}
catch (myError){
    myDocument.layers.add({name:"BodyText"});
}
with(myDocument.viewPreferences){
    rulerOrigin = RulerOrigin.pageOrigin;
    horizontalMeasurementUnits = MeasurementUnits.points;
    verticalMeasurementUnits = MeasurementUnits.points;
}
//Document baseline grid and document grid
with(myDocument.gridPreferences){
    baselineStart = 56;
    baselineDivision = 14;
    baselineShown = false;
    horizontalGridlineDivision = 14;
    horizontalGridSubdivision = 5
    verticalGridlineDivision = 14;
    verticalGridSubdivision = 5
    documentGridShown = false;
}
```

```
//Document XMP information.
with (myDocument.metadataPreferences){
    author = "Olav Martin Kvern";
    copyrightInfoURL = "http://www.adobe.com";
    copyrightNotice = "This document is not copyrighted.";
    copyrightStatus = CopyrightStatus.no;
    description = "Example 7 x 9 book layout";
    documentTitle = "Example";
    jobName = "7 x 9 book layout template";
    keywords = ["7 x 9", "book", "template"];
    var myNewContainer = createContainerItem("http://ns.adobe.com/xap/1.0/", "email");
    setProperty("http://ns.adobe.com/xap/1.0/", "email/*[1]", "okvern@adobe.com");
}
//Set up the master spread.
with(myDocument.masterSpreads.item(0)){
    with(pages.item(0)){
        //Left and right are reversed for left-hand pages (becoming "inside" and
"outside"--
        //this is also true in the InDesign user interface).
        var myBottomMargin = myDocument.documentPreferences.pageHeight -
marginPreferences.bottom;
        var myRightMargin = myDocument.documentPreferences.pageWidth -
marginPreferences.left;
        guides.add(myDocument.layers.item("GuideLayer"),
{orientation:HorizontalOrVertical.vertical,location:marginPreferences.right});
        guides.add(myDocument.layers.item("GuideLayer"),
{orientation:HorizontalOrVertical.vertical, location:myRightMargin});
        guides.add(myDocument.layers.item("GuideLayer"),
{orientation:HorizontalOrVertical.horizontal, location:marginPreferences.top,
fitToPage:false});
        guides.add(myDocument.layers.item("GuideLayer"),
{orientation:HorizontalOrVertical.horizontal, location:myBottomMargin,
fitToPage:false});
        guides.add(myDocument.layers.item("GuideLayer"),
{orientation:HorizontalOrVertical.horizontal, location:myBottomMargin + 14,
fitToPage:false});
        guides.add(myDocument.layers.item("GuideLayer"),
{orientation:HorizontalOrVertical.horizontal, location:myBottomMargin + 28,
fitToPage:false});
        var myLeftFooter = textFrames.add(myDocument.layers.item("Footer"), undefined,
undefined, {geometricBounds:[myBottomMargin+14, marginPreferences.right,
myBottomMargin+28, myRightMargin]})
        myLeftFooter.parentStory.insertionPoints.item(0).contents =
SpecialCharacters.sectionMarker;
        myLeftFooter.parentStory.insertionPoints.item(0).contents =
SpecialCharacters.emSpace;
        myLeftFooter.parentStory.insertionPoints.item(0).contents =
SpecialCharacters.autoPageNumber;
        myLeftFooter.parentStory.characters.item(0).appliedCharacterStyle =
myDocument.characterStyles.item("page_number");

myLeftFooter.parentStory.paragraphs.item(0).applyStyle(myDocument.paragraphStyles.ite
m("footer_left", false));
        //Slug information.
        with(myDocument.metadataPreferences){
            var myString = "Author:\t" + author + "\tDescription:\t" + description +
"\rCreation Date:\t" + new Date +
            "\tEmail Contact\t" + getProperty("http://ns.adobe.com/xap/1.0/",
"email/*[1]");
        }
```

```
        var myLeftSlug = textFrames.add(myDocument.layers.item("Slug"), undefined,
undefined, {geometricBounds:[myDocument.documentPreferences.pageHeight+36,
marginPreferences.right, myDocument.documentPreferences.pageHeight + 144,
myRightMargin], contents:myString});
        myLeftSlug.parentStory.tables.add();
        //Body text master text frame.
        var myLeftFrame = textFrames.add(myDocument.layers.item("BodyText"), undefined,
undefined, {geometricBounds:[marginPreferences.top, marginPreferences.right,
myBottomMargin, myRightMargin]});
    }
    with(pages.item(1)){
        var myBottomMargin = myDocument.documentPreferences.pageHeight -
marginPreferences.bottom;
        var myRightMargin = myDocument.documentPreferences.pageWidth -
marginPreferences.right;
        guides.add(myDocument.layers.item("GuideLayer"),
{orientation:HorizontalOrVertical.vertical,location:marginPreferences.left});
        guides.add(myDocument.layers.item("GuideLayer"),
{orientation:HorizontalOrVertical.vertical, location:myRightMargin});
        var myRightFooter = textFrames.add(myDocument.layers.item("Footer"), undefined,
undefined, {geometricBounds:[myBottomMargin+14, marginPreferences.left,
myBottomMargin+28, myRightMargin]})
        myRightFooter.parentStory.insertionPoints.item(0).contents =
SpecialCharacters.autoPageNumber;
        myRightFooter.parentStory.insertionPoints.item(0).contents =
SpecialCharacters.emSpace;
        myRightFooter.parentStory.insertionPoints.item(0).contents =
SpecialCharacters.sectionMarker;
        myRightFooter.parentStory.characters.item(-1).appliedCharacterStyle =
myDocument.characterStyles.item("page_number");

myRightFooter.parentStory.paragraphs.item(0).applyStyle(myDocument.paragraphStyles.it
em("footer_right", false));
        //Slug information.
        var myRightSlug = textFrames.add(myDocument.layers.item("Slug"), undefined,
undefined, {geometricBounds:[myDocument.documentPreferences.pageHeight+36,
marginPreferences.left, myDocument.documentPreferences.pageHeight + 144,
myRightMargin], contents:myString});
        myRightSlug.parentStory.tables.add();
        //Body text master text frame.
        var myRightFrame = textFrames.add(myDocument.layers.item("BodyText"),
undefined, undefined, {geometricBounds:[marginPreferences.top, marginPreferences.left,
myBottomMargin, myRightMargin], previousTextFrame:myLeftFrame});
    }
}
//Add section marker text--this text will appear in the footer.
myDocument.sections.item(0).marker = "Section 1";
//When you link the master page text frames, one of the frames sometimes becomes
selected. Deselect it.
app.select(NothingEnum.nothing, undefined);
```

Creating watermarks

You can apply watermarks to documents in InDesign or InDesign Server using scripting. Currently, no user interface component exists in InDesign for managing watermarks.

A document's watermark preferences can be set in two ways using scripting:

- Application-level watermark preferences, if any are set, are applied to the document watermark preferences for each new document created by InDesign. This setting has no effect on existing documents.
- Document-level watermark preferences apply only to that document. Setting or changing a document's watermark preferences replaces any previous watermark settings for the document.

Both the document and application watermark preference settings persist after the document or application is closed until a script changes them.

The same group of watermark preferences exist for both the document and the application objects.

Setting watermark preferences

The following script fragment shows how to set watermarks at the application level. A watermark will be applied to all documents created after this code finishes. (For the complete script for setting application preferences, see ApplicationWatermark.)

```
app.watermarkPreferences.watermarkVisibility = true;
app.watermarkPreferences.watermarkDoPrint = true;
app.watermarkPreferences.watermarkDrawInBack = true;
app.watermarkPreferences.watermarkText = "Confidential";
app.watermarkPreferences.watermarkFontFamily = "Arial";
app.watermarkPreferences.watermarkFontStyle = "Bold";
app.watermarkPreferences.watermarkFontPointSize = 72;
app.watermarkPreferences.watermarkFontColor = UIColors.red;
app.watermarkPreferences.watermarkOpacity = 60;
app.watermarkPreferences.watermarkRotation = -45;
app.watermarkPreferences.watermarkHorizontalPosition =
    WatermarkHorizontalPositionEnum.watermarkHCenter;
app.watermarkPreferences.watermarkHorizontalOffset = 0;
app.watermarkPreferences.watermarkVerticalPosition =
    WatermarkVerticalPositionEnum.watermarkVCenter;
app.watermarkPreferences.watermarkVerticalOffset = 0;
```

The same preferences can be applied to a document object by referring to a document, rather than to the application. (For the complete script for setting document preferences, see DocumentWatermark.)

```
var myDocument = app.documents.item(0);
myDocument.watermarkPreferences.watermarkVisibility = true;
myDocument.watermarkPreferences.watermarkDoPrint = true;
myDocument.watermarkPreferences.watermarkDrawInBack = true;
myDocument.watermarkPreferences.watermarkText = "Confidential";
myDocument.watermarkPreferences.watermarkFontFamily = "Arial";
myDocument.watermarkPreferences.watermarkFontStyle = "Bold";
myDocument.watermarkPreferences.watermarkFontPointSize = 72;
myDocument.watermarkPreferences.watermarkFontColor = UIColors.blue;
myDocument.watermarkPreferences.watermarkOpacity = 60;
myDocument.watermarkPreferences.watermarkRotation = -45;
myDocument.watermarkPreferences.watermarkHorizontalPosition =
    WatermarkHorizontalPositionEnum.watermarkHCenter;
myDocument.watermarkPreferences.watermarkHorizontalOffset = 0;
myDocument.watermarkPreferences.watermarkVerticalPosition =
    WatermarkVerticalPositionEnum.watermarkVCenter;
myDocument.watermarkPreferences.watermarkVerticalOffset = 0;
```

Disabling watermarks

After turning off the application setting for watermarks, InDesign no longer turns on the watermark settings for new documents by default. However, you can still set watermarks for individual documents. The following script fragment shows how to turn off application-level watermarks.

```
app.watermarkPreferences.watermarkVisibility = false;
```

You can turn off watermarks in an individual document at any time, as shown in the following script fragment.

```
app.documents.item(0).watermarkPreferences.watermarkVisibility = false;
```

Adjusting Page Sizes and Layout

InDesign allows different page sizes within a single InDesign document. For information on setting the default page size, see "Defining page size and document length".

You can also apply geometric transformations to individual pages.

Selecting pages

Before changing a page's size or applying a transformation to the page, you must select the page. In the InDesign user interface, you do this using the Page Tool on the Tools Panel. You can also select a page using scripting. The following script shows how. (For the complete script, see PageSelect.)

```
//Given a document with four pages (0, 1, 2, 3)...
var myDocument = app.activeDocument;
var myPages = myDocument.pages;
//Select page 1 and 2.
myPages.item(1).select();
myPages.item(2).select(SelectionOptions.ADD_TO);
//Select last page.
myDocument.select(myPages.item(-1), SelectionOptions.ADD_TO);
```

Resizing and reframing pages

You can resize or reframe page items on a page by scripting. You can also apply the resize and reframe operations to pages to change their sizes.

NOTE: Your minimum page size is determined by the page's margins. See "Setting page margins and columns" for more information.

The following script shows how to change a page's size using the resize method. (For the complete script, see PageResize.)

```
//Given a document with four pages (0, 1, 2, 3)...
var myDocument = app.activeDocument;
var myPages = myDocument.pages;
//Resize page to two times bigger
myPages.item(1).resize(CoordinateSpaces.INNER_COORDINATES,
                       AnchorPoint.CENTER_ANCHOR,
                       ResizeMethods.MULTIPLYING_CURRENT_DIMENSIONS_BY,
                       [2, 2]);
//Resize page to 400 points width and 600 points height.
myPages.item(2).resize(CoordinateSpaces.INNER_COORDINATES,
                       AnchorPoint.CENTER_ANCHOR,
                       ResizeMethods.REPLACING_CURRENT_DIMENSIONS_WITH,
                       [400, 600]);
```

Reframing changes the bounding box of a page, so reframing can be used to change a page's size by making the bounding box larger or smaller. The following script shows how to change a page's size using the reframe method. (For the complete script, see PageReframe.)

```
//Given a document with four pages (0, 1, 2, 3)...
var myDocument = app.activeDocument;
var myPages = myDocument.pages;
//Make the page one inch wider and one inch higher.
var myPage = myPages.item(1);
var myBounds = myPage.bounds;
var myY1 = myBounds[0];
var myX1 = myBounds[1];
var myY2 = myBounds[2]+72;
var myX2 = myBounds[3]+72;
myPage.reframe(CoordinateSpaces.INNER_COORDINATES, [[myX1, myY1], [myX2, myY2]]);
```

Transforming pages

Operations that change the geometry of objects are called transformations. The transform method can rotate, scale, shear, and move (translate) page items on a page and can also be used on pages. For technical details about transformation architecture, refer to "Transforming Page Items".

To transform a page:

1. Create a transformation matrix.
2. Apply the transformation matrix to the page using the transform method.

The following script shows how to transform a page with scripting. (For the complete script, see PageTransform.)

```
//Given a document with four pages (0, 1, 2, 3)...
var myDocument = app.activeDocument;
var myPages = myDocument.pages;

//Rotate a page around its center point.
var myRotateMatrix =
app.transformationMatrices.add({counterclockwiseRotationAngle:27});
myTransform(myPages.item(0), myRotateMatrix);

//Scale a page around its center point.
var myScaleMatrix = app.transformationMatrices.add({horizontalScaleFactor:0.8,
verticalScaleFactor:0.8});
myTransform(myPages.item(1), myScaleMatrix);

//Shear a page around its center point.
var myShearMatrix = app.transformationMatrices.add({clockwiseShearAngle:30});
myTransform(myPages.item(2), myShearMatrix);

function myTransform(myPage, myTransformationMatrix)
{
    myPage.transform(CoordinateSpaces.PASTEBOARD_COORDINATES,
                     AnchorPoint.CENTER_ANCHOR,
                     myTransformationMatrix);
}
```

Master page overlay

Because pages can have multiple sizes, it is possible for a page and its master page to be different sizes. In addition to tracking which master is applied, pages also maintain a matrix that determines how the master page draws on the page. This is called the Master Page Overlay. When you select a page using the Page Tool on the Tools Panel, you can see how the master page is positioned by checking the Show Master Page Overlay checkbox on the control panel. You can move the overlay around with the mouse. InDesign achieves this by applying a transform to the master overlay matrix. Although the user interface allows only translation (moving *x* and *y*), you can do more by scripting. The following script shows how to transform a master page overlay. (For the complete script, see MasterPageTransform.)

```
//Given a document with four pages (0, 1, 2, 3)...
var myDocument = app.activeDocument;
var myPages = myDocument.pages;

//Rotate master page overlay around its top-left corner.
var myRotateMatrix =
app.transformationMatrices.add({counterclockwiseRotationAngle:27});
myPages.item(0).masterPageTransform = myRotateMatrix;

//Scale master page overlay around its top-left corner.
var myScaleMatrix = app.transformationMatrices.add({horizontalScaleFactor:0.5,
verticalScaleFactor:0.5});
myPages.item(1).masterPageTransform = myScaleMatrix;

//Shear master page overlay around its top-left corner.
var myShearMatrix =app.transformationMatrices.add({clockwiseShearAngle:30});
myPages.item(2).masterPageTransform = myShearMatrix;

//Translate master page overlay 1 inch right and 2 inches down.
var myTranslateMatrix = app.transformationMatrices.add({horizontalTranslation:72,
verticalTranslation:144});
myPages.item(3).masterPageTransform = myTranslateMatrix;
```

Making an Adaptive Layout

Using InDesign's layout adaptation workflow, you create your primary layout the same way that you would create it for a document that will be printed, but then you can quickly adapt that layout to another orientation, to another dimension or aspect ratio, or to a completely different device class. This workflow mainly involves two features: Liquid Layout makes the layout flexible, and Create Alternate Layout creates alternative layouts for different orientations and device classes.

Making a flexible layout

Using layout rules, a designer can create one InDesign file with one set of physical pages that can display their content suitably on different devices with different sizes and orientations. You can optionally control the pages' appearance for each size and orientation. The following example script shows how to do this. (For the complete script, see LiquidLayout.)

```
var doc = app.documents.add();
var myPage = doc.pages.item(0);
//Set layout rule to LayoutRuleOptions.OBJECT_BASED
//Reposition and resize objects on the page as it resizes.
myPage.layoutRule = LayoutRuleOptions.OBJECT_BASED ;
//Create a text frame on the first page.
var myTextFrame = myPage.textFrames.add({
       geometricBounds:myGetBounds(doc, myPage),
       contents:"This is object-based layoutRule sample doc."
    });
//Create a rectangle
var myItem = myPage.rectangles.add({geometricBounds:[20,20,70,70]});
myItem.verticalLayoutConstraints =
    [DimensionsConstraints.flexibleDimension,
     DimensionsConstraints.flexibleDimension,
     DimensionsConstraints.flexibleDimension];
myItem.horizontalLayoutConstraints =
    [DimensionsConstraints.fixedDimension,
     DimensionsConstraints.flexibleDimension,
     DimensionsConstraints.flexibleDimension];

var doc2 = app.documents.add();
var myPage2 = doc2.pages.item(0);
//Set layout rule to LayoutRuleOptions.SCALE
//Scale objects on the page as it resizes.
myPage2.layoutRule = LayoutRuleOptions.SCALE;
//Create a text frame on the first page.
var myTextFrame2 = myPage2.textFrames.add({
       geometricBounds:myGetBounds(doc2, myPage2),
       contents:"This is scale layoutRule sample doc."
    });
var myItem2 = myPage2.rectangles.add({geometricBounds:[20,20,70,70]});

var doc3 = app.documents.add();
var myPage3 = doc3.pages.item(0);
//Set layout rule to LayoutRuleOptions.RECENTER
//Recenter objects on the page as it resizes.
myPage3.layoutRule = LayoutRuleOptions.RECENTER;
//Create a text frame on the first page.
var myTextFrame = myPage3.textFrames.add({
       geometricBounds:myGetBounds(doc3, myPage3),
       contents:"This is recenter layoutRule sample doc."
```

```
    });
var myItem = myPage3.rectangles.add({geometricBounds:[20,20,70,70]});

var doc4 = app.documents.add();
var myPage4 = doc4.pages.item(0);
//Set layout rule to LayoutRuleOptions.GUIDE_BASED
//Use guide slicing to resize objects on the page as it resizes.
myPage4.layoutRule = LayoutRuleOptions.GUIDE_BASED;
//Create a text frame on the first page.
var myTextFrame4 = myPage4.textFrames.add({
       geometricBounds:myGetBounds(doc4, myPage4),
        contents:"This is guide-based layoutRule sample doc."
    });
var myItem4 = myPage4.rectangles.add({geometricBounds:[20,20,70,70]});
with(doc4)
    {
       guides.add({
          guideType:GuideTypeOptions:LIQUID,
          location:40,
          orientation:HorizontalOrVertical.horizontal});
    }

var doc5 = app.documents.add();
var myPage5 = doc5.pages.item(0);
//Set layout rule to LayoutRuleOptions.USE_MASTER
//Use layout rule from the page's applied master page.
myPage5.layoutRule = LayoutRuleOptions.USE_MASTER;
//Create a text frame on the first page.
var myTextFrame5 = myPage5.textFrames.add(
       {geometricBounds:myGetBounds(doc5, myPage5),
       contents:"This is master page layoutRule sample doc."
    });
var myItem5 = myPage5.rectangles.add({geometricBounds:[20,20,70,70]});
```

Adding guides for a guide-based layout

The following example script shows how to add guide slices for guide-based layout rules. (For the complete script, see AddGuides.)

```
var doc = app.documents.add();
var myPage = doc.pages.item(0);
//Use guide slicing to resize objects on the page as it resizes.
myPage.layoutRule = LayoutRuleOptions.GUIDE_BASED;
//Create a text frame on the first page.
var myTextFrame = myPage.textFrames.add({
       geometricBounds:myGetBounds(doc, myPage),
       contents:"This is guide-based LayoutRule sample doc."
       });
var myItem = myPage.rectangles.add({geometricBounds:[20,20,70,70]});

with(doc)
{
    //add guide slice
    guides.add({
         guideType:GuideTypeOptions.LIQUID,
         location:20,
         orientation:HorizontalOrVertical.horizontal
       });
}
```

Setting constraints for an object-based layout

The following example script shows how to set constraints for object-based layout rules. (For the complete script, see SetConstraints.)

```
var doc = app.documents.add();
var myPage = doc.pages.item(0);
//Reposition and resize objects on the page as it resizes.
myPage.layoutRule = LayoutRuleOptions.OBJECT_BASED ;
//Create a text frame on the first page.
var myTextFrame = myPage.textFrames.add({
       geometricBounds:myGetBounds(doc, myPage),
       contents:"This is object-based layoutRule sample doc."
     });
//Create a rectagle
var myItem = myPage.rectangles.add({geometricBounds:[20,20,70,70]});
//set constrains
myItem.verticalLayoutConstraints =
    [DimensionsConstraints.flexibleDimension,
        DimensionsConstraints.flexibleDimension,
        DimensionsConstraints.flexibleDimension];
myItem.horizontalLayoutConstraints =
    [DimensionsConstraints.fixedDimension,
        DimensionsConstraints.flexibleDimension,
        DimensionsConstraints.flexibleDimension];
```

Creating alternative layouts

Create Alternate Layout takes a range of existing pages with content and automatically creates linked copies of the content on a set of new pages within the same document. Use this when you need to create unique layouts for different orientations and device classes. (For the complete script, see CreateAlternateLayout.)

```
myDoc = app.documents.add({
                        documentPreferences:
                        {
                            facingPages:false,
                            pagesPerDocument:1,
                            pageWidth:1024,
                            pageHeight:768
                         }
                });
//Reposition and resize objects on the page as it resizes.
myDoc.pages[0].layoutRule = LayoutRuleOptions.OBJECT_BASED ;

 with (myDoc)
 {
var myItem = pages[0].rectangles.add({
                                geometricBounds:[50,50,100,100]
                                });
//create alternate layout
myDoc.createAlternateLayout(myDoc.spreads,
                        "new layout",
                        myDoc.documentPreferences.pageHeight,
                        myDoc.documentPreferences.pageWidth,
                        true,
                        true,
                        LayoutRuleOptions.PRESERVE_EXISTING);
  }
```

Collecting and Dropping Content

With Content Dropper, you can copy the content from a file designed for printing and paste it into another document.

The following script shows how to collect content in a document using scripting. (For the complete script, see ContentCollectorAndPlacer.)

```
//Invoke load with full parameters
    contentPlacer.load(myPage.rectangles, true, true, true, true);
```

The following script shows how to place collected content into another document. (For the complete script, see ContentCollectorAndPlacer.)

```
//Invoke Page.contentPlace with full parameters
    myPage1.contentPlace(myPage.rectangles, //One or more page items to place or load
        true,            //Whether to link pageItems in content placer (Optional)
        true,            //Whether to link stories in content placer (Optional)
        true,            //Whether to map styles in content placer (Optional)
        [0, 0, 40, 40],  //The point at which to place (Optional)
        myDocument1.activeLayer,   //The layer on which to place (Optional)
        true);           //Whether to display the link options dialog (Optional)
```

Printing a Document

The following script prints the active document using the current print preferences. (For the complete script, see PrintDocument.)

```
app.activeDocument.print();
```

Printing using page ranges

To specify a page range to print, set the `pageRange` property of the document's `print preferences` object before printing, as shown in the following script fragment (from the PrintPageRange tutorial script):

```
//Prints a page range from the active document.
//Assumes that you have a document open, that it contains a page named "22".
//The page range can be either PageRange.allPages or a page range string.
//A page number entered in the page range must correspond to a page
//name in the document (i.e., not the page index). If the page name is
//not found, InDesign will display an error message.
app.activeDocument.printPreferences.pageRange = "22"
app.activeDocument.print(false);
```

Setting print preferences

The `print preferences` object contains properties corresponding to the options in the panels of the Print dialog. The following script shows how to set print preferences using scripting. (For the complete script, see PrintPreferences.)

```
//PrintPreferences.jsx
//An InDesign CS6 JavaScript
//Sets the print preferences of the active document.
with(app.activeDocument.printPreferences){
    //Properties corresponding to the controls in the General panel
    //of the Print dialog box. activePrinterPreset is ignored in this
    //example--we'll set our own print preferences. printer can be
    //either a string (the name of the printer) or Printer.postscriptFile.
    printer = Printer.postscriptFile;
    //Here's an example of setting the printer to a specific printer.
    //printer = "AGFA-SelectSet 5000SF v2013.108";
    //If the printer property is the name of a printer, then the ppd property
    //is locked (and will return an error if you try to set it).
    //ppd = "AGFA-SelectSet5000SF";
    //If the printer property is set to Printer.postscript file, the copies
    //property is unavailable. Attempting to set it will generate an error.
    //copies = 1;
    //If the printer property is set to Printer.postscript file, or if the
    //selected printer does not support collation, then the collating
    //property is unavailable. Attempting to set it will generate an error.
    //collating = false;
    reverseOrder = false;
    //pageRange can be either PageRange.allPages or a page range string.
    pageRange = PageRange.allPages;
    printSpreads = false;
    printMasterPages = false;
    //If the printer property is set to Printer.postScript file, then
    //the printFile property contains the file path to the output file.
    //printFile = "/c/test.ps";
    sequence = Sequences.all;
    //--------------------------------------------------------------------------
    //Properties corresponding to the controls in the
    //Output panel of the Print dialog box.
    //--------------------------------------------------------------------------
    //negative = true;
    //If a device independent PPD is specified, trying to set the colorOutput
    //parameter will result in an error.
    try{
```

```
    colorOutput = ColorOutputModes.separations;
    //Note the lowercase "i" in "Builtin"
    trapping = Trapping.applicationBuiltin;
    flip = Flip.none;
}
catch(myError){}
//If trapping is on, attempting to set the following
//properties will generate an error.
try{
    if(trapping == Trapping.off){
       printBlack = true;
       printCyan = true;
       printMagenta = true;
       printYellow = true;
    }
}
catch(myError){}
//Only change the ink angle and frequency when you want to override the
//screening set by the screening specified by the screening property.
//blackAngle = 45;
//blackFrequency = 175;
//cyanAngle = 15;
//cyanFrequency = 175;
//magentaAngle = 75;
//magentaFreqency = 175;
//yellowAngle = 0;
//yellowFrequency = 175;
//The following properties are not needed (because
//colorOutput is set to separations).
//compositeAngle = 45;
//compositeFrequency = 175;
//simulateOverprint = false;
//If trapping is on, setting the following properties will produce an error.
try{
    if(trapping == Trapping.off){
    printBlankPages = false;
    printGuidesGrids = false;
    printNonprinting = false;
    }
}
catch(myError){}
//-----------------------------------------------------------------------
//Properties corresponding to the controls in the
//Setup panel of the Print dialog box.
//-----------------------------------------------------------------------
try{
    paperSize = PaperSizes.custom;
    //Page width and height are ignored if paperSize is not PaperSizes.custom.
    //paperHeight = 1200;
    //paperWidth = 1200;
    printPageOrientation = PrintPageOrientation.portrait;
    pagePosition = PagePositions.centered;
    paperGap = 0;
    paperOffset = 0;
    paperTransverse = false;
    scaleHeight = 100;
    scaleWidth = 100;
    scaleMode = ScaleModes.scaleWidthHeight;
    scaleProportional = true;
}
```

```
catch(myError){}
//If trapping is on, attempting to set the
//following properties will produce an error.
try{
    if(trapping == Trapping.off){
        textAsBlack = false;
        thumbnails = false;
        //The following properties is not needed because
        //thumbnails is set to false.
        //thumbnailsPerPage = 4;
        tile = false;
        //The following properties are not needed because tile is set to false.
        //tilingOverlap = 12;
        //tilingType = TilingTypes.auto;
    }
}
catch(myError){}
//-----------------------------------------------------------------------
//Properties corresponding to the controls in the Marks and Bleed
//panel of the Print dialog box.
//-----------------------------------------------------------------------
//Set the following property to true to print all printer's marks.
//allPrinterMarks = true;
useDocumentBleedToPrint = false;
//If useDocumentBleedToPrint = false then setting
//any of the  bleed properties
//will result in an error.
//Get the bleed amounts from the document's bleed and add a bit.
bleedBottom = app.activeDocument.documentPreferences.
documentBleedBottomOffset+3;
bleedTop = app.activeDocument.documentPreferences.documentBleedTopOffset+3;
bleedInside = app.activeDocument.documentPreferences.
documentBleedInsideOrLeftOffset+3;
bleedOutside = app.activeDocument.documentPreferences.
documentBleedOutsideOrRightOffset+3;
//If any bleed area is greater than zero, then export the bleed marks.
if(bleedBottom == 0 && bleedTop == 0 && bleedInside == 0 &&
bleedOutside == 0){
    bleedMarks = true;
}
else{
    bleedMarks = false;
}
colorBars = true;
cropMarks = true;
includeSlugToPrint = false;
markLineWeight = MarkLineWeight.p125pt
markOffset = 6;
//markType = MarkTypes.default;
pageInformationMarks = true;
registrationMarks = true;
//-----------------------------------------------------------------------
//Properties corresponding to the controls in the
//Graphics panel of the Print dialog box.
//-----------------------------------------------------------------------
//If a device independent PPD is specified, trying to set the graphics
//send data will result in an error.
try{
    sendImageData = ImageDataTypes.allImageData;
}
```

```
    catch(myError){}
    fontDownloading = FontDownloading.complete;
    downloadPPDFOnts = true;
    try{
        dataFormat = DataFormat.binary;
    }
    catch(e){}
    try{
        postScriptLevel = PostScriptLevels.level3;
    }
    catch(e){}
    //-------------------------------------------------------------------------
    //Properties corresponding to the controls in the Color Management
    //panel of the Print dialog box.
    //-------------------------------------------------------------------------
    //If the useColorManagement property of app.colorSettings is false,
    //attempting to set the following properties will return an error.
    try{
        sourceSpace = SourceSpaces.useDocument;
        intent = RenderingIntent.useColorSettings;
        crd = ColorRenderingDictionary.useDocument;
        profile = Profile.postscriptCMS;
    }
    catch(e){}
    //-------------------------------------------------------------------------
    //Properties corresponding to the controls in the Advanced
    //panel of the Print dialog box.
    //-------------------------------------------------------------------------
    opiImageReplacement = false;
    omitBitmaps = false;
    omitEPS = false;
    omitPDF = false;
    //The following line assumes that you have a flattener
    //preset named "high quality flattener".
    try{
        flattenerPresetName = "high quality flattener";
    }
    catch(e){}
    ignoreSpreadOverrides = false;
}
```

Printing with printer presets

To print a document using a printer preset, include the printer preset in the `print` command.

Exporting a Document as PDF

InDesign scripting offers full control over the creation of PDF files from your page-layout documents.

Exporting to PDF

The following script exports the current document as PDF, using the current PDF export options. (For the complete script, see ExportPDF.)

```
app.activeDocument.exportFile(ExportFormat.pdfType, File("/c/myTestDocument.pdf"),
false);
```

The following script fragment shows how to export to PDF using a PDF export preset. (For the complete script, see ExportPDFWithPreset.)

```
var myPDFExportPreset = app.pdfExportPresets.item("prepress");
app.activeDocument.exportFile(ExportFormat.pdfType, File("/c/myTestDocument.pdf"),
false, myPDFExportPreset);
```

Setting PDF export options

The following script sets the PDF export options before exporting. (For the complete script, see ExportPDFWithOptions.)

```
with(app.pdfExportPreferences){
    //Basic PDF output options.
    pageRange = PageRange.allPages;
    acrobatCompatibility = AcrobatCompatibility.acrobat6;
    exportGuidesAndGrids = false;
    exportLayers = false;
    exportNonPrintingObjects = false;
    exportReaderSpreads = false;
    generateThumbnails = false;
    try{
        ignoreSpreadOverrides = false;
    }
    catch(e){}
    includeBookmarks = true;
    includeHyperlinks = true;
    includeICCProfiles = true;
    includeSlugWithPDF = false;
    includeStructure = false;
    interactiveElementsOption = InteractiveElementsOptions.doNotInclude;
    //Setting subsetFontsBelow to zero disallows font subsetting;
    //set subsetFontsBelow to some other value to use font subsetting.
    subsetFontsBelow = 0;
    //
    //Bitmap compression/sampling/quality options.
    colorBitmapCompression = BitmapCompression.zip;
    colorBitmapQuality = CompressionQuality.eightBit;
    colorBitmapSampling = Sampling.none;
    //thresholdToCompressColor is not needed in this example.
    //colorBitmapSamplingDPI is not needed when colorBitmapSampling
    //is set to none.
    grayscaleBitmapCompression = BitmapCompression.zip;
    grayscaleBitmapQuality = CompressionQuality.eightBit;
    grayscaleBitmapSampling = Sampling.none;
    //thresholdToCompressGray is not needed in this example.
    //grayscaleBitmapSamplingDPI is not needed when grayscaleBitmapSampling
    //is set to none.
    monochromeBitmapCompression = BitmapCompression.zip;
    monochromeBitmapSampling = Sampling.none;
    //thresholdToCompressMonochrome is not needed in this example.
    //monochromeBitmapSamplingDPI is not needed when
    //monochromeBitmapSampling is set to none.
    //
    //Other compression options.
    compressionType = PDFCompressionType.compressNone;
    compressTextAndLineArt = true;
    cropImagesToFrames = true;
    optimizePDF = true;
```

```
    //
    //Printers marks and prepress options.
    //Get the bleed amounts from the document's bleed.
    bleedBottom = app.activeDocument.documentPreferences.
    documentBleedBottomOffset;
    bleedTop = app.activeDocument.documentPreferences.documentBleedTopOffset;
    bleedInside = app.activeDocument.documentPreferences.
    documentBleedInsideOrLeftOffset;
    bleedOutside = app.activeDocument.documentPreferences.
    documentBleedOutsideOrRightOffset;
    //If any bleed area is greater than zero, then export the bleed marks.
    if(bleedBottom == 0 && bleedTop == 0 && bleedInside == 0 &&
    bleedOutside == 0){
        bleedMarks = true;
    }
    else{
        bleedMarks = false;
    }
    colorBars = true;
    colorTileSize = 128;
    grayTileSize = 128;
    cropMarks = true;
    omitBitmaps = false;
    omitEPS = false;
    omitPDF = false;
    pageInformationMarks = true;
    pageMarksOffset = 12;
    pdfColorSpace = PDFColorSpace.unchangedColorSpace;
    //Default mark type.
    pdfMarkType = 1147563124;
    printerMarkWeight = PDFMarkWeight.p125pt;
    registrationMarks = true;
    try{
        simulateOverprint = false;
    }
    catch(e){}
    useDocumentBleedWithPDF = true;
    //Set viewPDF to true to open the PDF in Acrobat or Adobe Reader.
    viewPDF = false;
}
//Now export the document. You'll have to fill in your own file path.
app.activeDocument.exportFile(ExportFormat.pdfType, File("/c/myTestDocument.pdf"),
false);
```

Exporting to grayscale PDF

An InDesign document that contains colors can be exported to a color PDF or to a grayscale PDF. You can modify the script in the preceding "Setting PDF export options" on page 52 section to export in grayscale by changing the `PDFColorSpace` option from `unchangedColorSpace` to gray. For example:

```
with(app.pdfExportPreferences)
{
    pdfColorSpace = PDFColorSpace.GRAY;
}
```

(For a different script that creates a small document containing color and exports it to PDF in grayscale, see GreyscalePDFforIDS.)

Exporting a range of pages to PDF

The following script shows how to export a specified page range as PDF. (For the complete script, see ExportPageRangeAsPDF.)

```
with(app.pdfExportPreferences){
    //pageRange can be either PageRange.allPages or a page range string
    //(just as you would enter it in the Print or Export PDF dialog box).
    pageRange = "1, 3-6, 7, 9-11, 12";
}
var myPDFExportPreset = app.pdfExportPresets.item("prepress")
app.activeDocument.exportFile(ExportFormat.pdfType, File("/c/myTestDocument.pdf"),
false, myPDFExportPreset);
```

Exporting individual pages to PDF

The following script exports each page from a document as an individual PDF file. (For the complete script, see ExportEachPageAsPDF.)

```
//Display a "choose folder" dialog box.
if(app.documents.length != 0){
    var myFolder = Folder.selectDialog ("Choose a Folder");
    if(myFolder != null){
        myExportPages(myFolder);
    }
}
else{
    alert("Please open a document and try again.");
}
function myExportPages(myFolder){
    var myPageName, myFilePath, myFile;
    var myDocument = app.activeDocument;
    var myDocumentName = myDocument.name;
    var myDialog = app.dialogs.add();
    with(myDialog.dialogColumns.add().dialogRows.add()){
        staticTexts.add({staticLabel:"Base name:"});
        var myBaseNameField = textEditboxes.add({editContents:myDocumentName,
        minWidth:160});
    }
    var myResult = myDialog.show({name:"ExportPages"});
    if(myResult == true){
        var myBaseName = myBaseNameField.editContents;
        //Remove the dialog box from memory.
        myDialog.destroy();
        for(var myCounter = 0; myCounter < myDocument.pages.length;
        myCounter++){
            myPageName = myDocument.pages.item(myCounter).name;
            app.pdfExportPreferences.pageRange = myPageName;
            //The name of the exported files will be the base name + the
```

```
            //page name + ".pdf".
            //If the page name contains a colon (as it will if the
            //document contains sections),
            //then remove the colon.
            var myRegExp = new RegExp(":","gi");
            myPageName = myPageName.replace(myRegExp, "_");
            myFilePath = myFolder + "/" + myBaseName + "_" + myPageName + ".pdf";
            myFile = new File(myFilePath);
            myDocument.exportFile(ExportFormat.pdfType, myFile, false);
        }
    }
    else{
        myDialog.destroy();
    }
}
```

Exporting PDF with interactive features

The following script exports a document with interactive features as a PDF. (For the complete script, see ExportInteractivePDF.)

```
//Given a document "myDocument," add page transitions...
for(var myCounter = 0; myCounter < myDocument.spreads.length; myCounter++){
    myDocument.spreads.item(myCounter).pageTransitionType
    PageTransitionTypeOptions.wipeTransition;
    myDocument.spreads.item(myCounter).pageTransitionDirection =
    PageTransitionDirectionOptions.down;
    myDocument.spreads.item(myCounter).pageTransitionDuration =
    PageTransitionDurationOptions.medium;
}
app.interactivePDFExportPreferences.flipPages = true;
app.interactivePDFExportPreferences.flipPagesSpeed = 5;
app.interactivePDFExportPreferences.openInFullScreen = true;
app.interactivePDFExportPreferences.interactivePDFInteractiveElementsOption =
InteractivePDFInteractiveElementsOptions.includeAllMedia;
//Export the document to PDF.
myDocument.exportFile(ExportFormat.interactivePDF,File(Folder.desktop +
"/InteractivePDF.pdf"), false);
```

Exporting as a PDF form

To create a PDF that contains form fields, you can either save the InDesign document as PDF, then open it in Acrobat and apply form fields, or apply text fields, radio buttons, signature fields, and so on directly in InDesign and export the document to PDF as a form with no additional Acrobat editing required.

The following script applies form elements to an InDesign document and exports the document as a PDF form. (For the complete script, see ExportInteractivePDFForm.)

```
var myTextFrame = myDocument.pages.item(0).textFrames.add
(
    {geometricBounds:[15, 15, 20, 35], contents:"FirstName: "}
);
//Create a textbox as firstname input box
var myTextBox = myDocument.pages.item(0).textBoxes.add
(
    {geometricBounds:[15, 40, 20, 75], contents:SpecialCharacters.autoPageNumber}
);
//Create another textframe as lastname label
var myTextFrame1 = myDocument.pages.item(0).textFrames.add
(
    {geometricBounds:[30, 15, 25, 35], contents:"LastName: "}
);
//Create another textbox as lastname input box
var myTextBox = myDocument.pages.item(0).textBoxes.add
(
    {geometricBounds:[30, 40, 25, 75], contents:SpecialCharacters.autoPageNumber}
);

 //Create a TextFrame to introduce the following checkbox
 var myTextFrame2 = myDocument.pages.item(0).textFrames.add
 (
    {geometricBounds:[15, 80, 20, 95], contents:"Hobby: "}
 );
 //Create some CheckBoxes
 var myCheckBox = myDocument.pages.item(0).checkBoxes.add
 (
    {geometricBounds:[15, 100, 20, 105], name:"Football "}
 );
 var myTextFrame3 = myDocument.pages.item(0).textFrames.add
 (
    {geometricBounds:[15, 107, 20, 125], contents:"Football "}
 );
 var myCheckBox1 = myDocument.pages.item(0).checkBoxes.add
 (
    {geometricBounds:[15, 130, 20, 135], name:"Basketball "}
 );
 var myTextFrame4 = myDocument.pages.item(0).textFrames.add
 (
    {geometricBounds:[15, 137, 20, 160], contents:"Basketball "}
 );
 var myCheckBox2 = myDocument.pages.item(0).checkBoxes.add
 (
    {geometricBounds:[15, 165, 20, 170], name:"Pingpong "}
 );
 var myTextFrame5 = myDocument.pages.item(0).textFrames.add
 (
    {geometricBounds:[15, 172, 20, 193], contents:"Pingpong "}
 );

//Create a button for submit
var submitButton = myDocument.pages.item(0).buttons.add
(
    {geometricBounds:[45, 15, 35, 35], name:"Submit"}
);
//Fill contents to the button
var myRightArrow1 = submitButton.states.item(0).polygons.add
(
    {fillColor:myDocument.colors.item("Green")}
```

```
);
myRightArrow1.paths.item(0).entirePath = [[15, 35],[35,40],[15, 45]];
//Add the Rollover state.
var myRolloverState1 = submitButton.states.add();
//Add a shadow to the polygon in the Rollover state.
var myRolloverArrow1 = myRolloverState1.polygons.add
(
    {fillColor:myDocument.colors.item("Green")}
);
myRolloverArrow1.paths.item(0).entirePath = [[15, 35],[35,40],[15, 45]];
var myFillTransparencySettings1 = myRolloverArrow1.fillTransparencySettings;
myFillTransparencySettings1.dropShadowSettings.mode = ShadowMode.drop;
myFillTransparencySettings1.dropShadowSettings.angle = 90;
myFillTransparencySettings1.dropShadowSettings.xOffset = 0;
myFillTransparencySettings1.dropShadowSettings.yOffset = 0;
myFillTransparencySettings1.dropShadowSettings.size = 6;
//Add a shadow to the polygon in the Click state.
var myClickState1 = submitButton.states.add();
var myClickArrow1 = myClickState1.polygons.add
(
    {fillColor:myDocument.colors.item("Blue")}
);
myClickArrow1.paths.item(0).entirePath = [[15, 35],[35,40],[15,45]];
//Set the behavior for the button.
var SubmitForm = submitButton.submitFormBehaviors.add
(
    {behaviorEvent:BehaviorEvents.mouseUp}
);

 //Create a button for print
var printButton = myDocument.pages.item(0).buttons.add
(
    {geometricBounds:[45, 40, 35, 60], content:"Print"}
);
//Fill contents to the button
var myRightArrow2 = printButton.states.item(0).polygons.add
(
    {fillColor:myDocument.colors.item("Red")}
);
myRightArrow2.paths.item(0).entirePath = [[40, 35],[60,40],[40, 45]];
//Add the Rollover state.
var myRolloverState2 = printButton.states.add();
//Add a shadow to the polygon in the Rollover state.
var myRolloverArrow2 = myRolloverState2.polygons.add
(
    {fillColor:myDocument.colors.item("Red")}
);
myRolloverArrow2.paths.item(0).entirePath = [[40, 35],[60,40],[40, 45]];
var myFillTransparencySettings2 = myRolloverArrow2.fillTransparencySettings;
myFillTransparencySettings2.dropShadowSettings.mode = ShadowMode.drop;
myFillTransparencySettings2.dropShadowSettings.angle = 90;
myFillTransparencySettings2.dropShadowSettings.xOffset = 0;
myFillTransparencySettings2.dropShadowSettings.yOffset = 0;
myFillTransparencySettings2.dropShadowSettings.size = 6;
//Add a shadow to the polygon in the Click state.
```

```
var myClickState2 = printButton.states.add();
var myClickArrow2 = myClickState2.polygons.add
(
    {fillColor:myDocument.colors.item("Blue")}
);
myClickArrow2.paths.item(0).entirePath = [[40, 35],[60,40],[40,45]];
//Set the behavior for the button.
var PrintForm = printButton.printFormBehaviors.add
(
    {behaviorEvent:BehaviorEvents.mouseUp}
);

//Export the document to interactive PDF.
myDocument.exportFile(ExportFormat.interactivePDF,File(Folder.desktop +
"/SubmitForm.pdf"), false)
```

Exporting Pages as EPS

When you export a document as EPS, InDesign saves each page of the file as a separate EPS graphic (an EPS, by definition, can contain only a single page). If you export more than a single page, InDesign appends the index of the page to the filename. The index of the page in the document is not necessarily the name of the page (as defined by the section options for the section containing the page).

Exporting all pages to EPS

The following script exports the pages of the active document to one or more EPS files. (For the complete script, see ExportAsEPS.)

```
var myFile = new File("/c/myTestFile.eps");
app.activeDocument.exportFile(ExportFormat.epsType, myFile, false);
```

Exporting a range of pages to EPS

To control which pages are exported as EPS, set the `page range` property of the EPS export preferences to a page-range string containing the page or pages you want to export, before exporting. (For the complete script, see ExportPageRangeAsEPS.)

```
//Enter the name of the page you want to export in the following line.
//Note that the page name is not necessarily the index of the page in the
//document (e.g., the first page of a document whose page numbering starts
//with page 21 will be "21", not 1).
app.epsExportPreferences.pageRange = "1-3, 6, 9";
var myFile = new File("/c/myTestFile.eps");
app.activeDocument.exportFile(ExportFormat.epsType, myFile, false);
```

Exporting as EPS with file naming

The following script exports each page as an EPS, but it offers more control over file naming than the earlier example. (For the complete script, see ExportEachPageAsEPS.)

```
//Display a "choose folder" dialog box.
if(app.documents.length != 0){
    var myFolder = Folder.selectDialog ("Choose a Folder");
    if(myFolder != null){
        myExportPages(myFolder);
    }
}
else{
    alert("Please open a document and try again.");
}
function myExportPages(myFolder){
    var myFilePath, myPageName, myFile;
    var myDocument = app.activeDocument;
    var myDocumentName = myDocument.name;
    var myDialog = app.dialogs.add({name:"ExportPages"});
    with(myDialog.dialogColumns.add().dialogRows.add()){
        staticTexts.add({staticLabel:"Base name:"});
        var myBaseNameField = textEditboxes.add({editContents:myDocumentName,
        minWidth:160});
    }
    var myResult = myDialog.show();
    if(myResult == true){
        //The name of the exported files will be the base name +
        //the page name + ".eps".
        var myBaseName = myBaseNameField.editContents;
        //Remove the dialog box from memory.
        myDialog.destroy();
        //Generate a file path from the folder name, the base document name,
        //page name.
        for(var myCounter = 0; myCounter < myDocument.pages.length;
        myCounter++){
            myPageName = myDocument.pages.item(myCounter).name;
            app.epsExportPreferences.pageRange = myPageName;
            //The name of the exported files will be the base name +
            //the page name + ".eps".
            //If the page name contains a colon (as it will if the
            //document contains sections),
            //then remove the colon.
            var myRegExp = new RegExp(":","gi");
            myPageName = myPageName.replace(myRegExp, "_");
            myFilePath = myFolder + "/" + myBaseName + "_" + myPageName + ".eps";
            myFile = new File(myFilePath);
            app.activeDocument.exportFile(ExportFormat.epsType, myFile, false);
        }
    }
    else{
        myDialog.destroy();
    }
}
```

Exporting to EPub

InDesign scripting offers full control over the creation of EPub files from your page-layout documents.

Exporting the current document

The following script exports the current document as EPub, using default options. (For the complete script, see ExportEPub.)

```
var myDocument = app.documents.item(0);
myDocument.exportFile(ExportFormat.EPUB, File("C:/test/ExportEPub.epub"), false);
```

To give the user more control over the the creation of EPub files, specify true for the third parameter. This opens the EPub export options dialog.

Setting EPub export options

The following script sets the EPub export options before exporting. (For the complete script, see ExportEPubWithOptions.)

```
//Sets EPub export options, then exports the active document as EPub.
with (myDocument.epubExportPreferences)
{
    //Apply image alignment to anchored object settings.
    applyImageAlignmentToAnchoredObjectSettings = false;

    //The unit of space.
    spaceUnit = SpaceUnitType.CSS_EM;
    //The unit of margin.
    marginUnit = SpaceUnitType.CSS_EM;

    //Bottom margin of the epub.
    bottomMargin = 5;
    //Left margin of the epub.
    leftMargin = 5;
    //Right margin of the epub.
    rightMargin = 5;
    //Top margin of the epub.
    topMargin = 5;

    //If true, break InDesign document into smaller pieces when generating epub.
    breakDocument = false;
    //The name of paragraph style to break InDesign document.
    //paragraphStyleName = "";

    //The bullet export option.
    bulletExportOption = BulletListExportOption.AS_TEXT;

    cssExportOption = StyleSheetExportOption.EMBEDDED_CSS;
    customImageSizeOption = ImageSizeOption.SIZE_RELATIVE_TO_PAGE_WIDTH;

    embedFont = true;
    epubCover = EpubCover.FIRST_PAGE;
    //This will take effect only when epubCover is set to EXTERNAL_IMAGE.
    //coverImageFile = "/c/cover.jpg";

    epubPublisher = "Adobe Devtech";

    //The export order.
    exportorder = ExportOrder.LAYOUT_ORDER;

    //If true, output footnote immediately after its paragraph.
```

```
    footnoteFollowParagraph = false;

    //If true, export epub in XHTML format. Otherwise, in DTBook format.
    format = true;

    gifOptionsInterlaced = true;
    gifOptionsPalette = GIFOptionsPalette.WINDOWS_PALETTE;

    //The epub unique identifier, like ISBN.
    id = "123";

    //Ignore object level image conversion settings.
    ignoreObjectConversionSettings = true;

    //Alignment applied to images.
    imageAlignment = ImageAlignmentType.ALIGN_CENTER;
    //The file format to use for converted images.
    //Valid only when copy optimized images and/or copy formatted images is true.
    imageConversion = ImageConversion.AUTOMATIC;
    imageExportResolution = ImageResolution.PPI_150;
    //Image page break settings to be used with objects.
    imagePageBreak = ImagePageBreakType.PAGE_BREAK_AFTER;
    //Space After applied to images.
    imageSpaceAfter = 2;
    //Space Before applied to images.
    imageSpaceBefore = 2;

    //If true, include CSS definition.
    includeCSSDefinition = true;
    //If true, output document metadata into epub.
    includeDocumentMetadata = true;

    jpegOptionsFormat = JPEGOptionsFormat.BASELINE_ENCODING;
    jpegOptionsQuality = JPEGOptionsQuality.HIGH;

    //The PNG compression level.
    level = 5;

    numberedListExportOption = NumberedListExportOption.AS_TEXT;

    //If true, format image based on layout appearence.
    preserveLayoutAppearence = true;
    //If true, output local style override.
    preserveLocalOverride = true;

    stripSoftReturn = true;
    //If true, image page break settings will be used in objects.
    useImagePageBreak = true;
    //Use InDesign TOC style to generate epub TOC.
    useTocStyle = true;

    viewDocumentAfterExport = false;
}
myDocument.exportFile(ExportFormat.EPUB, File("C:/test/ExportEPubWithOptions.epub"),
false);
```

4 Working with Layers

Chapter Update Status	
CS6	Unchanged

InDesign's layers are the key to controlling the stacking order of objects in your layout. You can think of layers as transparent planes stacked on top of each other. You also can use layers as an organizational tool, putting one type of content on a given layer or set of layers.

A document can contain one or more layers, and each document includes at least one layer. Layers are document wide, not bound to specific pages or spreads.

This chapter covers scripting techniques related to layers in an InDesign layout and discusses common operations involving layers.

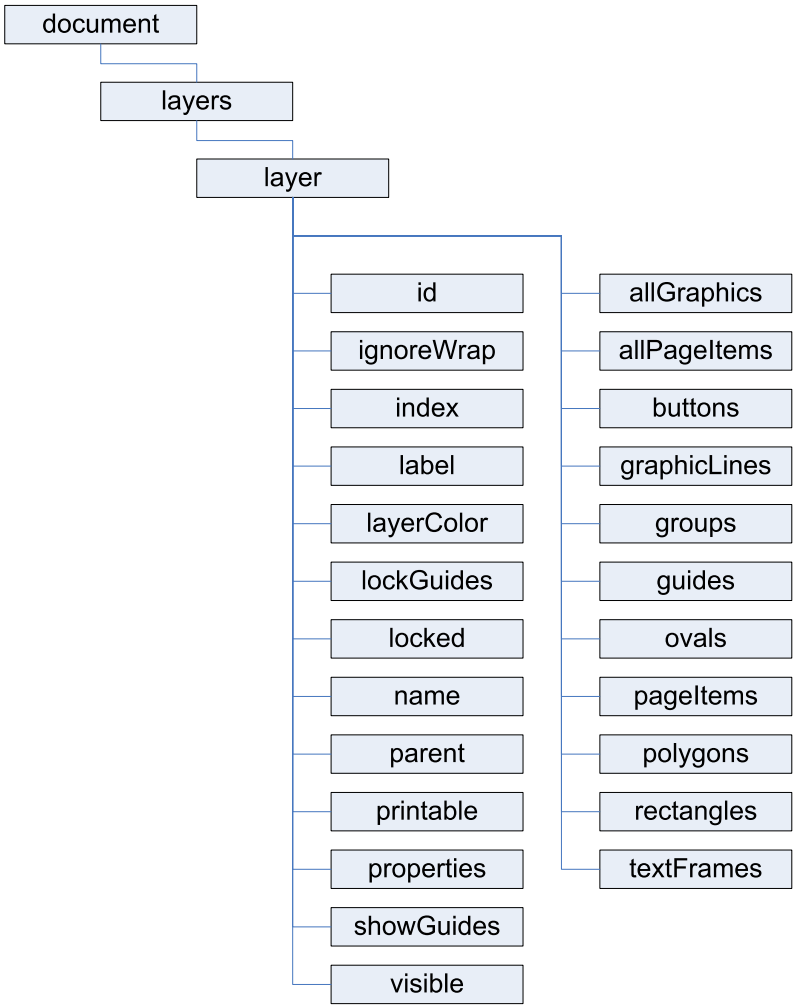
Understanding the Layer Object Model

The following figure shows the layer object model. Note the following about the diagram:

- It focuses on the location of a layer and its contents in the context of the object hierarchy of a document; it does not attempt to show all the other ways a script might work with the content of a layer (e.g., you can get a reference to a text-frame object from a story, text object, page, or spread, in addition to finding it inside a layer object).
- It uses the JavaScript form of the object names; however, the object hierarchy is the same in all scripting languages.
- The basic properties of a layer are shown in the column at the left of the figure; the objects that may be contained by the layer object, at the right.

It is important to note the distinction between the page-items collection and the `allPageItems` collection. The former is a collection containing only the top-level page items in a layer. If a page item is inside a group, for example, it will not appear in the `pageItems` collection. In contrast, the `allPageItems` collection is a flattened collection of all page items assigned to the layer, regardless of their location in the object hierarchy. A page item inside a group on the layer would appear in the `allPageItems` collection.

Similarly, the `allGraphics` property contains all graphics stored in page items assigned to the layer, regardless of their location in the object hierarchy.



Scripting Layers

In InDesign's user interface, you add, delete, rearrange, duplicate, and merge layers using the Layers panel. You also can change the layer to which a selected page item is assigned by dragging and dropping the layer proxy in the Layers panel. (For more on assigning objects to a layer, see the InDesign online help.) This section shows how to accomplish these tasks using InDesign scripting.

Creating layers

The following script fragment shows how to create a new layer. (For the complete script, see AddLayer.)

```
//Given a document "myDocument"...
var myLayer = myDocument.layers.add();
```

When you create a new layer, the layer appears above all other layers in the document.

Referring to layers

InDesign scripting offers several ways to refer to a layer object. This section describes the most common ways to refer to layers.

Getting the active layer

The *active layer* is the layer on which new objects are created. You can get the active layer using scripting, as shown in the following script fragment. (For the complete script, see ActiveLayer.)

```
//Given a document "myDocument"...
var myDocument = app.documents.item(0);
var myLayer = myDocument.activeLayer;
```

Referring to layers by layer index

You can get a reference to a layer using the index of the layer in the layers collection of a document. The script fragment below uses the layer index to iterate through layers. (For the complete script, see HideOtherLayers.)

```
//Given a document "myDocument"...
var myTargetLayer = myDocument.activeLayer;
for(var myCounter = 0; myCounter < myDocument.layers.length; myCounter++){
    //If the layer is not the target layer, hide it.
    if(myDocument.layers.item(myCounter).name != myTargetLayer.name){
        myDocument.layers.item(myCounter).visible = false;
    }
}
```

Note that you can use negative numbers to refer to the layers in the layers collection of a document. Layer -1 refers to the last (bottom) layer in the collection.

Referring to layers by layer name

You also can get a reference to a layer using the name of the layer, as shown in the following script fragment. (For the complete script, see LayerName.)

```
var myLayer = app.documents.item(0).layers.item("Text Layer");
```

Using relative references

Given a layer, you can refer to the layer above using the `previousItem` method, or refer to the layer below using the `nextItem` method, as shown in the following script fragment. (For the complete script, see RelativeLayerReferences.) Both methods take a reference layer as a parameter.

```
//Given a document "myDocument"...
var myLayer = myDocument.layers.item(4);
myDocument.activeLayer = myLayer;
var myNextLayer = myDocument.layers.nextItem(myLayer);
var myPreviousLayer = myDocument.layers.previousItem(myLayer);
var myString = "The layer below the target layer is " + myNextLayer.name + "\r";
myString += "The layer above the target layer is " + myPreviousLayer.name;
alert(myString);
```

The `previousItem` and `nextItem` methods return an invalid layer reference if the specified (next or previous) layer does not exist, rather than generating an error.

Referring to ranges of layers

To refer to a series of layers, you can use the `itemByRange` method. The following script fragment shows how to get a reference to a range of layers, then set a property on all layers in the range. (For the complete script, see HideLayersAbove.)

```
//Given a document "myDocument"...
var myLayer = myDocument.layers.item(4);
myDocument.activeLayer = myLayer;
//Now hide all of the layers above the current layer.
var myLayers = myDocument.layers.itemByRange(0, myLayer.index -1);
//Even though the result contains multiple layers, you can
//set a property on all of the layers without iterating.
myLayers.visible = false;
```

Deleting layers

Use the remove method to delete a layer from a specific document, as shown in the following script fragment. (For the complete script, see DeleteLayer.) You cannot delete the last remaining layer in a document.

```
//Given a document "myDocument" containing a layer named "Delete This Layer"...
var myLayer = myDocument.layers.item("Delete This Layer");
myLayer.remove();
```

Moving layers

Use the `move` method to change the stacking order of layers in a document, as shown in the following script fragment. (For the complete script, see MoveLayer.)

```
//Given a document "myDocument" containing at least two layers...
var myLayerA = myDocument.layers.item(0);
var myLayerB = myDocument.layers.item(1);
myLayerA.move(LocationOptions.AFTER, myLayerB);
```

Duplicating layers

Use the `duplicate` method to create a copy of a layer, as shown in the following script fragment. (For the complete script, see DuplicateLayer.)

```
//Given a layer "myLayer"...
ar myNewLayer = myLayer.duplicate();
```

Merging layers

The following script fragment shows how to merge two or more layers, including the page items assigned to them, into a single layer. (For the complete script, see MergeLayers.)

```
//Given the layers "myLayer1" and "myLayer2"...
myLayer1.merge(myLayer2);
```

Assigning page items to layers

You can assign a page item to a layer by either referring to the layer when you create the page item (the `add` method of all page items can take a layer as a parameter) or setting the `itemLayer` property of an existing page item. The following script fragment shows how to assign a page item to a layer using both techniques. (For the complete script, see AssignPageItemsToLayers.)

```
//Given a reference to a page "myPage," and a document "myDocument,"
//create a text frame on a layer named "TextFrames"
var myTextFrame = myPage.textFrames.add(myDocument.layers.item("TextFrames"));
myTextFrame.geometricBounds = [72, 72, 144, 144];
//Create a rectangle on the current target layer.
var myRectangle = myPage.rectangles.add({geometricBounds:[72, 144, 144, 216]});
//Move the rectangle to a specific layer.
myRectangle.itemLayer = myDocument.layers.item("Rectangles");
//Create a series of ovals.
for(var myCounter = 72; myCounter < 172; myCounter+=10){
    myPage.ovals.add({geometricBounds:[216, myCounter, 226, myCounter+10]});
}
//Move all of the ovals on the page to a specific layer.
myPage.ovals.everyItem().itemLayer = myDocument.layers.item("Ovals");
```

Setting layer properties

Layer properties control the layer name, color, visibility, and other attributes of a layer. This section shows how to work with layer properties.

Setting basic layer properties

Basic layer properties include the name of the layer, the highlight color of the layer, the visibility of the layer, and whether text objects on the layer ignore text-wrap settings. The following script fragment shows how to set these basic properties of a layer. (For the complete script, see BasicLayerProperties.)

```
//Given a document "myDocument"...
var myLayer = myDocument.layers.add();
myLayer.name = "myLayer";
myLayer.layerColor = UIColors.CHARCOAL;
myLayer.ignoreWrap = false;
myLayer.visible = true;
```

Working with layer guides

Guides can be assigned to a specific layer, just like page items. You can choose to show or hide the guides for a layer, and you can lock or unlock the guides on a layer. The following script fragment shows how to work with the guides on a layer. (For the complete script, see LayerGuides.)

```
//Given a document "myDocument" and a page "myPage" containing at least one guide...
//Create a new layer.
var myLayer = myDocument.layers.add();
//Move all of the guides on the page to the new layer.
myPage.guides.everyItem().itemLayer = myLayer;
myLayer.lockGuides = true;
myLayer.showGuides = true;
```

Controlling layer printing and visibility

You can control the printing and visibility of objects on a layer, as shown in the following script fragment. (For the complete script, see LayerControl.)

```
//Given a document "myDocument" containing layers named "Background,"
//"Language A,", "Language B," and "Language C," export the "Background"
//layer and each "Language" layer to PDF as separate PDF files...
var myVersion, myLanguageCounter, myFileName;
var myFolder = Folder.desktop;
for(var myCounter = 0; myCounter < 3; myCounter ++){
    switch(myCounter){
        case 0:
            myVersion = "Language A";
            break;
        case 1:
            myVersion = "Language B";
            break;
        case 2:
            myVersion = "Language C";
            break;
    }
    for(myLanguageCounter = 0; myLanguageCounter < myDocument.layers.length;
    myLanguageCounter ++){
        if((myDocument.layers.item(myLanguageCounter).name == myVersion)||
        (myDocument.layers.item(myLanguageCounter).name == "Background")){
            myDocument.layers.item(myLanguageCounter).visible = true;
            myDocument.layers.item(myLanguageCounter).printable = true;
        }
        else{
            myDocument.layers.item(myLanguageCounter).visible = false;
            myDocument.layers.item(myLanguageCounter).printable = false;
        }
    }
    myFileName = myFolder + "/" + myVersion + ".pdf";
    myDocument.exportFile(ExportFormat.pdfType, File(myFileName));
}
```

Locking layers

Layers can be locked, which means the page items on the layers cannot be edited. The following script fragment shows how to lock and unlock layers. (For the complete script, see LockLayersBelow.)

```
//Given a document "myDocument"...
var myTargetLayer = myDocument.activeLayer;
var myLayers = myDocument.layers.itemByRange(myDocument.layers.length-1,
myTargetLayer.index +1);
myLayers.locked = true;
```

5 Text and Type

Chapter Update Status		
CS6	Updated	Added section "Fitting text frames to content" on page 87.

Entering, editing, and formatting text are the tasks that make up the bulk of the time spent working on most InDesign documents. Because of this, automating text and type operations can result in large productivity gains.

This chapter shows how to script the most common operations involving text and type. The sample scripts in this chapter are presented in order of complexity, starting with very simple scripts and building toward more complex operations.

We assume that you have already read *Adobe InDesign Scripting Tutorial* and know how to create, install, and run a script. We also assume that you have some knowledge of working with text in InDesign and understand basic typesetting terms.

Entering and Importing Text

This section covers the process of getting text into your InDesign documents. Just as you can type text into text frames and place text files using the InDesign user interface, you can create text frames, insert text into a story, or place text files on pages using scripting.

Creating a text frame

The following script creates a text frame, sets the bounds (size) of the frame, then enters text in the frame (for the complete script, see the MakeTextFrame tutorial script):

```
var myDocument = app.documents.item(0);
var myPage = myDocument.pages.item(0);
    var myTextFrame = myPage.textFrames.add();
//Set the bounds of the text frame.
myTextFrame.geometricBounds = [72, 72, 288, 288];
//Enter text in the text frame.
myTextFrame.contents = "This is some example text."
//Note that you could also use a properties record to
//create the frame and set its bounds and contents in one line:
//var myTextFrame = myDocument.pages.item(0).textFrames.add(
{geometricBounds:[72, 72, 288, 288], contents:"This is some example text."});
```

The following script shows how to create a text frame that is the size of the area defined by the page margins. `myGetBounds` is a useful function that you can add to your own scripts, and it appears in many other examples in this chapter. (For the complete script, see MakeTextFrameWithinMargins.)

```
var myDocument = app.documents.item(0);
    var myPage = myDocument.pages.item(0);
//Create a text frame on the current page.
var myTextFrame = myPage.textFrames.add();
//Set the bounds of the text frame.
myTextFrame.geometricBounds = myGetBounds(myDocument, myPage);
//Enter text in the text frame.
myTextFrame.contents = "This is some example text."
```

The following script fragment shows the myGetBounds function.

```
function myGetBounds(myDocument, myPage){
    var myPageWidth = myDocument.documentPreferences.pageWidth;
    var myPageHeight = myDocument.documentPreferences.pageHeight
    if(myPage.side == PageSideOptions.leftHand){
        var myX2 = myPage.marginPreferences.left;
        var myX1 = myPage.marginPreferences.right;
    }
    else{
        var myX1 = myPage.marginPreferences.left;
        var myX2 = myPage.marginPreferences.right;
    }
    var myY1 = myPage.marginPreferences.top;
    var myX2 = myPageWidth - myX2;
    var myY2 = myPageHeight - myPage.marginPreferences.bottom;
    return [myY1, myX1, myY2, myX2];
}
```

Adding text

To add text to a story, use the `contents` property of the insertion point at the location where you want to insert the text. The following sample script uses this technique to add text at the end of a story (for the complete script, see AddText):

```
//Add text at the end of the text in the text frame.
//To do this, we'll use the last insertion point in the story.
//("\r" is a return character.)
var myNewText = "\rThis is a new paragraph of example text.";
myTextFrame.parentStory.insertionPoints.item(-1).contents = myNewText;
```

Stories and text frames

All text in an InDesign layout is part of a story, and every story can contain one or more text frames. Creating a text frame creates a story, and stories can contain multiple text frames.

In the preceding script, we added text at the end of the parent story rather than at the end of the text frame. This is because the end of the text frame might not be the end of the story; that depends on the length and formatting of the text. By adding the text to the end of the parent story, we can guarantee that the text is added, regardless of the composition of the text in the text frame.

You always can get a reference to the story using the `parentTextFrame` property of a text frame. It can be useful to work with the text of a story instead of the text of a text frame; the following script demonstrates the difference. The alerts shows that the text frame does not contain the overset text, but the story does (for the complete script, see StoryAndTextFrame).

```
var myDocument = app.activeDocument;
//Set the measurement units to points.
myDocument.viewPreferences.horizontalMeasurementUnits = MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
//Create a text frame on the current page.
var myTextFrame = app.activeWindow.activePage.textFrames.add();
//Set the bounds of the text frame.
myTextFrame.geometricBounds = [72, 72, 96, 288];
//Fill the text frame with placeholder text.
myTextFrame.contents = TextFrameContents.placeholderText;
//Now add text beyond the end of the text frame.
myTextFrame.insertionPoints.item(-1).contents = "\rThis is some overset text";
alert("The last paragraph in this alert should be \"This is some overset text\". Is
it?\r" + myTextFrame.contents);
alert("The last paragraph in this alert should be \"This is some overset text\". Is
it?\r" + myTextFrame.parentStory.contents);
```

For more on understanding the relationships between text objects in an InDesign document, see "Understanding Text Objects" on page 78.

Replacing text

The following script replaces a word with a phrase by changing the contents of the appropriate object (for the complete script, see ReplaceWord):

```
var myDocument = app.activeDocument;
//Set the measurement units to points.
myDocument.viewPreferences.horizontalMeasurementUnits = MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
//Create a text frame on the current page.
var myTextFrame = app.activeWindow.activePage.textFrames.add({geometricBounds:[72, 72,
288, 288], contents:"This is some example text."});
//Replace the third word "some" with the phrase
//"a little bit of".
myTextFrame.parentStory.words.item(2).contents = "a little bit of";
```

The following script replaces the text in a paragraph (for the complete script, see ReplaceText):

```
//Replace the text in the second paragraph without replacing
//the return character at the end of the paragraph. To do this,
//we'll use the ItemByRange method.
var myStartCharacter = myTextFrame.parentStory.paragraphs.item(1).characters.item(0);
var myEndCharacter = myTextFrame.parentStory.paragraphs.item(1).characters.item(-2);
myTextFrame.texts.itemByRange(myStartCharacter, myEndCharacter).contents = "This text
replaces the text in paragraph 2."
```

In the preceding script above, we excluded the return character because deleting the return might change the paragraph style applied to the paragraph. To do this, we used `ItemByRange` method, and we supplied two characters—the starting and ending characters of the paragraph—as parameters.

Inserting special characters

Because the ExtendScript Toolkit supports Unicode, you can simply enter Unicode characters in text strings that you send to InDesign. Alternately, you can use the JavaScript method of explicitly entering Unicode characters by their glyph ID number: `\u`*nnnn* (where *nnnn* is the Unicode code for the character). The following script shows several ways to enter special characters. (We omitted the `myGetBounds`

function from this listing; you can find it in "Creating a text frame" on page 68" or in the SpecialCharacters tutorial script.)

```
var myDocument = app.documents.item(0);
//Create a text frame on the current page.
var myTextFrame = myDocument.pages.item(0).textFrames.add();
//Set the bounds of the text frame.
myTextFrame.geometricBounds = myGetBounds(myDocument, myDocument.pages.item(0));
//Entering special characters directly.
myTextFrame.contents = "Registered trademark: Æ\rCopyright: ©\rTrademark: ?\r";
//Entering special characters by their Unicode glyph ID value:
myTextFrame.parentStory.insertionPoints.item(-1).contents = "Not equal to:
\u2260\rSquare root: \u221A\rParagraph: \u00B6\r";
//Entering InDesign special characters by their enumerations:
myTextFrame.parentStory.insertionPoints.item(-1).contents = "Automatic page number
marker:";
myTextFrame.parentStory.insertionPoints.item(-1).contents =
SpecialCharacters.autoPageNumber;
myTextFrame.parentStory.insertionPoints.item(-1).contents = "\r";
myTextFrame.parentStory.insertionPoints.item(-1).contents = "Section symbol:";
myTextFrame.parentStory.insertionPoints.item(-1).contents =
SpecialCharacters.sectionSymbol;
myTextFrame.parentStory.insertionPoints.item(-1).contents = "\r";
myTextFrame.parentStory.insertionPoints.item(-1).contents = "En dash:";
myTextFrame.parentStory.insertionPoints.item(-1).contents = SpecialCharacters.enDash;
myTextFrame.parentStory.insertionPoints.item(-1).contents = "\r";
```

The easiest way to find the Unicode ID for a character is to use InDesign's Glyphs palette: move the cursor over a character in the palette, and InDesign displays its Unicode value. To learn more about Unicode, visit http://www.unicode.org.

Placing Text and Setting Text-Import Preferences

In addition to entering text strings, you can place text files created using word processors and text editors. The following script shows how to place a text file on a document page (for the complete script, see PlaceTextFile):

```
var myDocument = app.documents.item(0);
//Set the measurement units to points.
myDocument.viewPreferences.horizontalMeasurementUnits = MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
//Get the current page.
var myPage = myDocument.pages.item(0);
//Get the top and left margins to use as a place point.
var myX = myPage.marginPreferences.left;
var myY = myPage.marginPreferences.top;
//Autoflow a text file on the current page.
//Parameters for Page.place():
//File as File object,
//[PlacePoint as Array [x, y]]
//[DestinationLayer as Layer object]
//[ShowingOptions as Boolean = False]
//[Autoflowing as Boolean = False]
//You'll have to fill in your own file path.
var myStory = myPage.place(File("/c/test.txt"), [myX, myY], undefined, false, true)[0];
//Note that if the PlacePoint parameter is inside a column, only the vertical (y)
//coordinate will be honored--the text frame will expand horizontally to fit the
column.
```

The following script shows how to place a text file in an existing text frame. (We omitted the `myGetBounds` function from this listing; you can find it in "Creating a text frame" on page 68," or see the PlaceTextFileInFrame tutorial script.)

```
var myDocument = app.documents.item(0);
var myPage = myDocument.pages.item(0);
var myTextFrame =
myPage.textFrames.add({geometricBounds:myGetBounds(myDocument,myPage)});
//Place a text file in the text frame.
//Parameters for TextFrame.place():
//File as File object,
//[ShowingOptions as Boolean = False]
//You'll have to fill in your own file path.
myTextFrame.place(File("/c/test.txt"));
```

The following script shows how to insert a text file at a specific location in text. (We omitted the `myGetBounds` function from this listing; you can find it in "Creating a text frame" on page 68," or see the InsertTextFile tutorial script.)

```
var myDocument = app.documents.item(0);
    var myPage = myDocument.pages.item(0);
    var myTextFrame = myPage.textFrames.item(0);
//Place a text file at the end of the text.
//Parameters for InsertionPoint.place():
//File as File object,
//[ShowingOptions as Boolean = False]
//You'll have to fill in your own file path.
myTextFrame.parentStory.insertionPoints.item(-1).place(File("/c/test.txt"));
```

To specify the import options for the specific type of text file you are placing, use the corresponding import-preferences object. The following script shows how to set text-import preferences (for the complete script, see TextImportPreferences). The comments in the script show the possible values for each property.

```
with(app.textImportPreferences){
    //Options for characterSet: TextImportCharacterSet.
    characterSet = TextImportCharacterSet.UTF8;
    convertSpacesIntoTabs = true;
    spacesIntoTabsCount = 3;
    //The dictionary property can take many values, such as French, Italian.
    dictionary = "English: USA";
    //platform options: ImportPlatform
    platform = ImportPlatform.macintosh;
    stripReturnsBetweenLines = true;
    stripReturnsBetweenParagraphs = true;
    useTypographersQuotes = true;
}
```

The following script shows how to set tagged text import preferences (for the complete script, see TaggedTextImportPreferences):

```
with(app.taggedTextImportPreferences){
    removeTextFormatting = false;
    //styleConflict property can be:
    //StyleConflict.publicationDefinition
    //StyleConflict.tagFileDefinition
    styleConflict = StyleConflict.publicationDefinition;
    useTypographersQuotes = true;
}
```

The following script shows how to set Word and RTF import preferences (for the complete script, see WordRTFImportPreferences):

```
with(app.wordRTFImportPreferences){
    //convertPageBreaks property can be:
    //ConvertPageBreaks.columnBreak
    //ConvertPageBreaks.none
    //ConvertPageBreaks.pageBreak
    convertPageBreaks = ConvertPageBreaks.none;
    //convertTablesTo property can be:
    //ConvertTablesOptions.unformattedTabbedText
    //ConvertTablesOptions.unformattedTable
    convertTablesTo = ConvertTablesOptions.unformattedTable;
    importEndnotes = true;
    importFootnotes = true;
    importIndex = true;
    importTOC = true;
    importUnusedStyles = false;
    preserveGraphics = false;
    preserveLocalOverrides = false;
    preserveTrackChanges = false;
    removeFormatting = false;
    //resolveCharacterSytleClash and resolveParagraphStyleClash properties can be:
    //ResolveStyleClash.resolveClashAutoRename
    //ResolveStyleClash.resolveClashUseExisting
    //ResolveStyleClash.resolveClashUseNew
    resolveCharacterStyleClash = ResolveStyleClash.resolveClashUseExisting;
    resolveParagraphStyleClash = ResolveStyleClash.resolveClashUseExisting;
    useTypographersQuotes = true;
}
```

The following script shows how to set Excel import preferences (for the complete script, see ExcelImportPreferences):

```
with(app.excelImportPreferences){
    //alignmentStyle property can be:
    //AlignmentStyleOptions.centerAlign
    //AlignmentStyleOptions.leftAlign
    //AlignmentStyleOptions.rightAlign
    //AlignmentStyleOptions.spreadsheet
    alignmentStyle = AlignmentStyleOptions.spreadsheet;
    decimalPlaces = 4;
    preserveGraphics = false;
    //Enter the range you want to import as "start cell:end cell".
    rangeName = "A1:B16";
    sheetIndex = 1;
    sheetName = "pathpoints";
    showHiddenCells = false;
    //tableFormatting property can be:
    //TableFormattingOptions.excelFormattedTable
    //TableFormattingOptions.excelUnformattedTabbedText
    //TableFormattingOptions.excelUnformattedTable
    tableFormatting = TableFormattingOptions.excelFormattedTable;
    useTypographersQuotes = true;
    viewName = "";
}
```

Exporting Text and Setting Text-Export Preferences

The following script shows how to export text from an InDesign document. Note that you must use text or story objects to export into text file formats; you cannot export all text in a document in one operation. (We omitted the `myGetBounds` function from this listing; you can find it in "Creating a text frame" on page 68," or see the ExportTextFile tutorial script.)

```
var myDocument = app.documents.item(0);
var myPage = myDocument.pages.item(0);
var myTextFrame = myPage.textFrames.item(0);
//Text exportFile method parameters:
//Format as ExportFormat
//To As File
//[ShowingOptions As Boolean = False]
//Format parameter can be:
//ExportFormat.inCopy
//ExportFormat.inCopyCS2Story
//ExportFormat.rtf
//ExportFormat.taggedText
//ExportFormat.textType
//Export the story as text. You'll have to fill in a valid file path on your system.
myTextFrame.parentStory.exportFile(ExportFormat.textType, File("/c/test.txt"));
```

The following example shows how to export a specific range of text. (We omitted the `myGetBounds` function from this listing; you can find it in "Creating a text frame" on page 68," or see the ExportTextRange tutorial script.)

```
var myDocument = app.documents.item(0);
var myStory = myDocument.stories.item(0);
    var myStart = myStory.characters.item(0);
    var myEnd = myStory.paragraphs.item(0).characters.item(-1);
myText = myStory.texts.itemByRange(myStart, myEnd);
//Text exportFile method parameters:
//Format as ExportFormat
//To As File
//[ShowingOptions As Boolean = False]
//Format parameter can be:
//ExportFormat.inCopy
//ExportFormat.inCopyCS2Story
//ExportFormat.rtf
//ExportFormat.taggedText
//ExportFormat.textType
//Export the text range. You'll have to fill in a valid file path on your system.
myText.exportFile(ExportFormat.textType, File("/c/test.txt"));
```

To specify the export options for the specific type of text file you're exporting, use the corresponding export preferences object. The following script sets text-export preferences (for the complete script, see TextExportPreferences):

```
with(app.textExportPreferences){
    //Options for characterSet: TextExportCharacterSet
    characterSet = TextExportCharacterSet.UTF8;
    //platform options: ImportPlatform
    platform = ImportPlatform.macintosh;
}
```

The following script sets tagged text export preferences (for the complete script, see TaggedTextExportPreferences):

```
with(app.taggedTextExportPreferences){
    //Options for characterSet:
    //TagTextExportCharacterSet.ansi
    //TagTextExportCharacterSet.ascii
    //TagTextExportCharacterSet.gb18030
    //TagTextExportCharacterSet.ksc5601
    //TagTextExportCharacterSet.shiftJIS
    //TagTextExportCharacterSet.unicode
    characterSet = TagTextExportCharacterSet.unicode;
    //tagForm options:
    //TagTextForm.abbreviated
    //TagTextForm.verbose
    tagForm = TagTextForm.verbose;
}
```

You cannot export all text in a document in one step. Instead, you need to either combine the text in the document into a single story and then export that story, or combine the text files by reading and writing files via scripting. The following script demonstrates the former approach. (We omitted the `myGetBounds` function from this listing; you can find it in "Creating a text frame" on page 68," or see the ExportAllText tutorial script.) For any format other than text only, the latter method can become quite complex.

```
if(app.documents.length != 0){
    if(app.documents.item(0).stories.length != 0){
        myExportAllText(app.documents.item(0).name);
    }
}
```

Here is the ExportAllText function referred to in the preceding fragment:

```
function myExportAllText(myDocumentName){
    var myStory;
    //File name for the exported text. Fill in a valid file path on your system.
    var myFileName = "/c/test.txt";
    //If you want to add a separator line between stories, set myAddSeparator to true.
    var myAddSeparator = true;
    var myNewDocument = app.documents.add();
    var myDocument = app.documents.item(myDocumentName);
    var myTextFrame = myNewDocument.pages.item(0).textFrames.add(
    {geometricBounds:myGetBounds(myNewDocument, myNewDocument.pages.item(0))});
    var myNewStory = myTextFrame.parentStory;
    for(myCounter = 0; myCounter < myDocument.stories.length; myCounter++){
        myStory = myDocument.stories.item(myCounter);
        //Export the story as tagged text.
        myStory.exportFile(ExportFormat.taggedText, File(myFileName));
        //Import (place) the file at the end of the temporary story.
        myNewStory.insertionPoints.item(-1).place(File(myFileName));
```

```
        //If the imported text did not end with a return, enter a return
        //to keep the stories from running together.
        if(myCounter != myDocument.stories.length -1){
            if(myNewStory.characters.item(-1).contents != "\r"){
                myNewStory.insertionPoints.item(-1).contents = "\r";
            }
            if(myAddSeparator == true){
                myNewStory.insertionPoints.item(-1).contents =
                "----------------------------------------\r";
            }
        }
    }
    myNewStory.exportFile(ExportFormat.taggedText, File("/c/test.txt"));
    myNewDocument.close(SaveOptions.no);
}
```

Do not assume that you are limited to exporting text using existing export filters. Because JavaScript can write text files to disk, you can have your script traverse the text in a document and export it in any order you like, using whatever text mark-up scheme you prefer. Here is a very simple example that shows how to export InDesign text as HTML. (We omitted the `myGetBounds` function from this listing; you can find it in "Creating a text frame" on page 68," or see the ExportHTML tutorial script.)

```
var myStory, myParagraph, myString, myTag, myStartTag;
var myEndTag, myTextStyleRange, myTable;
//Use the myStyleToTagMapping array to set up your paragraph style to tag mapping.
var myStyleToTagMapping = new Array;
//For each style to tag mapping, add a new item to the array.
myStyleToTagMapping.push(["body_text", "p"]);
myStyleToTagMapping.push(["heading1", "h1"]);
myStyleToTagMapping.push(["heading2", "h2"]);
myStyleToTagMapping.push(["heading3", "h3"]);
//End of style to tag mapping.
if(app.documents.length !=0){
    if(app.documents.item(0).stories.length != 0){
        //Open a new text file.
        var myTextFile = File.saveDialog("Save HTML As", undefined);
        //If the user clicked the Cancel button, the result is null.
        if(myTextFile != null){
            //Open the file with write access.
            myTextFile.open("w");
            //Iterate through the stories.
            for(var myCounter = 0; myCounter < app.documents.item(0).stories.length;
            myCounter ++){
                myStory = app.documents.item(0).stories.item(myCounter);
                for(var myParagraphCounter = 0; myParagraphCounter <
                myStory.paragraphs.length; myParagraphCounter ++){
                    myParagraph = myStory.paragraphs.item(myParagraphCounter);
                    if(myParagraph.tables.length == 0){
                        if(myParagraph.textStyleRanges.length == 1){
                            //If the paragraph is a simple paragraph--no tables, no local
                            //formatting--then simply export the text of the pararaph with
                            //the appropriate tag.
                            myTag = myFindTag(myParagraph.appliedParagraphStyle.name,
                            myStyleToTagMapping);
                            //If the tag comes back empty, map it to the
                            //basic paragraph tag.
                            if(myTag == ""){
                                myTag = "p";
                            }
                            myStartTag = "<" + myTag + ">";
```

```
            myEndTag = "</" + myTag + ">";
            //If the paragraph is not the last paragraph in the story,
            //omit the return character.
            if(myParagraph.characters.item(-1).contents == "\r"){
                myString = myParagraph.texts.itemByRange(
                myParagraph.characters.item(0),myParagraph.
                characters.item(-2)).contents;
            }
            else{
                myString = myParagraph.contents;
            }
            //Write the paragraphs' text to the text file.
            myTextFile.writeln(myStartTag + myString + myEndTag);
        }
        else{
            //Handle text style range export by iterating through the text
            //style ranges in the paragraph..
            for(var myRangeCounter = 0; myRangeCounter <
            myParagraph.textStyleRanges.length; myRangeCounter ++){
                myTextStyleRange = myParagraph.textStyleRanges.item
                (myRangeCounter);
                if(myTextStyleRange.characters.item(-1)=="\r"){
                    myString = myTextStyleRange.texts.itemByRange(
                    myTextStyleRange.characters.item(1),
                    myTextStyleRange.characters.item(-2)).contents;
                }
                else{
                    myString = myTextStyleRange.contents;
                }
                switch(myTextStyleRange.fontStyle){
                    case "Bold":
                        myString = "<b>" + myString + "</b>"
                        break;
                    case "Italic":
                        myString = "<i>" + myString + "</i>"
                        break;
                }
                myTextFile.write(myString);
            }
            myTextFile.write("\r");
        }
    }
    else{
        //Handle table export (assumes that there is only one table per
        //paragraph, and that the table is in the paragraph by itself).
        myTable = myParagraph.tables.item(0);
        myTextFile.writeln("<table border = 1>");
        for(var myRowCounter = 0; myRowCounter < myTable.rows.length;
        myRowCounter ++){
            myTextFile.writeln("<tr>");
            for(var myColumnCounter = 0; myColumnCounter <
            myTable.columns.length; myColumnCounter++){
                if(myRowCounter == 0){
                    myString = "<th>" +
                    myTable.rows.item(myRowCounter).cells.
                    item(myColumnCounter).texts.item(0).contents + "</th>";
                }
                else{
                    myString = "<td>" + myTable.rows.item(myRowCounter).
                    cells.item(myColumnCounter).texts.item(0).contents +
```

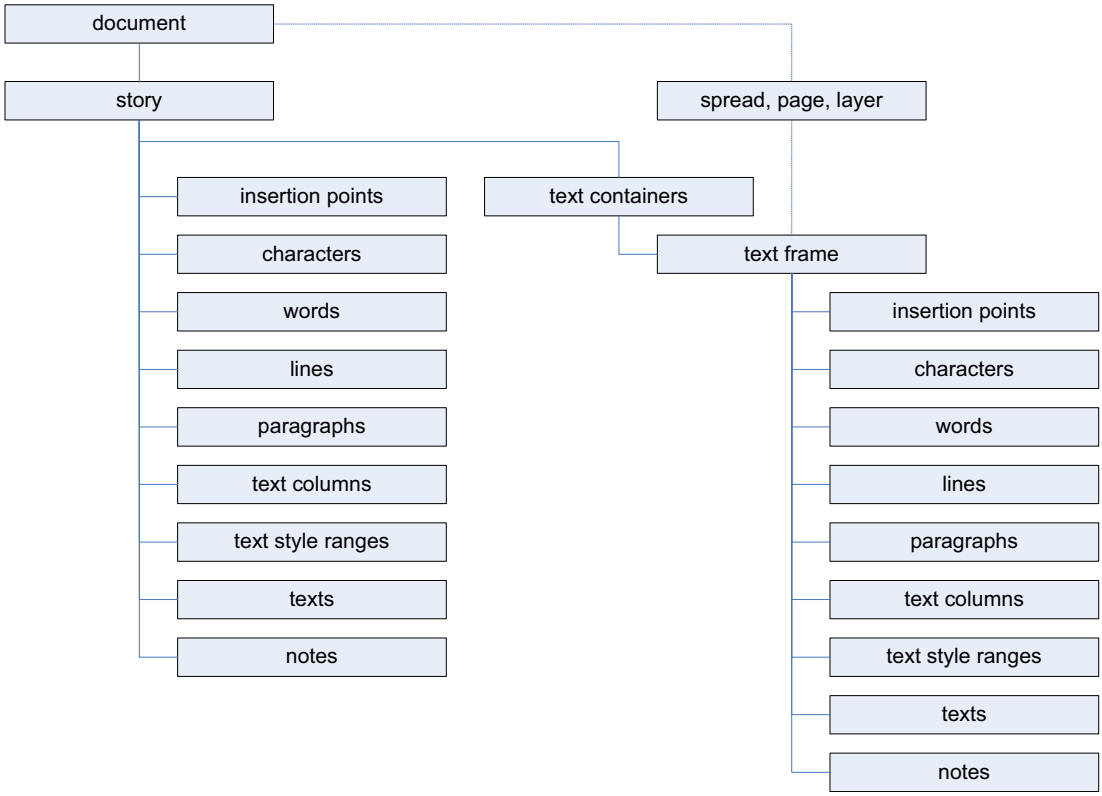
```
                                "</td>";
                            }
                            myTextFile.writeln(myString);
                        }
                        myTextFile.writeln("</tr>");
                    }
                    myTextFile.writeln("</table>");
                }
            }
        }
        //Close the text file.
        myTextFile.close();
    }
}
```

Here is the `myFindTag` function referred to in the above script:

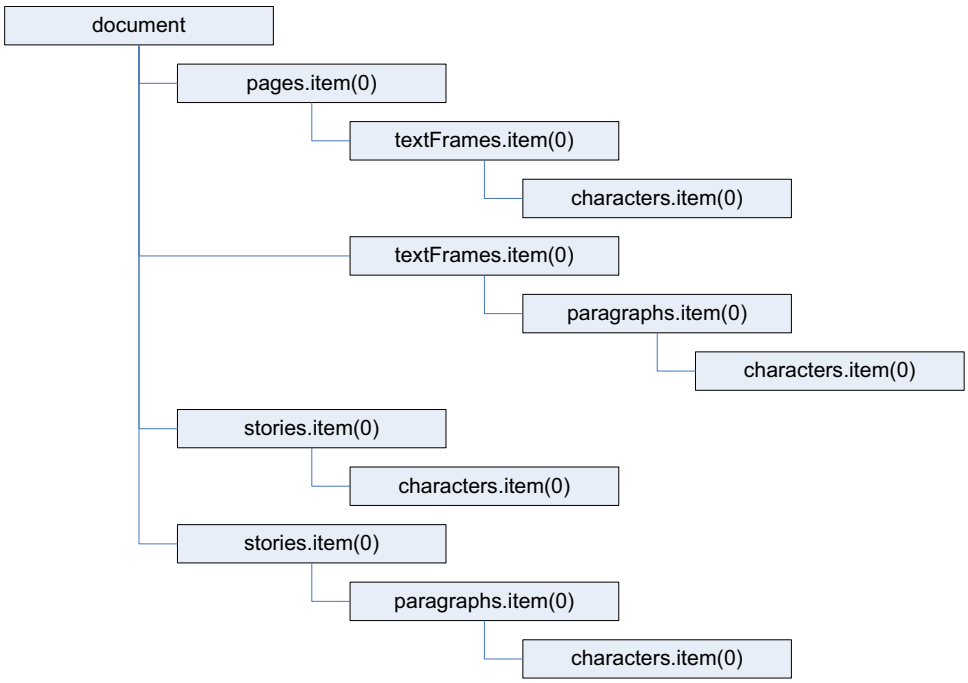
```
function myFindTag (myStyleName, myStyleToTagMapping){
    var myTag = "";
    var myDone = false;
    var myCounter = 0;
    do{
        if(myStyleToTagMapping[myCounter][0] == myStyleName){
            myTag = myStyleToTagMapping[myCounter][1];
            break;
        }
        myCounter ++;
    } while((myDone == false)||(myCounter < myStyleToTagMapping.length))
    return myTag;
}
```

Understanding Text Objects

The following diagram shows a view of InDesign's text object model. As you can see, there are two main types of text object: *layout* objects (text frames) and *text-stream* objects (for example, stories, insertion points, characters, and words):



There are many ways to get a reference to a given text object. The following diagram shows a few ways to refer to the first character in the first text frame of the first page of a new document:



For any text stream object, the `parent` of the object is the story containing the object. To get a reference to the text frame (or text frames) containing the text object, use the `parentTextFrames` property.

For a text frame, the `parent` of the text frame usually is the page or spread containing the text frame. If the text frame is inside a group or was pasted inside another page item, the `parent` of the text frame is the containing page item. If the text frame was converted to an anchored frame, the `parent` of the text frame is the character containing the anchored frame.

Working with text selections

Text-related scripts often act on a text selection. The following script demonstrates a way to determine whether the current selection is a text selection. Unlike many of the other sample scripts, this script does not actually do anything; it simply presents a selection-filtering routine that you can use in your own scripts (for the complete script, see TextSelection).

```
if (app.documents.length != 0){
    //If the selection contains more than one item, the selection
    //is not text selected with the Type tool.
    if (app.selection.length == 1){
        //Evaluate the selection based on its type.
        switch (app.selection[0].constructor.name){
            case "InsertionPoint":
            case "Character":
            case "Word":
            case "TextStyleRange":
            case "Line":
            case "Paragraph":
            case "TextColumn":
            case "Text":
            case "Story":
                //The object is a text object; pass it on to a function.
                myProcessText(app.selection[0]);
                break;
            //In addition to checking for the above text objects, we can
            //also continue if the selection is a text frame selected with
            //the Selection tool or the Direct Selection tool.
            case "TextFrame":
                //If the selection is a text frame, get a reference to the
                //text in the text frame.
                myProcessText(app.selection[0].texts.item(0));
                break;
            default:
                alert("The selected object is not a text object.
                Select some text and try again.");
                break;
        }
    }
    else{
        alert("Please select some text and try again.");
    }
}
```

Moving and copying text

You can move a text object to another location in text using the `move` method. To copy the text, use the `duplicate` method (whose arguments are identical to the `move` method). The following script fragment shows how it works (for the complete script, see MoveText):

```
var myDocument = app.documents.item(0);
var myPage = myDocument.pages.item(0);
    var myTextFrameA = myPage.textFrames.item(3);
    var myTextFrameB = myPage.textFrames.item(2);
    var myTextFrameC = myPage.textFrames.item(1);
    var myTextFrameD = myPage.textFrames.item(0);
//Move WordC between the words in TextFrameC.
myTextFrameD.parentStory.paragraphs.item(-1).words.item(0).move(LocationOptions.befor
e, myTextFrameC.parentStory.paragraphs.item(0).words.item(1))
//Move WordB after the word in TextFrameB.
myTextFrameD.parentStory.paragraphs.item(-2).words.item(0).move(LocationOptions.after
, myTextFrameB.parentStory.paragraphs.item(0).words.item(0))
//Move WordA to before the word in TextFrameA.
myTextFrameD.parentStory.paragraphs.item(-3).words.item(0).move(LocationOptions.befor
e, myTextFrameA.parentStory.paragraphs.item(0).words.item(0))
//Note that moving text removes it from its original location.
```

When you want to transfer formatted text from one document to another, you also can use the `move` method. Using the `move` or `duplicate` method is better than using copy and paste; to use copy and paste, you must make the document visible and select the text you want to copy. Using `move` or `duplicate` is much faster and more robust. The following script shows how to move text from one document to another using `move` and `duplicate`. (We omitted the `myGetBounds` function from this listing; you can find it in "Creating a text frame" on page 68," or see the MoveTextBetweenDocuments tutorial script.)

```
//Create the source document.
var mySourceDocument = app.documents.add();
var mySourcePage = mySourceDocument.pages.item(0);
var mySourceTextFrame =
mySourcePage.textFrames.add({geometricBounds:myGetBounds(mySourceDocument,
mySourcePage), contents:"This is the source text.\rThis text is not the source
text."});
var mySoureParagraph = mySourceTextFrame.parentStory.paragraphs.item(0);
mySoureParagraph.pointSize = 24;
//Create the target document.
var myTargetDocument = app.documents.add();
var myTargetPage = myTargetDocument.pages.item(0);
var myTargetTextFrame =
myTargetPage.textFrames.add({geometricBounds:myGetBounds(myTargetDocument,
myTargetDocument.pages.item(0)), contents:"This is the target text. Insert the source
text before this paragraph.\r"});
//Move the text from the source document to the target document.
//This deletes the text from the source document.
mySoureParagraph.move(LocationOptions.AT_BEGINNING,
myTargetTextFrame.insertionPoints.item(0));
//To duplicate (rather than move) the text, use the following:
//mySoureParagraph.duplicate(LocationOptions.AT_BEGINNING,
myTargetTextFrame.insertionPoints.item(0));
```

When you need to copy and paste text, you can use the `copy` method of the application. You will need to select the text before you copy. Again, you should use copy and paste only as a last resort; other approaches are faster, less fragile, and do not depend on the document being visible. (We omitted the `myGetBounds` function from this listing; you can find it in "Creating a text frame" on page 68," or see the CopyPasteText tutorial script.)

```
var myDocumentA = app.documents.add();
var myPageA = myDocumentA.pages.item(0);
var myString = "Example text.\r";
var myTextFrameA = myPageA.textFrames.add({geometricBounds:myGetBounds(myDocumentA,
myPageA), contents:myString});
var myDocumentB = app.documents.add();
var myPageB = myDocumentB.pages.item(0);
var myTextFrameB = myPageB.textFrames.add({geometricBounds:myGetBounds(myDocumentB,
myPageB)});
//Make document A the active document.
app.activeDocument = myDocumentA;
//Select the text.
app.select(myTextFrameA.parentStory.texts.item(0));
app.copy();
//Make document B the active document.
app.activeDocument = myDocumentB;
//Select the insertion point at which you want to paste the text.
app.select(myTextFrameB.parentStory.insertionPoints.item(0));
app.paste();
```

One way to copy unformatted text from one text object to another is to get the `contents` property of a text object, then use that string to set the `contents` property of another text object. The following script shows how to do this (for the complete script, see CopyUnformattedText):

```
var myDocument = app.documents.item(0);
var myPage = myDocument.pages.item(0);
//Create a text frame on the active page.
var myTextFrameA = myPage.textFrames.add({geometricBounds:[72, 72, 144, 288]});
myTextFrameA.contents = "This is a formatted string.";
myTextFrameA.parentStory.texts.item(0).fontStyle = "Bold";
//Create another text frame on the active page.
var myTextFrameB = myPage.textFrames.add({geometricBounds:[228, 72, 300, 288]});
myTextFrameB.contents = "This is the destination text frame. Text pasted here will
retain its formatting.";
myTextFrameB.parentStory.texts.item(0).fontStyle = "Italic";
//Copy from one frame to another using a simple copy.
app.select(myTextFrameA.texts.item(0));
app.copy();
app.select(myTextFrameB.parentStory.insertionPoints.item(-1));
app.paste();
//Create another text frame on the active page.
var myTextFrameC = myPage.textFrames.add({geometricBounds:[312, 72, 444, 288]});
myTextFrameC.contents = "Text copied here will take on the formatting of the existing
text.";
myTextFrameC.parentStory.texts.item(0).fontStyle = "Italic";
//Copy the unformatted string from text frame A to the end of text frame C (note
//that this doesn't really copy the text; it replicates the text string from one
//text frame in another text frame):
myTextFrameC.parentStory.insertionPoints.item(-1).contents =
myTextFrameA.parentStory.texts.item(0).contents;
```

Text objects and iteration

When your script moves, deletes, or adds text while iterating through a series of text objects, you can easily end up with invalid text references. The following script demonstrates this problem. (We omitted the `myGetBounds` function from this listing; you can find it in "Creating a text frame" on page 68, or see the TextIterationWrong tutorial script.)

```
var myDocument = app.documents.item(0);
var myStory = myDocument.stories.item(0);
//The following for loop will fail to format all of the paragraphs.
for(var myParagraphCounter = 0; myParagraphCounter < myStory.paragraphs.length;
myParagraphCounter ++){
    if(myStory.paragraphs.item(myParagraphCounter).words.item(0).contents=="Delete"){
        myStory.paragraphs.item(myParagraphCounter).remove();
    }
    else{
        myStory.paragraphs.item(myParagraphCounter).pointSize = 24;
    }
}
```

In the preceding example, some of the paragraphs are left unformatted. How does this happen? The loop in the script iterates through the paragraphs from the first paragraph in the story to the last. As it does so, it deletes paragraphs that begin with the word "Delete." When the script deletes the second paragraph, the third paragraph moves up to take its place. When the loop counter reaches 2, the script processes the paragraph that had been the fourth paragraph in the story; the original third paragraph is now the second paragraph and is skipped.

To avoid this problem, iterate backward through the text objects, as shown in the following script. (We omitted the `myGetBounds` function from this listing; you can find it in "Creating a text frame" on page 68," or see the TextIterationRight tutorial script.)

```
var myDocument = app.documents.item(0);
var myStory = myDocument.stories.item(0);
//The following for loop will format all of the paragraphs by iterating
//backwards through the paragraphs in the story.
for(var myParagraphCounter = myStory.paragraphs.length-1; myParagraphCounter >= 0;
myParagraphCounter --){
    if(myStory.paragraphs.item(myParagraphCounter).words.item(0).contents=="Delete"){
        myStory.paragraphs.item(myParagraphCounter).remove();
    }
    else{
        myStory.paragraphs.item(myParagraphCounter).pointSize = 24;
    }
}
```

Working with Text Frames

In the previous sections of this chapter, we concentrated on working with text stream objects; in this section, we focus on text frames, the page-layout items that contain text in an InDesign document.

Linking text frames

The `nextTextFrame` and `previousTextFrame` properties of a text frame are the keys to linking (or "threading") text frames in InDesign scripting. These properties correspond to the in port and out port on InDesign text frames, as shown in the following script fragment (for the complete script, see LinkTextFrames):

```
var myDocument = app.documents.item(0);
var myPage = myDocument.pages.item(0);
var myTextFrameA = myPage.textFrames.item(1);
var myTextFrameB = myPage.textFrames.item(0);
//Add a page.
var myNewPage = myDocument.pages.add();
//Create another text frame on the new page.
var myTextFrameC = myNewPage.textFrames.add({geometricBounds:[72, 72, 144, 144]})
//Link TextFrameA to TextFrameB using the nextTextFrame property.
myTextFrameA.nextTextFrame = myTextFrameB;
//Link TextFrameC to TextFrameB using the previousTextFrame property.
myTextFrameC.previousTextFrame = myTextFrameB;
//Fill the text frames with placeholder text.
myTextFrameA.contents = TextFrameContents.placeholderText;
```

Unlinking text frames

The following example script shows how to unlink text frames (for the complete script, see UnlinkTextFrames):

```
//Unlink the two text frames.
myTextFrameA.nextTextFrame = NothingEnum.nothing;
```

Removing a frame from a story

In InDesign, deleting a frame from a story does not delete the text in the frame, unless the frame is the only frame in the story. The following script fragment shows how to delete a frame and the text it contains from a story without disturbing the other frames in the story (for the complete script, see BreakFrame):

```
var myObjectList = new Array;
//Script does nothing if no documents are open or if no objects are selected.
if(app.documents.length != 0){
    if(app.selection.length != 0){
        //Process the objects in the selection to create a list of
        //qualifying objects (text frames).
        for(myCounter = 0; myCounter < app.selection.length; myCounter ++){
            switch(app.selection[myCounter].constructor.name){
                case "TextFrame":
                    myObjectList.push(app.selection[myCounter]);
                    break;
                default:
                    if(app.selection.length == 1){
                        //If text is selected, then get the parent text frame.
                        switch(app.selection[myCounter].constructor.name){
                            case "Text":
                            case "InsertionPoint":
                            case "Character":
                            case "Word":
                            case "Line":
                            case "TextStyleRange":
                            case "Paragraph":
                            case "TextColumn":
```

```
                                myObjectList.push(app.selection[myCounter].
                                parentTextFrames[0]);
                                break;
                                }
                        }
                        break;
                }
            }
            //If the object list is not empty, pass it on to the function
            //that does the real work.
            if(myObjectList.length != 0){
                myBreakFrames(myObjectList);
            }
        }
}
```

Here is the myBreakFrames function referred to in the preceding script.

```
function myBreakFrames(myObjectList){
    myObjectList.sort(myReverseSortByTextFrameIndex);
    for(var myCounter = 0; myCounter < myObjectList.length; myCounter ++){
        myBreakFrame(myObjectList[myCounter]);
    }
}
function myBreakFrame(myTextFrame){
    if((myTextFrame.nextTextFrame != null)&&(myTextFrame.previousTextFrame != null)){
        var myNewFrame = myTextFrame.duplicate();
        if(myTextFrame.contents != ""){
            myTextFrame.texts.item(0).remove();
        }
        myTextFrame.remove();
    }
}
function myReverseSortByTextFrameIndex(a,b){
    //By combining the story id with the text frame index, we can sort the text frames
    //into the right (reverse) order in a single pass.
    $.write("padded a: " + myPadString(a.id, 8)+myPadString(a.textFrameIndex, 8));
    $.write("padded b: " + myPadString(b.id, 8)+myPadString(b.textFrameIndex, 8));
    if((myPadString(a.id, 8)+myPadString(a.textFrameIndex, 8)) >
    (myPadString(b.id, 8)+myPadString(b.textFrameIndex, 8))){
        return -1;
    }
    if((myPadString(a.id,8)+myPadString(a.textFrameIndex,8)) <
    (myPadString(b.id,8)+myPadString(b.textFrameIndex,8))){
        return 1;
    }
    return 0;
}
function myPadString(myString, myLength) {
  var myTempString = "";
  var myNewLength = myLength-String(myString).length;
  for (var myCounter = 0; myCounter<myNewLength; myCounter++) {
    myTempString += "0";
  }
  return myTempString + myString;
}
```

Splitting all frames in a story

The following script fragment shows how to split all frames in a story into separate, independent stories, each containing one unlinked text frame (for the complete script, see SplitStory):

```
if(app.documents.length != 0){
    if(app.selection.length != 0){
        //Get the first item in the selection.
        var mySelection = app.selection[0];
        //Process the selection. If text or a text frame is
        //selected, do something; otherwise, do nothing.
        switch(mySelection.constructor.name){
            case "Text":
            case "InsertionPoint":
            case "Character":
            case "Word":
            case "Line":
            case "TextStyleRange":
            case "Paragraph":
            case "TextColumn":
            case "TextFrame":
                //If the text frame is the only text frame in the story, do nothing.
                if(mySelection.parentStory.textContainers.length != 1){
                    //Splitting the story is a two-step process: first, duplicate
                    //the text frames, second, delete the original text frames.
                    mySplitStory(mySelection.parentStory);
                    myRemoveFrames(mySelection.parentStory);
                }
                break;
        }
    }
}
```

Here is the `mySplitStory` function referred to in the preceding script:

```
function mySplitStory(myStory){
    var myTextFrame;
    //Duplicate each text frame in the story.
    for(var myCounter = myStory.textContainers.length-1; myCounter >= 0;
    myCounter --){
        myTextFrame = myStory.textContainers[myCounter];
         myTextFrame.duplicate();
    }
}
function myRemoveFrames(myStory){
    //Remove each text frame in the story. Iterate backwards to
    //avoid invalid references.
    for(var myCounter = myStory.textContainers.length-1; myCounter >= 0; myCounter --){
        myStory.textContainers[myCounter].remove();
    }
}
```

Creating an anchored frame

To create an anchored frame (also known as an inline frame), you can create a text frame (or rectangle, oval, polygon, or graphic line) at a specific location in text (usually an insertion point). The following script fragment shows an example (for the complete script, see AnchoredFrame):

```
var myInsertionPoint = myTextFrame.paragraphs.item(0).insertionPoints.item(0);
var myInlineFrame = myInsertionPoint.textFrames.add();
//Recompose the text to make sure that getting the
//geometric bounds of the inline graphic will work.
myTextFrame.texts.item(0).recompose;
//Get the geometric bounds of the inline frame.
var myBounds = myInlineFrame.geometricBounds;
//Set the width and height of the inline frame. In this example, we'll
//make the frame 24 points tall by 72 points wide.
var myArray = [myBounds[0], myBounds[1], myBounds[0]+24, myBounds[1]+72];
myInlineFrame.geometricBounds = myArray;
myInlineFrame.contents = "This is an inline frame.";
myInsertionPoint = myTextFrame.paragraphs.item(1).insertionPoints.item(0);
var myAnchoredFrame = myInsertionPoint.textFrames.add();
//Recompose the text to make sure that getting the
//geometric bounds of the inline graphic will work.
myTextFrame.texts.item(0).recompose;
//Get the geometric bounds of the inline frame.
var myBounds = myAnchoredFrame.geometricBounds;
//Set the width and height of the inline frame. In this example, we'll
//make the frame 24 points tall by 72 points wide.
myArray = [myBounds[0], myBounds[1], myBounds[0]+24, myBounds[1]+72];
myAnchoredFrame.geometricBounds = myArray;
myAnchoredFrame.contents = "This is an anchored frame.";
with(myAnchoredFrame.anchoredObjectSettings){
    anchoredPosition = AnchorPosition.anchored;
    anchorPoint = AnchorPoint.topLeftAnchor;
    horizontalReferencePoint = AnchoredRelativeTo.anchorLocation;
    horizontalAlignment = HorizontalAlignment.leftAlign;
    anchorXoffset = 72;
    verticalReferencePoint = VerticallyRelativeTo.lineBaseline;
    anchorYoffset = 24;
    anchorSpaceAbove = 24;
}
```

Fitting text frames to content

The following example script shows how to set rules on a text frame to determine how it grows when the user inputs text (for the complete script, see PersistedTextFrameFill):

```
var myDocument = app.documents.item(0);
var myTextFrame = myDocument.pages.item(0).textFrames.item(3);
//Now add text at the end of the text frame.
myTextFrame.insertionPoints.item(-1).contents = "\rThis is some overset text";
alert("The last paragraph in this alert should be \"This is some overset text\". Is
it?\r"
     + myTextFrame.contents);
alert("The last paragraph in this alert should be \"This is some overset text\". Is
it?\r"
     + myTextFrame.parentStory.contents);
var myTextFrame2 = myDocument.pages.item(0).textFrames.item(2);
//Set auto size dimension of the text frame
with(myTextFrame2.textFramePreferences)
{
    autoSizingDimension=AutoSizingDimension.HEIGHT_AND_WIDTH_PROPORTIONALLY;
}
//Now add text at the end of the text frame.
myTextFrame2.insertionPoints.item(-1).contents = "\rThis is some overset text";
var myTextFrame3 = myDocument.pages.item(0).textFrames.item(1);
```

```
//Set  auto size dimension of the text frame
//useMinimumHeightForAutoSizing and useNoLineBreaksForAutoSizing
with(myTextFrame3.textFramePreferences)
{
    autoSizingDimension=AutoSizingDimension.HEIGHT_ONLY;
    useMinimumHeightForAutoSizing=true
    useNoLineBreaksForAutoSizing=true
}
//Now add text at the end of the text frame.
myTextFrame3.insertionPoints.item(-1).contents = "\rThis is some overset text";
var myTextFrame4 = myDocument.pages.item(0).textFrames.item(0);
//Set  auto size dimension of the text frame and autoSizingReferencePoint
with(myTextFrame4.textFramePreferences)
{
    autoSizingDimension=AutoSizingDimension.WIDTH_ONLY;
    autoSizingReferencePoint=AutoSizingReferencePoint.TOP_LEFT_POSITION;
}
//Now add text at the end of the text frame.
myTextFrame4.insertionPoints.item(-1).contents = "\rThis is some overset text";
```

Formatting Text

In the previous sections of this chapter, we added text to a document, linked text frames, and worked with stories and text objects. In this section, we apply formatting to text. All the typesetting capabilities of InDesign are available to scripting.

Setting text defaults

You can set text defaults for both the application and each document. Text defaults for the application determine the text defaults in all new documents; text defaults for a document set the formatting of all new text objects in that document. (For the complete script, see TextDefaults.)

```
var myDocument = app.documents.item(0);
//To set the application text formatting defaults, replace the variable "myDocument"
//with "app" in the following lines.
with(myDocument.textDefaults){
    alignToBaseline = true;
    //Because the font might not be available, it's usually best
    //to apply the font within a try...catch structure. Fill in the
    //name of a font on your system.
    try{
        appliedFont = app.fonts.item("Minion Pro");
    }
    catch(e){}
    //Because the font style might not be available, it's usually best
    //to apply the font style within a try...catch structure.
    try{
        fontStyle = "Regular";
    }
    catch(e){}
    //Because the language might not be available, it's usually best
    //to apply the language within a try...catch structure.
    try{
        appliedLanguage = "English: USA";
    }
    catch(e){}
    autoLeading = 100;
```

```
balanceRaggedLines = false;
baselineShift = 0;
capitalization = Capitalization.normal;
composer = "Adobe Paragraph Composer";
desiredGlyphScaling = 100;
desiredLetterSpacing = 0;
desiredWordSpacing = 100;
dropCapCharacters = 0;
if(dropCapCharacters != 0){
    dropCapLines = 3;
    //Assumes that the application has a default character style named "myDropCap"
    dropCapStyle = myDocument.characterStyles.item("myDropCap");
}
fillColor = myDocument.colors.item("Black");
fillTint = 100;
firstLineIndent = 14;
gridAlignFirstLineOnly = false;
horizontalScale = 100;
hyphenateAfterFirst = 3;
hyphenateBeforeLast = 4;
hyphenateCapitalizedWords = false;
hyphenateLadderLimit = 1;
hyphenateWordsLongerThan = 5;
hyphenation = true;
hyphenationZone = 36;
hyphenWeight = 9;
justification = Justification.leftAlign;
keepAllLinesTogether = false;
keepLinesTogether = true;
keepFirstLines = 2;
keepLastLines = 2;
keepWithNext = 0;
kerningMethod = "Optical";
kerningValue = 0;
leading = 14;
leftIndent = 0;
ligatures = true;
maximumGlyphScaling = 100;
maximumLetterSpacing = 0;
maximumWordSpacing = 160;
minimumGlyphScaling = 100;
minimumLetterSpacing = 0;
minimumWordSpacing = 80;
noBreak = false;
otfContextualAlternate = true;
otfDiscretionaryLigature = false;
otfFigureStyle = OTFFigureStyle.proportionalOldstyle;
otfFraction = true;
otfHistorical = false;
otfOrdinal = false;
otfSlashedZero = false;
otfSwash = false;
otfTitling = false;
overprintFill = false;
overprintStroke = false;
pointSize = 11;
position = Position.normal;
rightIndent = 0;
ruleAbove = false;
if(ruleAbove == true){
```

```
    ruleAboveColor = myDocument.colors.item("Black");
    ruleAboveGapColor = myDocument.swatches.item("None");
    ruleAboveGapOverprint = false;
    ruleAboveGapTint = 100;
    ruleAboveLeftIndent = 0;
    ruleAboveLineWeight = .25;
    ruleAboveOffset = 14;
    ruleAboveOverprint = false;
    ruleAboveRightIndent = 0;
    ruleAboveTint = 100;
    ruleAboveType = myDocument.strokeStyles.item("Solid");
    ruleAboveWidth = RuleWidth.columnWidth;
}
ruleBelow = false;
if(ruleBelow == true){
    ruleBelowColor = myDocument.colors.item("Black");
    ruleBelowGapColor = myDocument.swatches.item("None");
    ruleBelowGapOverprint = false;
    ruleBelowGapTint = 100;
    ruleBelowLeftIndent = 0;
    ruleBelowLineWeight = .25;
    ruleBelowOffset = 0;
    ruleBelowOverprint = false;
    ruleBelowRightIndent = 0;
    ruleBelowTint = 100;
    ruleBelowType = app.strokeStyles.item("Solid");
    ruleBelowWidth = RuleWidth.columnWidth;
}
singleWordJustification = SingleWordJustification.leftAlign;
skew = 0;
spaceAfter = 0;
spaceBefore = 0;
startParagraph = StartParagraph.anywhere;
strikeThru = false;
if(strikeThru == true){
    strikeThroughColor = myDocument.colors.item("Black");
    strikeThroughGapColor = myDocument.swatches.item("None");
    strikeThroughGapOverprint = false;
    strikeThroughGapTint = 100;
    strikeThroughOffset = 3;
    strikeThroughOverprint = false;
    strikeThroughTint = 100;
    strikeThroughType = myDocument.strokeStyles.item("Solid");
    strikeThroughWeight = .25;
}
strokeColor = myDocument.swatches.item("None");
strokeTint = 100;
strokeWeight = 0;
```

```
    tracking = 0;
    underline = false;
    if(underline == true){
        underlineColor = myDocument.colors.item("Black");
        underlineGapColor = myDocument.swatches.item("None");
        underlineGapOverprint = false;
        underlineGapTint = 100;
        underlineOffset = 3;
        underlineOverprint = false;
        underlineTint = 100;
        underlineType = myDocument.strokeStyles.item("Solid");
        underlineWeight = .25
    }
    verticalScale = 100;
}
```

Working with fonts

The fonts collection of the InDesign application object contains all fonts accessible to InDesign. The fonts collection of a document, by contrast, contains only those fonts used in the document. The fonts collection of a document also contains any missing fonts—fonts used in the document that are not accessible to InDesign. The following script shows the difference between application fonts and document fonts. (We omitted the `myGetBounds` function here; for the complete script, see FontCollections.)

```
var myApplicationFonts = app.fonts;
var myDocument = app.documents.item(0);
var myStory = myDocument.stories.item(0);
var myDocumentFonts = myDocument.fonts;
var myFontNames = myApplicationFonts.everyItem().name;
var myDocumentFontNames = myDocument.fonts.everyItem().name;
var myString = "Document Fonts:\r";
for(var myCounter = 0;myCounter<myDocumentFontNames.length; myCounter++){
    myString += myDocumentFontNames[myCounter] + "\r";
}
myString += "\rApplication Fonts:\r";
for(var myCounter = 0;myCounter<myFontNames.length; myCounter++){
    myString += myFontNames[myCounter] + "\r";
}
myStory.contents = myString;
```

NOTE: Font names typically are of the form *familyName*<tab>*fontStyle*, where *familyName* is the name of the font family, <tab> is a tab character, and *fontStyle* is the name of the font style. For example:

```
"Adobe Caslon Pro<tab>Semibold Italic"
```

Applying a font

To apply a local font change to a range of text, use the `appliedFont` property, as shown in the following script fragment (from the ApplyFont tutorial script):

```
//Given a font name "myFontName" and a text object "myText"...
myText.appliedFont = app.fonts.item(myFontName);
```

You also can apply a font by specifying the font family name and font style, as shown in the following script fragment:

```
myText.appliedFont = app.fonts.item("Adobe Caslon Pro");
myText.fontStyle = "Semibold Italic";
```

Changing text properties

Text objects in InDesign have literally dozens of properties corresponding to their formatting attributes. Even one insertion point features properties that affect the formatting of text—up to and including properties of the paragraph containing the insertion point. The SetTextProperties tutorial script shows how to set every property of a text object. A fragment of the script is shown below:

```
var myDocument = app.documents.item(0);
var myPage = myDocument.pages.item(0);
var myTextFrame = myPage.textFrames.add();
myTextFrame.contents = "x";
var myTextObject = myTextFrame.parentStory.characters.item(0);
myTextObject.alignToBaseline = false;
myTextObject.appliedCharacterStyle = myDocument.characterStyles.item("[None]");
myTextObject.appliedFont = app.fonts.item("Minion ProRegular");
myTextObject.appliedLanguage = app.languagesWithVendors.item("English: USA");
myTextObject.appliedNumberingList = myDocument.numberingLists.item("[Default]");
myTextObject.appliedParagraphStyle = myDocument.paragraphStyles.item("[No Paragraph
Style]");
myTextObject.autoLeading = 120;
myTextObject.balanceRaggedLines = BalanceLinesStyle.noBalancing;
myTextObject.baselineShift = 0;
myTextObject.bulletsAlignment = ListAlignment.leftAlign;
myTextObject.bulletsAndNumberingListType = ListType.noList;
myTextObject.bulletsCharacterStyle = myDocument.characterStyles.item("[None]");
myTextObject.bulletsTextAfter = "^t";
myTextObject.capitalization = Capitalization.normal;
myTextObject.composer = "Adobe Paragraph Composer";
myTextObject.desiredGlyphScaling = 100;
myTextObject.desiredLetterSpacing = 0;
myTextObject.desiredWordSpacing = 100;
myTextObject.dropCapCharacters = 0;
myTextObject.dropCapLines = 0;
myTextObject.dropCapStyle = myDocument.characterStyles.item("[None]");
myTextObject.dropcapDetail = 0;
myTextObject.fillColor = myDocument.colors.item("Black");
myTextObject.fillTint = -1;
myTextObject.firstLineIndent = 0;
myTextObject.fontStyle = "Regular";
myTextObject.gradientFillAngle = 0;
myTextObject.gradientFillLength = -1;
myTextObject.gradientFillStart = [0,0];
myTextObject.gradientStrokeAngle = 0;
myTextObject.gradientStrokeLength = -1;
myTextObject.gradientStrokeStart = [0,0];
myTextObject.gridAlignFirstLineOnly = false;
myTextObject.horizontalScale = 100;
myTextObject.hyphenWeight = 5;
myTextObject.hyphenateAcrossColumns = true;
myTextObject.hyphenateAfterFirst = 2;
myTextObject.hyphenateBeforeLast = 2;
myTextObject.hyphenateCapitalizedWords = true;
myTextObject.hyphenateLadderLimit = 3;
myTextObject.hyphenateLastWord = true;
myTextObject.hyphenateWordsLongerThan = 5;
myTextObject.hyphenation = true;
myTextObject.hyphenationZone = 3;
myTextObject.ignoreEdgeAlignment = false;
myTextObject.justification = Justification.leftAlign;
```

```
myTextObject.keepAllLinesTogether = false;
myTextObject.keepFirstLines = 2;
myTextObject.keepLastLines = 2;
myTextObject.keepLinesTogether = false;
myTextObject.keepRuleAboveInFrame = false;
myTextObject.keepWithNext = 0;
myTextObject.kerningMethod = "Optical";
//myTextObject.kerningValue = error;
myTextObject.lastLineIndent = 0;
myTextObject.leading = 12;
myTextObject.leftIndent = 0;
myTextObject.ligatures = true;
myTextObject.maximumGlyphScaling = 100;
myTextObject.maximumLetterSpacing = 0;
myTextObject.maximumWordSpacing = 133;
myTextObject.minimumGlyphScaling = 100;
myTextObject.minimumLetterSpacing = 0;
myTextObject.minimumWordSpacing = 80;
myTextObject.noBreak = false;
myTextObject.numberingAlignment = ListAlignment.leftAlign;
myTextObject.numberingApplyRestartPolicy = true;
myTextObject.numberingCharacterStyle = myDocument.characterStyles.item("[None]");
myTextObject.numberingContinue = true;
myTextObject.numberingExpression = "^#.^t";
myTextObject.numberingFormat = "1, 2, 3, 4...";
myTextObject.numberingLevel = 1;
myTextObject.numberingStartAt = 1;
myTextObject.otfContextualAlternate = true;
myTextObject.otfDiscretionaryLigature = false;
myTextObject.otfFigureStyle = OTFFigureStyle.proportionalLining;
myTextObject.otfFraction = false;
myTextObject.otfHistorical = false;
myTextObject.otfLocale = true;
myTextObject.otfMark = true;
myTextObject.otfOrdinal = false;
myTextObject.otfSlashedZero = false;
myTextObject.otfStylisticSets = 0;
myTextObject.otfSwash = false;
myTextObject.otfTitling = false;
myTextObject.overprintFill = false;
myTextObject.overprintStroke = false;
myTextObject.pointSize = 12;
myTextObject.position = Position.normal;
myTextObject.positionalForm = PositionalForms.none;
myTextObject.rightIndent = 0;
myTextObject.ruleAbove = false;
myTextObject.ruleAboveColor = "Text Color";
myTextObject.ruleAboveGapColor = myDocument.swatches.item("None");
myTextObject.ruleAboveGapOverprint = false;
myTextObject.ruleAboveGapTint = -1;
myTextObject.ruleAboveLeftIndent = 0;
myTextObject.ruleAboveLineWeight = 1;
myTextObject.ruleAboveOffset = 0;
myTextObject.ruleAboveOverprint = false;
myTextObject.ruleAboveRightIndent = 0;
myTextObject.ruleAboveTint = -1;
myTextObject.ruleAboveType = myDocument.strokeStyles.item("Solid");
myTextObject.ruleAboveWidth = RuleWidth.columnWidth;
myTextObject.ruleBelow = false;
myTextObject.ruleBelowColor = "Text Color";
```

```
myTextObject.ruleBelowGapColor = myDocument.swatches.item("None");
myTextObject.ruleBelowGapOverprint = false;
myTextObject.ruleBelowGapTint = -1;
myTextObject.ruleBelowLeftIndent = 0;
myTextObject.ruleBelowLineWeight = 1;
myTextObject.ruleBelowOffset = 0;
myTextObject.ruleBelowOverprint = false;
myTextObject.ruleBelowRightIndent = 0;
myTextObject.ruleBelowTint = -1;
myTextObject.ruleBelowType = myDocument.strokeStyles.item("Solid");
myTextObject.ruleBelowWidth = RuleWidth.columnWidth;
myTextObject.singleWordJustification = 1718971500;
myTextObject.skew = 0;
myTextObject.spaceAfter = 0;
myTextObject.spaceBefore = 0;
myTextObject.startParagraph = 1851945579;
myTextObject.strikeThroughColor = "Text Color";
myTextObject.strikeThroughGapColor = myDocument.swatches.item("None");
myTextObject.strikeThroughGapOverprint = false;
myTextObject.strikeThroughGapTint = -1;
myTextObject.strikeThroughOffset = -9999;
myTextObject.strikeThroughOverprint = false;
myTextObject.strikeThroughTint = -1;
myTextObject.strikeThroughType = myDocument.strokeStyles.item("Solid");
myTextObject.strikeThroughWeight = -9999;
myTextObject.strikeThru = false;
myTextObject.strokeColor = myDocument.swatches.item("None");
myTextObject.strokeTint = -1;
myTextObject.strokeWeight = 1;
myTextObject.tracking = 0;
myTextObject.underline = false;
myTextObject.underlineColor = "Text Color";
myTextObject.underlineGapColor = myDocument.swatches.item("None");
myTextObject.underlineGapOverprint = false;
myTextObject.underlineGapTint = -1;
myTextObject.underlineOffset = -9999;
myTextObject.underlineOverprint = false;
myTextObject.underlineTint = -1;
myTextObject.underlineType = myDocument.strokeStyles.item("Solid");
myTextObject.underlineWeight = -9999;
myTextObject.verticalScale = 100;
```

Changing text color

You can apply colors to the fill and stroke of text characters, as shown in the following script fragment (from the TextColors tutorial script):

```
var myColorA, myColorB, myName;
//Access the active document and page.
var myDocument = app.activeDocument;
var myPage = app.activeWindow.activePage;
//Create a color.
try{
    myColorA = myDocument.colors.item("DGC1_664a");
    //If the color does not exist, trying to get its name will generate an error.
    myName = myColorA.name;
}
catch (myError){
    //The color style did not exist, so create it.
    myColorA = myDocument.colors.add({name:"DGC1_664a", model:ColorModel.process,
colorValue:[90, 100, 70, 0]});
}
//Create another color.
try{
    myColorB = myDocument.colors.item("DGC1_664b");
    //If the color does not exist, trying to get its name will generate an error.
    myName = myColorB.name;
}
catch (myError){
    //The color style did not exist, so create it.
    myColorB = myDocument.colors.add({name:"DGC1_664b", model:ColorModel.process,
colorValue:[70, 0, 30, 50]});
}
//Create a text frame on the active page.
var myTextFrame = myPage.textFrames.add();
//Set the bounds of the text frame.
myTextFrame.geometricBounds = myGetBounds(myDocument, myPage);
//Enter text in the text frame.
myTextFrame.contents = "Text\rColor"
var myText = myTextFrame.parentStory.paragraphs.item(0)
myText.pointSize = 72;
myText.justification = Justification.centerAlign;
//Apply a color to the fill of the text.
myText.fillColor = myColorA;
//Use the itemByRange method to apply the color to the stroke of the text.
myText.strokeColor = myColorB;
var myText = myTextFrame.parentStory.paragraphs.item(1)
myText.strokeWeight = 3;
myText.pointSize = 144;
myText.justification = Justification.centerAlign;
myText.fillColor = myColorB;
myText.strokeColor = myColorA;
myText.strokeWeight = 3;
```

Creating and applying styles

While you can use scripting to apply local formatting—as in some of the examples earlier in this chapter—you probably will want to use character and paragraph styles to format your text. Using styles creates a link between the formatted text and the style, which makes it easier to redefine the style, collect the text formatted with a given style, or find and/or change the text. Paragraph and character styles are the keys to text formatting productivity and should be a central part of any script that applies text formatting.

The following example script fragment shows how to create and apply paragraph and character styles (for the complete script, see CreateStyles):

```
var myDocument = app.documents.item(0);
var myPage = myDocument.pages.item(0);
//Create a color for use by one of the paragraph styles we'll create.
try{
    myColor = myDocument.colors.item("Red");
    //If the color does not exist, trying to get its name will generate an error.
    myName = myColor.name;
}
catch (myError){
    //The color style did not exist, so create it.
    myColor = myDocument.colors.add({name:"Red", model:ColorModel.process,
colorValue:[0, 100, 100, 0]});
}
//Create a text frame on the active page.
var myTextFrame = myPage.textFrames.add();
//Set the bounds of the text frame.
myTextFrame.geometricBounds = myGetBounds(myDocument, myPage);
//Fill the text frame with placeholder text.
myTextFrame.contents = "Normal text. Text with a character style applied to it. More
normal text.";
//Create a character style named "myCharacterStyle" if
//no style by that name already exists.
try{
    myCharacterStyle = myDocument.characterStyles.item("myCharacterStyle");
    //If the style does not exist, trying to get its name will generate an error.
    myName = myCharacterStyle.name;
}
catch (myError){
    //The style did not exist, so create it.
    myCharacterStyle = myDocument.characterStyles.add({name:"myCharacterStyle"});
}
//At this point, the variable myCharacterStyle contains a reference to a character
//style object, which you can now use to specify formatting.
myCharacterStyle.fillColor = myColor;
//Create a paragraph style named "myParagraphStyle" if
//no style by that name already exists.
try{
    myParagraphStyle = myDocument.paragraphStyles.item("myParagraphStyle");
    //If the paragraph style does not exist, trying to get its name will generate an
error.
    myName = myParagraphStyle.name;
}
catch (myError){
    //The paragraph style did not exist, so create it.
    myParagraphStyle = myDocument.paragraphStyles.add({name:"myParagraphStyle"});
}
//At this point, the variable myParagraphStyle contains a reference to a paragraph
//style object, which you can now use to specify formatting.
myTextFrame.parentStory.texts.item(0).applyParagraphStyle(myParagraphStyle, true);
var myStartCharacter = myTextFrame.parentStory.characters.item(13);
var myEndCharacter = myTextFrame.parentStory.characters.item(54);
myTextFrame.parentStory.texts.itemByRange(myStartCharacter,
myEndCharacter).applyParagraphStyle(myParagraphStyle, true);
```

Why use the `applyParagraphStyle` method instead of setting the `appliedParagraphStyle` property of the text object? The `applyParagraphStyle` method gives the ability to override existing formatting; setting the property to a style retains local formatting.

Why check for the existence of a style when creating a new document? It always is possible that the style exists as an application default style. If it does, trying to create a new style with the same name results in an error.

Nested styles apply character-style formatting to a paragraph according to a pattern. The following script fragment shows how to create a paragraph style containing nested styles (for the complete script, see NestedStyles):

```
var myDocument = app.documents.item(0);
var myPage = myDocument.pages.item(0);
var myTextFrame = myPage.textFrames.item(0);
var myParagraphStyle = myDocument.paragraphStyles.item("myParagraphStyle");
var myNestedStyle =
myParagraphStyle.nestedStyles.add({appliedCharacterStyle:myCharacterStyle,
delimiter:".", inclusive:true, repetition:1});
var myStartCharacter = myTextFrame.parentStory.characters.item(0);
var myEndCharacter = myTextFrame.parentStory.characters.item(-1);
//Use the itemByRange method to apply the paragraph to all of the text in the story.
//(Note that the story object does not have the applyParagraphStyle method.)
myTextFrame.parentStory.texts.itemByRange(myStartCharacter,
myEndCharacter).applyParagraphStyle(myParagraphStyle, true);
```

Deleting a style

When you delete a style using the user interface, you can choose the way you want to format any text tagged with that style. InDesign scripting works the same way, as shown in the following script fragment (from the RemoveStyle tutorial script):

```
var myDocument = app.activeDocument;
var myParagraphStyleA = myDocument.paragraphStyles.item("myParagraphStyleA");
//Remove the paragraph style myParagraphStyleA and replace with myParagraphStyleB.
myParagraphStyleA.remove(myDocument.paragraphStyles.item("myParagraphStyleB"));
```

Importing paragraph and character styles

You can import character and paragraph styles from other InDesign documents, as shown in the following script fragment (from the ImportTextStyles tutorial script):

```
//Create a new document.
myDocument = app.documents.add();
//Import the styles from the saved document.
//importStyles parameters:
// Format as ImportFormat enumeration. Options for text styles are:
// ImportFormat.paragraphStylesFormat
// ImportFormat.characterStylesFormat
// ImportFormat.textStylesFormat
// From as File
// GlobalStrategy as GlobalClashResolutionStrategy enumeration. Options are:
// GlobalClashResolutionStrategy.doNotLoadTheStyle
// GlobalClashResolutionStrategy.loadAllWithOverwrite
// GlobalClashResolutionStrategy.loadAllWithRename
myDocument.importStyles(ImportFormat.textStylesFormat, File("/c/styles.indd"),
GlobalClashResolutionStrategy.loadAllWithOverwrite);
```

Finding and Changing Text

The find/change feature is one of the most powerful InDesign tools for working with text. It is fully supported by scripting, and scripts can use find/change to go far beyond what can be done using the InDesign user interface. InDesign has three ways of searching for text:

- You can find text and/or text formatting and change it to other text and/or text formatting. This type of find/change operation uses the `findTextPreferences` and `changeTextPreferences` objects to specify parameters for the `findText` and `changeText` methods.
- You can find text using regular expressions, or "grep." This type of find/change operation uses the `findGrepPreferences` and `changeGrepPreferences` objects to specify parameters for the `findGrep` and `changeGrep` methods.
- You can find specific glyphs (and their formatting) and replace them with other glyphs and formatting. This type of find/change operation uses the `findGlyphPreferences` and `changeGlyphPreferences` objects to specify parameters for the `findGlyph` and `changeGlyph` methods.

All the find/change methods take one optional parameter, `ReverseOrder`, which specifies the order in which the results of the search are returned. If you are processing the results of a find or change operation in a way that adds or removes text from a story, you might face the problem of invalid text references, as discussed earlier in this chapter. In this case, you can either construct your loops to iterate backward through the collection of returned text objects, or you can have the search operation return the results in reverse order and then iterate through the collection normally.

About find/change preferences

Before you search for text, you probably will want to clear find and change preferences, to make sure the settings from previous searches have no effect on your search. You also need to set some find/change preferences to specify the text, formatting, regular expression, or glyph you want to find and/or change. A typical find/change operation involves the following steps:

1. Clear the find/change preferences. Depending on the type of find/change operation, this can take one of the following three forms:

   ▷
   ```
   //find/change text preferences
   app.findTextPreferences = NothingEnum.nothing;
   app.changeTextPreferences = NothingEnum.nothing;
   ```

   ▷
   ```
   //find/change grep preferences
   app.findGrepPreferences = NothingEnum.nothing;
   app.changeGrepPreferences = NothingEnum.nothing;
   ```

   ▷
   ```
   //find/change glyph preferences
   app.findGlyphPreferences = NothingEnum.nothing;
   app.changeGlyphPreferences = NothingEnum.nothing;
   ```

2. Set up search parameters.
3. Execute the find/change operation.
4. Clear find/change preferences again.

Finding and changing text

The following script fragment shows how to find a specified string of text. While the following script fragment searches the entire document, you also can search stories, text frames, paragraphs, text columns, or any other text object. The `findText` method and its parameters are the same for all text objects. (For the complete script, see FindText.)

```
var myDocument = app.activeDocument;
//Clear the find/change text preferences.
app.findTextPreferences = NothingEnum.nothing;
app.changeTextPreferences = NothingEnum.nothing;
//Search the document for the string "Text".
app.findTextPreferences.findWhat = "text";
//Set the find options.
app.findChangeTextOptions.caseSensitive = false;
app.findChangeTextOptions.includeFootnotes = false;
app.findChangeTextOptions.includeHiddenLayers = false;
app.findChangeTextOptions.includeLockedLayersForFind = false;
app.findChangeTextOptions.includeLockedStoriesForFind = false;
app.findChangeTextOptions.includeMasterPages = false;
app.findChangeTextOptions.wholeWord = false;
var myFoundItems = myDocument.findText();
alert("Found " + myFoundItems.length + " instances of the search string.");
app.findTextPreferences = NothingEnum.nothing;
app.changeTextPreferences = NothingEnum.nothing;
```

The following script fragment shows how to find a specified string of text and replace it with a different string (for the complete script, see ChangeText):

```
var myDocument = app.activeDocument;
//Clear the find/change text preferences.
app.findTextPreferences = NothingEnum.nothing;
app.changeTextPreferences = NothingEnum.nothing;
//Set the find options.
app.findChangeTextOptions.caseSensitive = false;
app.findChangeTextOptions.includeFootnotes = false;
app.findChangeTextOptions.includeHiddenLayers = false;
app.findChangeTextOptions.includeLockedLayersForFind = false;
app.findChangeTextOptions.includeLockedStoriesForFind = false;
app.findChangeTextOptions.includeMasterPages = false;
app.findChangeTextOptions.wholeWord = false;
//Search the document for the string "copy" and change it to "text".
app.findTextPreferences.findWhat = "copy";
app.changeTextPreferences.changeTo = "text";
myDocument.changeText();
//Clear the find/change text preferences after the search.
app.findTextPreferences = NothingEnum.nothing;
app.changeTextPreferences = NothingEnum.nothing;
```

Finding and changing text formatting

To find and change text formatting, you set other properties of the `findTextPreferences` and `changeTextPreferences` objects, as shown in the script fragment below (from the FindChangeFormatting tutorial script):

```
var myDocument = app.documents.item(0);
//Clear the find/change preferences.
app.findTextPreferences = NothingEnum.nothing;
app.changeTextPreferences = NothingEnum.nothing;
//Set the find options.
app.findChangeTextOptions.caseSensitive = false;
app.findChangeTextOptions.includeFootnotes = false;
app.findChangeTextOptions.includeHiddenLayers = false;
app.findChangeTextOptions.includeLockedLayersForFind = false;
app.findChangeTextOptions.includeLockedStoriesForFind = false;
app.findChangeTextOptions.includeMasterPages = false;
app.findChangeTextOptions.wholeWord = false;
//Search the document for the 24 point text and change it to 10 point text.
app.findTextPreferences.pointSize = 24;
app.changeTextPreferences.pointSize = 10;
myDocument.changeText();
//Clear the find/change preferences after the search.
app.findTextPreferences = NothingEnum.nothing;
app.changeTextPreferences = NothingEnum.nothing;
```

Using grep

InDesign supports regular expression find/change through the `findGrep` and `changeGrep` methods. Regular-expression find/change also can find text with a specified format or replace the formatting of the text with formatting specified in the properties of the `changeGrepPreferences` object. The following script fragment shows how to use these methods and the related preferences objects (for the complete script, see FindGrep):

```
var myDocument = app.documents.item(0);
//Clear the find/change grep preferences.
app.findGrepPreferences = NothingEnum.nothing;
app.changeGrepPreferences = NothingEnum.nothing;
//Set the find options.
app.findChangeGrepOptions.includeFootnotes = false;
app.findChangeGrepOptions.includeHiddenLayers = false;
app.findChangeGrepOptions.includeLockedLayersForFind = false;
app.findChangeGrepOptions.includeLockedStoriesForFind = false;
app.findChangeGrepOptions.includeMasterPages = false;
//Regular expression for finding an email address.
app.findGrepPreferences.findWhat = "(?i)[A-Z]*?@[A-Z]*?[.]...";
//Apply the change to 24-point text only.
app.findGrepPreferences.pointSize = 24;
app.changeGrepPreferences.underline = true;
myDocument.changeGrep();
//Clear the find/change preferences after the search.
app.findGrepPreferences = NothingEnum.nothing;
app.changeGrepPreferences = NothingEnum.nothing;
```

NOTE: The `findChangeGrepOptions` object lacks two properties of the `findChangeTextOptions` object: `wholeWord` and `caseSensitive`. This is because you can set these options using the regular expression string itself. Use `(?i)` to turn case sensitivity on and `(?-i)` to turn case sensitivity off. Use `\>` to match the beginning of a word and `\<` to match the end of a word, or use `\b` to match a word boundary.

One handy use for grep find/change is to convert text mark-up (i.e., some form of tagging plain text with formatting instructions) into InDesign formatted text. PageMaker paragraph tags (which are not the same as PageMaker tagged-text format files) are an example of a simplified text mark-up scheme. In a text file marked up using this scheme, paragraph style names appear at the start of a paragraph, as shown below:

```
<heading1>This is a heading.
<body_text>This is body text.
```

We can create a script that uses grep find in conjunction with text find/change operations to apply formatting to the text and remove the mark-up tags, as shown in the following script fragment (from the ReadPMTags tutorial script):

```
var myDocument = app.documents.item;
var myStory = myDocument.stories.item(0);
myReadPMTags(myStory);
```

Here is the myReadPMTags function referred to in the above script.

```
function myReadPMTags(myStory){
    var myName, myString, myStyle, myStyleName;
    var myDocument = app.documents.item(0);
    //Reset the findGrepPreferences to ensure that previous settings
    //do not affect the search.
    app.findGrepPreferences = NothingEnum.nothing;
    app.changeGrepPreferences = NothingEnum.nothing;
    //Find the tags (since this is a JavaScript string,
    //the backslashes must be escaped).
    app.findGrepPreferences.findWhat = "(?i)^<\\s*\\w+\\s*>";
    var myFoundItems = myStory.findGrep();
    if(myFoundItems.length != 0){
        var myFoundTags = new Array;
        for(var myCounter = 0; myCounter<myFoundItems.length; myCounter++){
            myFoundTags.push(myFoundItems[myCounter].contents);
        }
        myFoundTags = myRemoveDuplicates(myFoundTags);
        //At this point, we have a list of tags to search for.
        for(myCounter = 0; myCounter < myFoundTags.length; myCounter++){
            myString = myFoundTags[myCounter];
            //Find the tag using findWhat.
            app.findTextPreferences.findWhat = myString;
            //Extract the style name from the tag.
            myStyleName = myString.substring(1, myString.length-1);
            //Create the style if it does not already exist.
            try{
                myStyle = myDocument.paragraphStyles.item(myStyleName);
                myName = myStyle.name;
            }
            catch (myError){
                myStyle = myDocument.paragraphStyles.add({name:myStyleName});
            }
            //Apply the style to each instance of the tag.
            app.changeTextPreferences.appliedParagraphStyle = myStyle;
            myStory.changeText();
            //Reset the changeTextPreferences.
            app.changeTextPreferences = NothingEnum.nothing;
            //Set the changeTo to an empty string.
            app.changeTextPreferences.changeTo = "";
            //Search to remove the tags.
            myStory.changeText();
            //Reset the find/change preferences again.
            app.changeTextPreferences = NothingEnum.nothing;
        }
    }
    //Reset the findGrepPreferences.
    app.findGrepPreferences  = NothingEnum.nothing;
```

```
}
function myRemoveDuplicates(myArray){
    //Semi-clever method of removing duplicate array items; much faster
    //than comparing every item to every other item!
    var myNewArray = new Array;
    myArray = myArray.sort();
    myNewArray.push(myArray[0]);
    if(myArray.length > 1){
        for(var myCounter = 1; myCounter < myArray.length; myCounter ++){
            if(myArray[myCounter] != myNewArray[myNewArray.length -1]){
                myNewArray.push(myArray[myCounter]);
            }
        }
    }
    return myNewArray;
}
```

Using glyph search

You can find and change individual characters in a specific font using the `findGlyph` and `changeGlyph` methods and the associated `findGlyphPreferences` and `changeGlyphPreferences` objects. The following scripts fragment shows how to find and change a glyph in an example document (for the complete script, see FindChangeGlyph):

```
//Clear glyph search preferences.
app.findGlyphPreferences = NothingEnum.nothing;
app.changeGlyphPreferences = NothingEnum.nothing;
var myDocument = app.documents.item(0);
//You must provide a font that is used in the document for the
//appliedFont property of the findGlyphPreferences object.
app.findGlyphPreferences.appliedFont = app.fonts.itemByName("Minion ProRegular");
//Provide the glyph ID, not the glyph Unicode value.
app.findGlyphPreferences.glyphID = 374;
//The appliedFont of the changeGlyphPreferences object can be
//any font available to the application.
app.changeGlyphPreferences.appliedFont = app.fonts.itemByName("Times New RomanBold");
app.changeGlyphPreferences.glyphID = 85;
myDocument.changeGlyph(false);
//Clear glyph search preferences.
app.findGlyphPreferences = NothingEnum.nothing;
app.changeGlyphPreferences = NothingEnum.nothing;
```

Working with Tables

Tables can be created from existing text using the `convertTextToTable` method, or an empty table can be created at any insertion point in a story. The following script fragment shows three different ways to create a table (for the complete script, see MakeTable):

```
var myDocument = app.documents.item(0);
var myStory = myDocument.stories.item(0);
var myStartCharacter = myStory.paragraphs.item(6).characters.item(0);
var myEndCharacter = myStory.paragraphs.item(6).characters.item(-2);
var myText = myStory.texts.itemByRange(myStartCharacter, myEndCharacter);
//The convertToTable method takes three parameters:
//[ColumnSeparator as string]
//[RowSeparator as string]
//[NumberOfColumns as integer] (only used if the ColumnSeparator
//and RowSeparator values are the same)
//In the last paragraph in the story, columns are separated by commas
//and rows are separated by semicolons, so we provide those characters
//to the method as parameters.
var myTable = myText.convertToTable(",",";");
var myStartCharacter = myStory.paragraphs.item(1).characters.item(0);
var myEndCharacter = myStory.paragraphs.item(4).characters.item(-2);
var myText = myStory.texts.itemByRange(myStartCharacter, myEndCharacter);
//In the second through the fifth paragraphs, colums are separated by
//tabs and rows are separated by returns. These are the default delimiter
//parameters, so we don't need to provide them to the method.
var myTable = myText.convertToTable();
//You can also explicitly add a table--you don't have to convert text to a table.
var myTable = myStory.insertionPoints.item(-1).tables.add();
myTable.columnCount = 3;
myTable.bodyRowCount = 3;
```

The following script fragment shows how to merge table cells. (For the complete script, see MergeTableCells.)

```
var myDocument = app.documents.item(0);
var myStory = myDocument.stories.item(0);
var myTable = myStory.insertionPoints.item(-1).tables.add();
myTable.columnCount = 4;
myTable.bodyRowCount = 4;
//Merge all of the cells in the first column.
myTable.cells.item(0).merge(myTable.columns.item(0).cells.item(-1));
//Convert column 2 into 2 cells (rather than 4).
myTable.columns.item(1).cells.item(-1).merge(myTable.columns.item(1).cells.item(-2));
myTable.columns.item(1).cells.item(0).merge(myTable.columns.item(1).cells.item(1));
//Merge the last two cells in row 1.
myTable.rows.item(0).cells.item(-2).merge(myTable.rows.item(0).cells.item(-1));
//Merge the last two cells in row 3.
myTable.rows.item(2).cells.item(-2).merge(myTable.rows.item(2).cells.item(-1));
```

The following script fragment shows how to split table cells. (For the complete script, see SplitTableCells.)

```
var myStory = myDocument.stories.item(0);
var myTable = myStory.insertionPoints.item(-1).tables.add();
myTable.columnCount = 1;
myTable.bodyRowCount = 1;
var myArray = myGetBounds(myDocument, myDocument.pages.item(0))
var myWidth = myArray[3]-myArray[1];
myTable.columns.item(0).width = myWidth;
myTable.cells.item(0).split(HorizontalOrVertical.horizontal);
myTable.columns.item(0).split(HorizontalOrVertical.vertical);
myTable.cells.item(0).split(HorizontalOrVertical.vertical);
myTable.rows.item(-1).split(HorizontalOrVertical.horizontal);
myTable.cells.item(-1).split(HorizontalOrVertical.vertical);
for(myRowCounter = 0; myRowCounter < myTable.rows.length; myRowCounter ++){
    myRow = myTable.rows.item(myRowCounter);
    for(myCellCounter = 0; myCellCounter < myRow.cells.length; myCellCounter ++){
        myString = "Row: " + myRowCounter + " Cell: " + myCellCounter;
        myRow.cells.item(myCellCounter).contents = myString;
    }
}
```

The following script fragment shows how to create header and footer rows in a table (for the complete script, see HeaderAndFooterRows):

```
var myDocument = app.documents.item(0);
var myTable = myDocument.stories.item(0).tables.item(0);
//Convert the first row to a header row.
myTable.rows.item(0).rowType = RowTypes.headerRow;
//Convert the last row to a footer row.
myTable.rows.item(-1).rowType = RowTypes.footerRow;
```

The following script fragment shows how to apply formatting to a table (for the complete script, see TableFormatting):

```
var myDocument = app.documents.item(0);
var myTable = myDocument.stories.item(0).tables.item(0);
//Convert the first row to a header row.
myTable.rows.item(0).rowType = RowTypes.headerRow;
//Use a reference to a swatch, rather than to a color.
myTable.rows.item(0).fillColor = myDocument.swatches.item("DGC1_446b");
myTable.rows.item(0).fillTint = 40;
myTable.rows.item(1).fillColor = myDocument.swatches.item("DGC1_446a");
myTable.rows.item(1).fillTint = 40;
myTable.rows.item(2).fillColor = myDocument.swatches.item("DGC1_446a");
myTable.rows.item(2).fillTint = 20;
myTable.rows.item(3).fillColor = myDocument.swatches.item("DGC1_446a");
myTable.rows.item(3).fillTint = 40;
//Use everyItem to set the formatting of multiple cells at once.
myTable.cells.everyItem().topEdgeStrokeColor = myDocument.swatches.item("DGC1_446b");
myTable.cells.everyItem().topEdgeStrokeWeight = 1;
myTable.cells.everyItem().bottomEdgeStrokeColor =
myDocument.swatches.item("DGC1_446b");
myTable.cells.everyItem().bottomEdgeStrokeWeight = 1;
//When you set a cell stroke to a swatch, make certain
//that you also set the stroke weight.
myTable.cells.everyItem().leftEdgeStrokeColor = myDocument.swatches.item("None");
myTable.cells.everyItem().leftEdgeStrokeWeight = 0;
myTable.cells.everyItem().rightEdgeStrokeColor = myDocument.swatches.item("None");
myTable.cells.everyItem().rightEdgeStrokeWeight = 0;
```

The following script fragment shows how to add alternating row formatting to a table (for the complete script, see AlternatingRows):

```
//Given a table "myTable," apply alternating fills to the table.
myTable.alternatingFills = AlternatingFillsTypes.alternatingRows;
myTable.startRowFillColor = myDocument.swatches.item("DGC1_446a");
myTable.startRowFillTint = 60;
myTable.endRowFillColor = myDocument.swatches.item("DGC1_446b");
myTable.endRowFillTint = 50;
```

The following script fragment shows how to process the selection when text or table cells are selected. In this example, the script displays an alert for each selection condition, but a real production script would then do something with the selected item(s). (For the complete script, see TableSelection.)

```
if(app.documents.length != 0){
    if(app.selection.length != 0){
        switch(app.selection[0].constructor.name){
            //When a row, a column, or a range of cells is selected,
            //the type returned is "Cell"
            case "Cell":
                alert("A cell is selected.");
                break;
            case "Table":
                alert("A table is selected.");
                break;
            case "InsertionPoint":
            case "Character":
            case "Word":
            case "TextStyleRange":
            case "Line":
            case "Paragraph":
            case "TextColumn":
            case "Text":
                if(app.selection[0].parent.constructor.name == "Cell"){
                    alert("The selection is inside a table cell.");
                }
                break;
            case "Rectangle":
            case "Oval":
            case "Polygon":
            case "GraphicLine":
                if(app.selection[0].parent.parent.constructor.name == "Cell"){
                    alert("The selection is inside a table cell.");
                }
                break;
            case "Image":
            case "PDF":
            case "EPS":
                if(app.selection[0].parent.parent.parent.constructor.name == "Cell"){
                    alert("The selection is inside a table cell.");
                }
                break;
            default:
                alert("The selection is not inside a table.");
                break;
        }
    }
}
```

Adding Path Text

You can add path text to any rectangle, oval, polygon, graphic line, or text frame. The following script fragment shows how to add path text to a page item (for the complete script, see PathText):

```
//Given a document "myDocument" with a rectangle on page 1...
var myPage = myDocument.pages.item(0);
var myRectangle = myPage.rectangles.item(0);
var myTextPath = myRectangle.textPaths.add({contents:"This is path text."});
```

To link text paths to another text path or text frame, use the `nextTextFrame` and `previousTextFrame` properties, just as you would for a text frame (see "Working with Text Frames" on page 83).

Using Autocorrect

The autocorrect feature can correct text as you type. The following script shows how to use it (for the complete script, see Autocorrect):

```
//The autocorrect preferences object turns the
//autocorrect feature on or off.
app.autoCorrectPreferences.autoCorrect = true;
app.autoCorrectPreferences.autoCorrectCapitalizationErrors = true;
//Add a word pair to the autocorrect list. Each AutoCorrectTable is linked
//to a specific language.
var myAutoCorrectTable = app.autoCorrectTables.item("English: USA");
//To safely add a word pair to the auto correct table, get the current
//word pair list, then add the new word pair to that array, and then
//set the autocorrect word pair list to the array.
var myWordPairList = myAutoCorrectTable.autoCorrectWordPairList;
//Add a new word pair to the array.
myWordPairList.push(["paragarph", "paragraph"]);
//Update the word pair list.
myAutoCorrectTable.autoCorrectWordPairList = myWordPairList;
//To clear all autocorrect word pairs in the current dictionary:
//myAutoCorrectTable.autoCorrectWordPairList = [[]];
```

Adding Footnotes

The following script fragment shows how to add footnotes to a story (for the complete script, including the `myGetRandom` function, see Footnotes):

```
var myDocument = app.documents.item(0);
var myPage = myDocument.pages.item(0);
var myTextFrame = myPage.textFrames.item(0);
//Add four footnotes at random locations in the story.
for(myCounter = 0; myCounter < 4; myCounter ++){
    myWord = myTextFrame.parentStory.words.item(myGetRandom(0,
    myTextFrame.parentStory.words.length));
    var myFootnote = myWord.insertionPoints.item(-1).footnotes.add();
    //Note: when you create a footnote, it contains text--the footnote marker
    //and the separator text (if any). If you try to set the text of the footnote
    //by setting the footnote contents, you will delete the marker. Instead, append
    //the footnote text, as shown below.
    myFootnote.insertionPoints.item(-1).contents = "This is a footnote.";
}
```

Spanning Columns

A paragraph layout can span multiple columns or split into subcolumns with the Span Columns attribute or Split Column attribute applied. The following script fragment shows how to set the Span Columns and Split Column style for a paragraph (for the complete script, see SpanColumns):

```
var myDocument = app.activeDocument;
var myPage = myDocument.pages.item(0);
var myTextFrame = myPage.textFrames.item(0);
myTextFrame.textFramePreferences.textColumnCount = 3;
var myStory = myTextFrame.parentStory;
//Split Column
with(myStory.paragraphs[0]) {
    spanColumnType = SpanColumnTypeOptions.splitColumns;
    spanSplitColumnCount = 2;
    splitColumnOutsideGutter = 0;
    splitColumnInsideGutter = 1;
}
//Span Columns
var mySpanIndex = Math.floor(myStory.paragraphs.length / 2);
with(myStory.paragraphs[mySpanIndex]) {
    spanColumnType = SpanColumnTypeOptions.spanColumns;
    spanSplitColumnCount = SpanColumnCountOptions.all;
}
```

Setting Text Preferences

The following script shows how to set general text preferences (for the complete script, see TextPreferences):

```
//The following sets the text preferences for the application; to set the
//text preferences for the front-most document, replace "app.textPreferences" with
//"app.documents.item(0).textPreferences"
with(app.textPreferences){
    abutTextToTextWrap = true;
    //baseline shift key increment can range from .001 to 200 points.
    baselineShiftKeyIncrement = 1;
    highlightCustomSpacing = false;
    highlightHjViolations = true;
    highlightKeeps = true;
    highlightSubstitutedFonts = true;
    highlightSubstitutedGlyphs = true;
    justifyTextWraps = true;
    //kerning key increment value is 1/1000 of an em.
    kerningKeyIncrement = 10;
    //leading key increment value can range from .001 to 200 points.
    leadingKeyIncrement= 1;
    linkTextFilesWhenImporting = false;
    scalingAdjustsText = false;
    showInvisibles = true;
    smallCap = 60;
    subscriptPosition = 30;
```

```
    subscriptSize = 60;
    superscriptPosition = 30;
    superscriptSize = 60;
    typographersQuotes = false;
    useOpticalSize = false;
    useParagraphLeading = false;
    zOrderTextWrap = false;
}
//Text editing preferences are application-wide.
with(app.textEditingPreferences){
    allowDragAndDropTextInStory = true;
    dragAndDropTextInLayout = true;
    smartCutAndPaste = true;
    tripleClickSelectsLine = false;
}
```

Working with Linked Stories

Linked stories make it easier to manager multiple versions of the same story or text content in the same document; it makes it easier to support emerging digital publishing workflows, where, for example, you need to design for vertical and horizontal layouts. Linked stories behave similarly to traditional links. You can designate a story as a parent, and then place the same story at other places in the document as child stories. Whenever the parent story is updated, the child stories are flagged in the Links panel and can be updated to synchronize with the parent story.

Creating linked stories

There are several ways to create a linked story. It can be created by linking existing text frames with the parent story, or by telling the Page, Spread, Master or Document object to create the text frame at the specified location and link it with the parent story.

To link an existing text frame with the parent story, use the following script (for the complete script, see CreateLinkedStories).

```
childTextFrame1.placeAndLink(myTextFrame.parentStory, false);
```

If you specify true for the second parameter, which is optional, the linked story options dialog opens; otherwise, the default options are used.

To tell the Page object to create a linked story for the parent story, the placeAndLink method includes two additional parameters that you can use to specify the layer and point at which to place the linked story, as shown in the following script fragment (for the complete script, see CreateLinkedStories).

```
var newPage = myDocument.pages.add();
newPage.placeAndLink(myTextFrame.parentStory, [originY, originX],
myDocument.activeLayer, false);
```

To tell the Spread object to create linked story for the parent story, the placeAndLink method takes the same parameters as the placeAndLink method for Page object, as shown in the following script fragment (for the complete script, see CreateLinkedStories).

```
var newSpread = myDocument.spreads.add();
newSpread.placeAndLink(myTextFrame.parentStory, [originY, originX],
myDocument.activeLayer, false);
```

You can also call the placeAndLink method for the Document object. The method takes the parent story as the only parameter. This method does not create the linked story; instead, it loads the place gun and lets the user decide where to place the linked story.

Changing linked story options

To change the linked story options, use the following script fragment (for the complete script, see CreateLinkedStories).

```
with (myDocument.linkedStoryOptions)
{
    removeForcedLineBreaks = true;
    updateWhileSaving = true;
    warnOnUpdateOfEditedStory = true;
}
```

Frequently asked questions

How can I tell whether a given link is a link for a linked story?

Use Link.filePath. If it is empty, then it is a link for a linked story.

Given a link for linked story, how can I get its link source?

Use this to select the link source:

```
Link.editoriginal();
```

Then use this to find the active selection, which is the link source:

```
Application.selection();
```

6 Working with Page Items

Chapter Update Status	
CS6	Unchanged

This chapter covers scripting techniques related to the page items (rectangles, ellipses, graphic lines, polygons, text frames, buttons, and groups) that can appear in an InDesign layout.

This document discusses the following:

- Creating page items.
- Page item geometry.
- Working with paths and path points.
- Creating groups.
- Duplicating and moving page items.
- Transforming page items.
- Working with articles.

Creating Page Items

Page items in an InDesign layout are arranged in a hierarchy, and appear within a *container* object of some sort. Spreads, pages, other page items, groups, and text characters are all examples of objects that can contain page items. This hierarchy of containers in the InDesign scripting object model is the same as in the InDesign user interface--when you create a rectangle by dragging the Rectangle tool on a page, you are specifying that the page is the container, or *parent*, of the rectangle. When you paste an ellipse into a polygon, you are specifying that the polygon is the parent of the ellipse, which, in turn, is a *child* object of its parent, a page.

In general, creating a new page item is as simple as telling the object you want to contain the page item to create the page item, as shown in the MakeRectangle script.

```
//Given a page "myPage", create a new rectangle at the default size and location...
var myRectangle = myPage.rectangles.add();
```

In the above script, a new rectangle is created on the first page of a new document. The rectangle appears at the default location (near the upper left corner of the page) and has a default size (around ten points square). Moving the rectangle and changing its dimensions are both accomplished by filling its geometric bounds property with new values, as shown in the MakeRectangleWithProperties script.

```
//Given a page "myPage", create a new rectangle and specify its size and location...
var myRectangle = myPage.rectangles.add({geometricBounds:[72, 72, 144, 144]});
```

Page item types

It is important to note that you cannot create a "generic" page item--you have to create a page item of a specific type (a rectangle, oval, graphic line, polygon, text frame, or button). You will also notice that InDesign changes the type of a page item as the geometry of the page item changes. A rectangle, for example, is always made up of a single, closed path containing four path points and having 90 degree interior angles. Change the location of a single point, however, or add another path, and the type of the page item changes to a polygon. Open the path and remove two of the four points, and InDesign will change the type to a graphic line. The only things that define the type of a rectangle, ellipse, graphic line, or polygon are:

- The number of paths in the object. Any page item with more than one path is a polygon.
- The number and location of points on the first path in the object.

To determine the type of a page item, use this example:

```
var myPageItemType = myPageItem.constructor.name;
```

The result of the above will be a string containing the type of the page item.

Getting the type of a page item

When you have a reference to a generic page item, and want to find out what type of a page item it is, use constructor.name to get the specific type.

```
//Given a generic page item "myPageItem"...
var myType = myPageItem.constructor.name;
alert(myType);
```

Referring to page items

When you refer to page items inside a given container (a document, layer, page, spread, group, text frame, or page item), you use the `pageItems` collection of the container object. This gives you a collection of the top level page items inside the object. For example:

```
var myPageItems = app.documents.item(0).pages.item(0).pageItems;
```

The resulting collection (`myPageItems`) does not include objects inside groups (though it does include the group), objects inside other page items (thought it does contain the parent page item), or page items in text frames. To get a reference to all of the items in a given container, including items nested inside other page items, use the `allPageItems` property.

```
var myAllPageItems = app.documents.item(0).pages.item(0).pageItems;
```

The resulting collection (`myAllPageItems`) includes all objects on the page, regardless of their position in the hierarchy.

Another way to refer to page items is to use their label property, much as you can use the name property of other objects (such as paragraph styles or layers). In the following examples, we will get an array of page items whose label has been set to `myLabel`.

```
var myPageItems = app.documents.item(0).pages.item(0).pageItems("myLabel");
```

If no page items on the page have the specified label, InDesign returns an empty array.

Page-item geometry

If you are working with page items, it is almost impossible to do anything without understanding the way that rulers and measurements work together to specify the location and shape of an InDesign page item. If you use the Control panel in InDesign's user interface, you probably are already familiar with InDesign's geometry, but here is a quick summary:

- Object are constructed relative to the coordinates shown on the rulers.
- Changing the zero point location by either dragging the zero point or by changing the ruler origin changes the coordinates on the rulers.
- Page items are made up of one or more paths, which, in turn, are made up of two or more path points. Paths can be open or closed.
- Path points contain an anchor point (the location of the point itself) and two control handles (left direction, which controls the curve of the line segment preceding the point on the path; and right direction, which controls the curve of the segment following the point). Each of these properties contains an array in the form (`x`, `y`) (where `x` is the horizontal location of the point, and `y` is the vertical location). This array holds the location, in current ruler coordinates, of the point or control handle.

All of the above means that if your scripts need to construct page items, you also need to control the location of the zero point, and you may want to set the measurement units in use.

Working with paths and path points

For most simple page items, you do not need to worry about the paths and path points that define the shape of the object. Rectangles, ellipses, and text frames can be created by specifying their geometric bounds, as we did in the earlier example in this chapter.

In some cases, however, you may want to construct or change the shape of a path by specifying path point locations, you can either set the anchor point, left direction, and right direction of each path point on the path individually (as shown in the DrawRegularPolygon_Slow script), or you can use the entirePath property of the path to set all of the path point locations at once (as shown in the DrawRegularPolygon_Fast script). The latter approach is much faster.

The items in the array you use for the entirePath property can contain anchor points only, or a anchor points and control handles. Here is an example array containing only anchor point locations:

```
[[x1, y1], [x2, y2], ...]
```

Where `x` and `y` specify the location of the anchor.

Here is an example containing fully-specified path points (i.e., arrays containing the left direction, anchor, and right direction, in that order):

```
[[xL1, YL1], [x1, y1], [xR1, yR1]], [[xL2, YL2], [x2, y2], [xR2, yR2]], ...]
```

Where `xL` and `yL` specify the left direction, `x` and `y` specify the anchor point, and `xR` and `yR` specify the right direction.

You can also mix the two approaches, as shown in the following example:

```
[[[xL1, YL1], [x1, y1], [xR1, yR1]], [x2, y2], ...]
```

Note that the original path does not have to have the same number of points as you specify in the array—InDesign will add or subtract points from the path as it applies the array to the entirePath property.

The AddPathPoint script shows how to add path points to a path without using the entirePath property.

```
//Given a graphic line "myGraphicLine"...
var myPathPoint = myGraphicLine.paths.item(0).pathPoints.add();
//Move the path point to a specific location.
myPathPoint.anchor = [144, 144];
```

The DeletePathPoint script shows how to delete a path point from a path.

```
//Given a polygon "myPolygon", remove the
//last path point in the first path.
myPolygon.paths.item(0).pathPoints.item(-1).remove();
```

Grouping Page Items

In the InDesign user interface, you create groups of page items by selecting them and then choosing Group from the Object menu (or by pressing the corresponding keyboard shortcut). In InDesign scripting, you tell the object containing the page items you want to group (usually a page or spread) to group the page items, as shown in the Group script.

```
//Given a page "myPage" containing at least two ovals and two rectangles...
var myArray = new Array;
//Add the items to the array.
myArray.push(myPage.rectangles.item(0));
myArray.push(myPage.ovals.item(0));
myArray.push(myPage.rectangles.item(1));
myArray.push(myPage.ovals.item(1));
//Group the items.
myPage.groups.add(myArray);
```

To ungroup, you tell the group itself to ungroup, as shown in the Ungroup script.

```
//Given a group "myGroup"...
myPageItems = myGroup.ungroup();
```

There is no need to ungroup a group to change the shape, formatting, or content of the page items in the group. Instead, simply get a reference to the page item you want to change, just as you would with any other page item.

Duplicating and Moving Page Items

In the InDesign user interface, you can move page items by selecting them and dragging them to a new location. You can also create copies of page items by copying and pasting, by holding down Option/Alt as you drag an object, or by choosing Duplicate, Paste In Place, or Step and Repeat from the Edit menu. In InDesign scripting, you can use the move method to change the location of page items, and the duplicate method to create a copy of a page item (and, optionally, move it to another location).

The move method can take one of two optional parameters: moveTo and moveBy. Both parameters consist of an array of two measurement units, consisting of a horizontal value and a vertical value. moveTo specifies an absolute move to the location specified by the array, relative to the current location of the zero point. moveBy specifies how far to move the page item relative to the current location of the page item itself. The Move script shows the difference between these two approaches.

```
//Given a reference to a rectangle "myRectangle"...
//Move the rectangle to the location (12, 12).
//Absolute move:
myRectangle.move([12, 12]);
//Move the rectangle *by* 12 points horizontally, 12 points vertically.
//Relative move (note undefined first parameter):
myRectangle.move(undefined, [12, 12]);
//Move the rectangle to another page (rectangle appears at (0,0);
var myPage = app.documents.item(0).pages.add();
myRectangle.move(myPage);
//To move a page item to another document, use the duplicate method.
```

Note that the move method truly moves the object—when you move a page item to another document, it is deleted from the original document. To move the object to another while retaining the original, use the duplicate method (see below).

Use the duplicate method to create a copy of a page item. By default, the duplicate method creates a "clone" of an object in the same location as the original object. Optional parameters can be used with the duplicate method to move the duplicated object to a new location (including other pages in the same document, or to another document entirely).

```
//Given a reference to a rectangle "myRectangle"...
//Duplicate the rectangle and move the
//duplicate to the location (12, 12).
//Absolute move:
var myDuplicate = myRectangle.duplicate([12, 12]);
//Duplicate the rectangle and move the duplicate *by* 12
//points horizontally, 12 points vertically.
//Relative move (note undefined first parameter):
var myDuplicate = myRectangle.duplicate(undefined, [12, 12]);
//Duplicate the rectangle to another page (rectangle appears at (0,0).
var myPage = app.documents.item(0).pages.add();
var myDuplicate = myRectangle.duplicate(myPage);
//Duplicate the rectangle to another document.
var myDocument = app.documents.add();
var myDuplicate = myRectangle.duplicate(myDocument.pages.item(0));
```

You can also use copy and paste in InDesign scripting, but scripts using on these methods require that you select objects (to copy) and rely on the current view to set the location of the pasted elements (when you paste). This means that scripts that use copy and paste tend to be more fragile (i.e., more likely to fail) than scripts that use duplicate and move. Whenever possible, try to write scripts that do not depend on the current view or selection state.

Creating compound paths

InDesign can combine the paths of two or more page items into a single page item containing multiple paths using the Object > Paths > Make Compound Path menu option. You can do this in InDesign scripting using the makeCompoundPath method of a page item, as shown in the following script fragment (for the complete script, refer to the MakeCompoundPath script).

```
//Given a rectangle "myRectangle" and an Oval "myOval"...
myRectangle.makeCompoundPath(myOval);
```

When you create a compound path, regardless of the types of the objects used to create the compound path, the type of the resulting object is polygon.

To release a compound path and convert each path in the compound path into a separate page item, use the releaseCompoundPath method of a page item, as shown in the following script fragment (for the complete script, refer to the ReleaseCompoundPath script).

```
//Given a polygon "myPolygon"...
var myPageItems = myPolygon.releaseCompoundPath();
```

Using Pathfinder operations

The InDesign Pathfinder features offer ways to work with relationships between page items on an InDesign page. You can merge the paths of page items, or subtract the area of one page item from another page item, or create a new page item from the area of intersection of two or more page items. Every page item supports the following methods related to the Pathfinder features: AddPath, ExcludeOverlapPath, IntersectPath, MinusBack, and SubtractPath.

All of the Pathfinder methods work the same way--you provide an array of page items to use as the basis for the operation (just as you select a series of page items before choosing the Pathfinder operation in the user interface).

Note that it is very likely that the type of the object will change after you apply one of the Pathfinder operations. Which object type it will change to depends on the number and location of the points in the path or paths resulting from the operation.

To merge two page items into a single page item, for example, you would use something like the approach shown in the following fragment (for the complete script, refer to AddPath).

```
//Given a rectangle "myRectangle" and an Oval "myOval"...
myRectangle.addPath(myOval);
```

The excludeOverlapPath method creates a new path based on the non-intersecting areas of two or more overlapping page items, as shown in the following script fragment (for the complete script, refer to ExcludeOverlapPath).

```
//Given a rectangle "myRectangle" and an Oval "myOval"...
myRectangle.excludeOverlapPath(myOval);
```

The intersectPath method creates a new page item from the area of intersection of two or more page items, as shown in the following script fragment (for the complete script, refer to IntersectPath).

```
//Given a rectangle "myRectangle" and an Oval "myOval"...
myRectangle.intersect(myOval);
```

The minusBack method removes the area of intersection of the back-most object from the page item or page items in front of it, as shown in the following script fragment (for the complete script, refer to MinusBack).

```
//Given a rectangle "myRectangle" and an Oval "myOval"...
myRectangle.minusBack(myOval);
```

The subtractPath method removes the area of intersection of the frontmost object from the page item or page items behind it, as shown in the following script fragment (for the complete script, refer to SubtractPath).

```
//Given a rectangle "myRectangle" and an Oval "myOval"...
myOval.subtractPath(myRetangle);
```

Converting page-item shapes

InDesign page items can be converted to other shapes using the options in the Object > Convert Shape menu or the Pathfinder panel (Window > Object and Layout > Pathfinder). In InDesign scripting, page items support the convertShape method, as demonstrated in the following script fragment (for the complete script, refer to ConvertShape).

```
//Given a rectangle "myRectangle"...
myRectangle.convertShape(ConvertShapeOptions.convertToRoundedRectangle);
```

The convertShape method also provides a way to open or close reverse paths, as shown in the following script fragment (for the complete script, refer to OpenPath).

```
//Given a rectangle "myRectangle"...
myRectangle.convertShape(ConvertShapeOptions.convertToOpenPath);
```

Arranging page items

Page items in an InDesign layout can be arranged in front of or behind each other by adjusting their stacking order within a layer, or can be placed on different layers. The following script fragment shows how to bring objects to the front or back of their layer, and how to control the stacking order of objects relative to each other (for the complete script, refer to StackingOrder).

```
//Given a rectangle "myRectangle" and an oval "myOval",
//where "myOval" is in front of "myRectangle", bring
//the rectangle to the front...
myRectangle.bringToFront();
```

When you create a page item, you can specify its layer, but you can also move a page item from one layer to another. The item layeritemLayerItemLayer property of the page item is the key to doing this, as shown in the following script fragment (for the complete script, refer to ItemLayer).

```
//Given a rectangle "myRectangle" and a layer "myLayer",
//send the rectangle to the layer...
myRectangle.itemLayer = app.Documents.item(0).layers.item("myLayer");
```

The stacking order of layers in a document can also be changed using the move method of the layer itself, as shown in the following script fragment (for the complete script, refer to MoveLayer).

```
//Given a layer "myLayer", move the layer behind
//the default layer (the lowest layer in the document
//is layers.item(-1).
myLayer.move(LocationOptionsafter, app.documents.item(0).layers.item(-1));
```

Transforming Page Items

Transformations include scaling, rotation, shearing (skewing), and movement (or translation). In scripting, you apply transformations using the `transform` method. This one method replaces the `resize`, `rotate`, and `shear` methods used in versions of InDesign prior to InDesign CS3 (5.0).

Using the transform method

The `transform` method requires a transformation matrix (`transformationMatrix`) object that defines the transformation or series of transformations to apply to the object. A transformation matrix can contain any combination of scale, rotate, shear, or translate operations.

The order in which transformations are applied to an object is important. Applying transformations in differing orders can produce very different results.

To transform an object, you follow two steps:

1. Create a transformation matrix.
2. Apply the transformation matrix to the object using the `transform` method. When you do this, you also specify the coordinate system in which the transformation is to take place. For more on coordinate systems, see "Coordinate spaces" on page 119. In addition, you specify the center of transformation, or transformation origin. For more on specifying the transformation origin, see "Transformation origin" on page 120.

The following scripting example demonstrates the basic process of transforming a page item. (For the complete script, see TransformExamples.)

```
//Rotate a rectangle "myRectangle" around its center point.
var myRotateMatrix =
app.transformationMatrices.add({counterclockwiseRotationAngle:27});
myRectangle.transform(CoordinateSpaces.pasteboardCoordinates,
AnchorPoint.centerAnchor, myRotateMatrix);
//Scale a rectangle "myRectangle" around its center point.
var myScaleMatrix = app.transformationMatrices.add({horizontalScaleFactor:.5,
verticalScaleFactor:.5});
myRectangle.transform(CoordinateSpaces.pasteboardCoordinates,
AnchorPoint.centerAnchor, myScaleMatrix);
//Shear a rectangle "myRectangle" around its center point.
var myShearMatrix =app.transformationMatrices.add({clockwiseShearAngle:30});
myRectangle.transform(CoordinateSpaces.pasteboardCoordinates,
AnchorPoint.centerAnchor, myShearMatrix);
//Rotate a rectangle "myRectangle" around a specified ruler point ([72, 72]).
var myRotateMatrix =
app.transformationMatrices.add({counterclockwiseRotationAngle:27});
myRectangle.transform(CoordinateSpaces.pasteboardCoordinates, [[72, 72],
AnchorPoint.topLeftAnchor], myRotateMatrix, undefined, true);
//Scale a rectangle "myRectangle" around a specified ruler point ([72, 72]).
var myScaleMatrix = app.transformationMatrices.add({horizontalScaleFactor:.5,
verticalScaleFactor:.5});
myRectangle.transform(CoordinateSpaces.pasteboardCoordinates, [[72, 72],
AnchorPoint.topLeftAnchor], myScaleMatrix, undefined, true);
```

For a script that "wraps" transformation routines in a series of easy-to-use functions, refer to the Transform script.

Working with transformation matrices

A transformation matrix cannot be changed once it has been created, but a variety of methods can interact with the transformation matrix to create a new transformation matrix based on the existing transformation matrix. In the following examples, we show how to apply transformations to a transformation matrix and replace the original matrix. (For the complete script, see TransformMatrix.)

```
//Scale a transformation matrix by 50% in both horizontal and vertical dimensions.
var myTransformationMatrix = myTransformationMatrix.scaleMatrix(.5, .5);
//Rotate a transformation matrix by 45 degrees.
myTransformationMatrix = myTransformationMatrix.rotateMatrix(45);
//Shear a transformation matrix by 15 degrees.
myTransformationMatrix = myTransformationMatrix.shearMatrix(15);
```

When you use the `rotateMatrix` method, you can use a sine or cosine value to transform the matrix, rather than an angle in degrees, as shown in the RotateMatrix script.

```
//The following statements are equivalent
//(0.25881904510252 is the sine of 15 degrees; 0.96592582628907, the cosine).
myTransformationMatrix = myTransformationMatrix.rotateMatrix(15);
myTransformationMatrix = myTransformationMatrix.rotateMatrix(undefined,
0.96592582628907);
myTransformationMatrix = myTransformationMatrix.rotateMatrix(undefined, undefined,
0.25881904510252);
```

When you use the `shearMatrixmethod`, you can provide a slope, rather than an angle in degrees, as shown in the ShearMatrix script.

```
//The following statements are equivalent. slope = rise/run--so
//the slope of 45 degrees is 1.
myTransformationMatrix = myTransformationMatrix.shearMatrix(45);
myTransformationMatrix = myTransformationMatrix.shearMatrix(undefined, 1);
```

You can get the inverse of a transformation matrix using the `invertMatrix`method, as shown in the following example. (For the complete script, see InvertMatrix.) You can use the inverted transformation matrix to undo the effect of the matrix.

```
var myRectangle = app.documents.item(0).pages.item(0).rectangles.item(0);
var myTransformationMatrix =
app.transformationMatrices.add({counterclockwiseRotationAngle:30,
horizontalTranslation:12, verticalTranslation:12});
myRectangle.transform(CoordinateSpaces.pasteboardCoordinates,
AnchorPoint.centerAnchor, myTransformationMatrix);
var myNewRectangle = myRectangle.duplicate();
//Move the duplicated rectangle to the location of the original
//rectangle by inverting, then applying the transformation matrix.
myTransformationMatrix = myTransformationMatrix.invertMatrix();
myRectangle.transform(CoordinateSpaces.pasteboardCoordinates,
AnchorPoint.centerAnchor, myTransformationMatrix);
```

You can add transformation matrices using the `catenateMatrix`method, as shown in the following example. (For the complete script, see CatenateMatrix.)

```
var myTransformationMatrixA =
app.transformationMatrices.add({counterclockwiseRotationAngle:30});
var myTransformationMatrixB =
app.transformationMatrices.add({horizontalTranslation:12, verticalTranslation:12});
var myRectangle = app.documents.item(0).pages.item(0).rectangles.item(-1);
var myNewRectangle = myRectangle.duplicate();
    //Rotate the duplicated rectangle.
myNewRectangle.transform(CoordinateSpaces.pasteboardCoordinates,
AnchorPoint.centerAnchor, myTransformationMatrixA);
myNewRectangle = myRectangle.duplicate();
//Move the duplicate (unrotated) rectangle.
myNewRectangle.transform(CoordinateSpaces.pasteboardCoordinates,
AnchorPoint.centerAnchor, myTransformationMatrixB);
//Merge the two transformation matrices.
myTransformationMatrix =
myTransformationMatrixA.catenateMatrix(myTransformationMatrixB);
myNewRectangle = myRectangle.duplicate();
//The duplicated rectangle will be both moved and rotated.
myNewRectangle.transform(CoordinateSpaces.pasteboardCoordinates,
AnchorPoint.centerAnchor, myTransformationMatrix);
```

When an object is transformed, you can get the transformation matrix that was applied to it, using the `transformValuesOf` method, as shown in the following script fragment. (For the complete script, see TransformValuesOf.)

```
//Note that transformValuesOf() always returns an array
//containing a single transformationMatrix.
var myTransformArray =
myRectangle.transformValuesOf(CoordinateSpaces.parentCoordinates);
var myTransformationMatrix = myTransformArray[0];
var myRotationAngle = myTransformationMatrix.counterclockwiseRotationAngle;
var myShearAngle = myTransformationMatrix.clockwiseShearAngle;
var myXScale = myTransformationMatrix.horizontalScaleFactor;
var myYScale = myTransformationMatrix.verticalScaleFactor;
var myXTranslate = myTransformationMatrix.horizontalTranslation;
var myYTranslate = myTransformationMatrix.verticalTranslation;
var myString = "Rotation Angle: " + myRotationAngle + "\r";
myString += "Shear Angle: " + myShearAngle + "\r";
myString += "Horizontal Scale Factor: " + myXScale + "\r";
myString += "Vertical Scale Factor: " + myYScale + "\r";
myString += "Horizontal Translation: " + myXTranslate + "\r";
myString += "Vertical Translation: " + myYTranslate + "\r";
alert(myString);
```

NOTE: The values in the horizontal- and vertical-translation fields of the transformation matrix returned by this method are the location of the upper-left anchor of the object, in pasteboard coordinates.

Coordinate spaces

In the transformation scripts we presented earlier, you might have noticed the `CoordinateSpaces.pasteboardCoordinates` enumeration provided as a parameter for the transform method. This parameter determines the system of coordinates, or *coordinate space*, in which the transform operation occurs. The coordinate space can be one of the following values:

- **`CoordinateSpaces.pasteboardCoordinates`** is the coordinate space of the entire InDesign document. This coordinate space extends behind all spreads in a document. It does not correspond to InDesign's rulers or zero point, nor does it have anything to do with the pasteboard area you can see around pages in the InDesign user interface. Transformations applied to objects have no effect on this coordinate space (e.g., the angle of the horizontal and vertical axes do not change).

- **`CoordinateSpaces.parentCoordinates`** is the coordinate space of the parent of the object. Any transformations applied to the parent affect the parent coordinates; for example, rotating the parent object changes the angle of the horizontal and vertical axes of this coordinate space. In this case, the parent object refers to the group or page item containing the object; if the parent of the object is a page or spread, parent coordinates are the same as spread coordinates.

- **`CoordinateSpaces.innerCoordinates`** is the coordinate space in which the object itself was created.

- **`CoordinateSpaces.spreadCoordinates`** is the coordinate space of the spread. The origin of this space is at the center of the spread, and does not correspond to the rulers you see in the user interface.

The following script shows the differences between the coordinate spaces. (For the complete script, see CoordinateSpaces.)

```
var myRectangle =
app.documents.item(0).pages.item(0).groups.item(-1).rectangles.item(0);
alert("The page contains a group which has been\rrotated 45 degrees
(counterclockwise).\rThe rectangle inside the group was\rrotated 45 degrees
counterclockwise\rbefore it was added to the group.\r\rWatch as we apply a series of
scaling\roperations in different coordinate spaces.");
var myTransformationMatrix =
app.transformationMatrices.add({horizontalScaleFactor:2});
//Transform the rectangle using inner coordinates.
myRectangle.transform(CoordinateSpaces.innerCoordinates, AnchorPoint.centerAnchor,
myTransformationMatrix);
//Select the rectangle and display an alert.
app.select(myRectangle);
alert("Transformed by inner coordinates.");
//Undo the transformation.
app.documents.item(0).undo();
//Transform using parent coordinates.
myRectangle.transform(CoordinateSpaces.parentCoordinates, AnchorPoint.centerAnchor,
myTransformationMatrix);
app.select(myRectangle);
alert("Transformed by parent coordinates.");
app.documents.item(0).undo();
//Transform using pasteboard coordinates.
myRectangle.transform(CoordinateSpaces.pasteboardCoordinates,
AnchorPoint.centerAnchor, myTransformationMatrix);
app.select(myRectangle);
alert("Transformed by pasteboard coordinates.");
app.documents.item(0).undo();
```

Transformation origin

The transformation origin is the center point of the transformation. The transformation origin can be specified in several ways:

- Bounds space:
  - anchor — An anchor point on the object itself.

    ```
    AnchorPoint.centerAnchor
    ```

  - anchor, bounds type — An anchor point specified relative to the geometric bounds of the object (`BoundingBoxLimits.geometricPathBounds`) or the visible bounds of the object (`BoundingBoxLimits.outerStrokeBounds`).

    ```
    [AnchorPoint.bottomLeftAnchor, BoundingBoxLimits.outerStrokeBounds]
    ```

  - anchor, bounds type, coordinate system — An anchor point specified as the geometric bounds of the object (`BoundingBoxLimits.geometricPathBounds`) or the visible bounds of the object (`BoundingBoxLimits.outerStrokeBounds`) in a given coordinate space.

    ```
    [AnchorPoint.bottomLeftAnchor, BoundingBoxLimits.outerStrokeBounds,
    CoordinateSpaces.pasteboardCoordinates]
    ```

  - (x,y), bounds type — A point specified relative to the geometric bounds of the object (`BoundingBoxLimits.geometricPathBounds`) or the visible bounds of the object (`BoundingBoxLimits.outerStrokeBounds`). In this case, the top-left corner of the bounding box is (0, 0); the bottom-right corner, (1, 1). The center anchor is located at (.5, .5).

    ```
    [[.5, .5], BoundingBoxLimits.outerStrokeBounds]
    ```

- (x, y), bounds type, coordinate space — A point specified relative to the geometric bounds of the object (`BoundingBoxLimits.geometricPathBounds`) or the visible bounds of the object (`BoundingBoxLimits.outerStrokeBounds`) in a given coordinate space. In this case, the top-left corner of the bounding box is (0, 0); the bottom-right corner, (1, 1). The center anchor is located at (.5, .5).

```
[[.5, .5], BoundingBoxLimits.outerStrokeBounds,
CoordinateSpaces.pasteboardCoordinates]
```

- Ruler space:
  - (x, y), page index — A point, relative to the ruler origin on a specified page of a spread.

```
[[72, 144], 0]
```

  - (x, y), location — A point, relative to the parent page of the specified location of the object. Location can be specified as an anchor point or a coordinate pair. It can be specified relative to the object's geometric or visible bounds, and it can be specified in a given coordinate space.

```
[[72, 144], AnchorPoint.centerAnchor]
```

- Transform space:
  - (x, y) — A point in the pasteboard coordinate space.

```
[72, 72]
```

  - (x, y), coordinate system — A point in the specified coordinate space.

```
[[72, 72], CoordinateSpaces.parentCoordinates]
```

  - ((x, y)) — A point in the coordinate space given as the `in` parameter of the `transform` method.

```
[[72, 72]]
```

The following script example shows how to use some of the transformation origin options. (For the complete script, see TransformationOrigin.)

```
//Rotate around the duplicated rectangle's center point.
myNewRectangle.transform(CoordinateSpaces.pasteboardCoordinates,
AnchorPoint.centerAnchor, myTransformationMatrix);
//Rotate the rectangle around the ruler location [-100, -100].
//Note that the anchor point specified here specifes the page
//containing the point--*not* that transformation point itself.
//The transformation gets the ruler coordinate [-100, -100] based
//on that page. Setting the considerRulerUnits parameter to true makes
//certain that the transformation uses the current ruler units.
myNewRectangle.transform(CoordinateSpaces.pasteboardCoordinates, [[-100, -100],
AnchorPoint.topLeftAnchor], myTransformationMatrix, undefined, true);
```

Resolving locations

Sometimes, you need to get the location of a point specified in one coordinate space in the context of another coordinate space. To do this, use the `resolve` method, as shown in the following script example. (For the complete script, see ResolveLocation.)

```
var myPageLocation = myRectangle.resolve([[72, 72], AnchorPoint.topRightAnchor],
CoordinateSpaces.pasteboardCoordinates, true);
//resolve() returns an array containing a single item.
alert("X: " + myPageLocation[0][0] + "\rY: " + myPageLocation[0][1]);
```

Transforming points

You can transform points as well as objects, which means scripts can perform a variety of mathematical operations without having to include the calculations in the script itself. The ChangeCoordinates sample script shows how to draw a series of regular polygons using this approach:

```
//General purpose routine for drawing regular polygons from their center point.
function myDrawPolygon(myParent, myCenterPoint, myNumberOfPoints, myRadius,
myStarPolygon, myStarInset){
    var myTransformedPoint;
    var myPathPoints = new Array;
    var myPoint = [0,0];
    if(myStarPolygon == true){
        myNumberOfPoints = myNumberOfPoints * 2;
    }
    var myInnerRadius = myRadius * myStarInset;
    var myAngle = 360/myNumberOfPoints;
    var myRotateMatrix = app.transformationMatrices.add({
    counterclockwiseRotationAngle:myAngle});
    var myOuterTranslateMatrix = app.transformationMatrices.add({
    horizontalTranslation:myRadius});
    var myInnerTranslateMatrix = app.transformationMatrices.add({
    horizontalTranslation:myInnerRadius});
    for (var myPointCounter = 0; myPointCounter < myNumberOfPoints;
    myPointCounter ++){
        //Translate the point to the inner/outer radius.
        if ((myStarInset == 1)||(myIsEven(myPointCounter)==true)){
            myTransformedPoint = myOuterTranslateMatrix.changeCoordinates(myPoint);
        }
        else{
            myTransformedPoint = myInnerTranslateMatrix.changeCoordinates(myPoint);
        }
        myTransformedPoint = myRotateMatrix.changeCoordinates(myTransformedPoint);
        myPathPoints.push(myTransformedPoint);
        myRotateMatrix = myRotateMatrix.rotateMatrix(myAngle);
    }
    //Create a new polygon.
    var myPolygon = myParent.polygons.add();
    //Set the entire path of the polygon to the array we've created.
    myPolygon.paths.item(0).entirePath = myPathPoints;
    //If the center point is somewhere other than [0,0],
    //translate the polygon to the center point.
    if((myCenterPoint[0] != 0)||((myCenterPoint[1] != 0))){
        var myTranslateMatrix = app.transformationMatrices.add({
        horizontalTranslation:myCenterPoint[0],
        verticalTranslation:myCenterPoint[1]});
        myPolygon.transform(CoordinateSpaces.pasteboardCoordinates,
        AnchorPoint.centerAnchor, myTranslateMatrix);
    }
}
//This function returns true if myNumber is even, false if it is not.
function myIsEven(myNumber){
    var myResult = (myNumber%2)?false:true;
    return myResult;
}
```

You also can use the `changeCoordinates` method to change the positions of curve control points, as shown in the FunWithTransformations sample script.

Transforming again

Just as you can apply a transformation or sequence of transformations again in the user interface, you can do so using scripting. There are four methods for applying transformations again:

- `transformAgain`
- `transformAgainIndividually`
- `transformSequenceAgain`
- `transformSequenceAgainIndividually`

The following script fragment shows how to use `transformAgain`. (For the complete script, see TransformAgain.)

```
var myRectangle = myPage.rectangles.item(0);
var myBounds = myRectangle.geometricBounds;
var myX1 = myBounds[1];
var myY1 = myBounds[0];
var myRectangleA = myPage.rectangles.add({geometricBounds:[myY1-12, myX1-12, myY1+12,
myX1+12]});
var myTransformationMatrix =
app.transformationMatrices.add({counterclockwiseRotationAngle:45});
myRectangleA.transform(CoordinateSpaces.pasteboardCoordinates,
AnchorPoint.centerAnchor, myTransformationMatrix);
var myRectangleB = myRectangleA.duplicate();
myRectangleB.transform(CoordinateSpaces.pasteboardCoordinates, [[0,0],
AnchorPoint.topLeftAnchor], myTransformationMatrix, undefined, true);
var myRectangleC = myRectangleB.duplicate();
myRectangleC.transformAgain();
var myRectangleD = myRectangleC.duplicate();
myRectangleD.transformAgain();
var myRectangleE = myRectangleD.duplicate();
myRectangleE.transformAgain();
var myRectangleF = myRectangleE.duplicate();
myRectangleF.transformAgain();
var myRectangleG = myRectangleF.duplicate();
myRectangleG.transformAgain();
var myRectangleH = myRectangleG.duplicate();
myRectangleH.transformAgain();
myRectangleB.transform(CoordinateSpaces.pasteboardCoordinates,
AnchorPoint.centerAnchor, myTransformationMatrix);
myRectangleD.transformAgain();
myRectangleF.transformAgain();
myRectangleH.transformAgain();
```

Resize and Reframe

In addition to scaling page items using the transform method, you can also change the size of the shape using two other methods: resize and reframe. These methods change the location of the path points of the page item without scaling the content or stroke weight of the page item. The following script fragment shows how to use the resize method. For the complete script, see Resize.

```
//Given a reference to a rectangle "myRectangle"...
var myDuplicate = myRectangle.duplicate();
myDuplicate.resize(CoordinateSpaces.innerCoordinates, AnchorPoint.centerAnchor,
ResizeMethods.multiplyingCurrentDimensionsBy, [2, 2]);
```

The following script fragment shows how to use the reframe method. For the complete script, see Reframe.

```
//Given a reference to a rectangle "myRectangle"...
var myBounds = myRectangle.geometricBounds;
var myX1 = myBounds[1]-72;
var myY1 = myBounds[0]-72;
var myX2 = myBounds[3]+72;
var myY2 = myBounds[2]+72;
myDuplicate = myRectangle.duplicate();
myDuplicate.reframe(CoordinateSpaces.innerCoordinates, [[myY1, myX1],[myY2, myX2]]);
```

Working with Articles

Articles provide designers and production artists an easy way to create relationships among page items. These relationships can be used to define the content to be used while exporting to ePub, HTML or Accessible PDFs; and to define the order of the content. You can create articles from a combination of existing page items within a layout, including images, graphics, or text. Once an article has been created, page items can be added, removed, or re-ordered.

Retrieving an article list

To get the articles in a document, use the following script (for the complete script, see CreateArticle).

```
var myDocument = app.documents.item(0);
var articles = myDocument.articles;
```

Adding new articles

To add a new article to a document, use the following script (for the complete script, see CreateArticle).

```
var article1 = myDocument.articles.add("Article1", true);
```

Removing an article

To remove an article, tell the article to remove itself as shown in the following script (for the complete script, see RemoveArticle).

```
article.remove();
```

Reordering articles

To reorder articles, tell the article to move the reference article and the location relative to the reference article as shown in the following script (for the complete script, see ReorderArticles).

```
var article1 = articles.add("Article1", true);
var article2 = articles.add("Article2", true);
var article3 = articles.add("Article3", true);
var article4 = articles.add("Article4", true);
//Reverse the order of articles.
//Move article4 to the first.
article4.move(LocationOptions.AT_BEGINNING);
//Move article1 to the end.
article1.move(LocationOptions.AT_END);
//Move article3 to the second place.
article3.move(LocationOptions.AFTER, article4);
//Move article2 to the third place.
article2.move(LocationOptions.BEFORE, article1);
```

Adding new members to an article

To add new members to an article, use the following script (for the complete script, see AddRemoveArticleMember).

```
var myDocument = app.documents.item(0);
//Retrieve the articles for the document.
var articles = myDocument.articles;

//Create new articles
var article1 = articles.add("Article1", true);
var article2 = articles.add("Article2", true);
var article3 = articles.add("Article3", true);

var myPage = myDocument.pages.item(0);

var article1Members = article1.articleMembers;
article1Members.add(myPage.rectangles.item(0));
article1Members.add(myPage.ovals.item(0));

var article2Members = article2.articleMembers;
article2Members.add(myPage.rectangles.item(1));
article2Members.add(myPage.ovals.item(1));

var article3Members = article3.articleMembers;
var myGroup = myPage.groups.item(0);
//Add group as article member.
article3Members.add(myGroup);
```

Reordering members of an article

To reorder members of an article, tell the member to move the reference object and the location relative to the reference object as shown in the following script (for the complete script, see ReorderArticleMembers).

```
//Assume there are four members in the article in the following order: rectangle, oval,
text frame, group
var articleMembers = article.articleMembers;
articleMembers.add(myPage.rectangles.item(0));
articleMembers.add(myPage.ovals.item(0));
articleMembers.add(myPage.textFrames.item(0));
articleMembers.add(myPage.groups.item(0));

articleMembers.item(0).move(LocationOptions.AT_END);
articleMembers.item(2).move(LocationOptions.AT_BEGINNING);
articleMembers.item(1).move(LocationOptions.AFTER, articleMembers.item(2));
```

Removing members from an article

To remove a member from an article, tell the article member to remove itself, as shown in the following script (for the complete script, see AddRemoveArticleMember).

```
var articleMember = article1Members.item(0);
articleMember.remove();
```

7 User Interfaces

Chapter Update Status	
CS6	Unchanged

JavaScript can create dialogs for simple yes/no questions and text entry, but you probably will need to create more complex dialogs for your scripts. InDesign scripting can add dialogs and populate them with common user-interface controls, like pop-up lists, text-entry fields, and numeric-entry fields. If you want your script to collect and act on information entered by you or any other user of your script, use the `dialog` object.

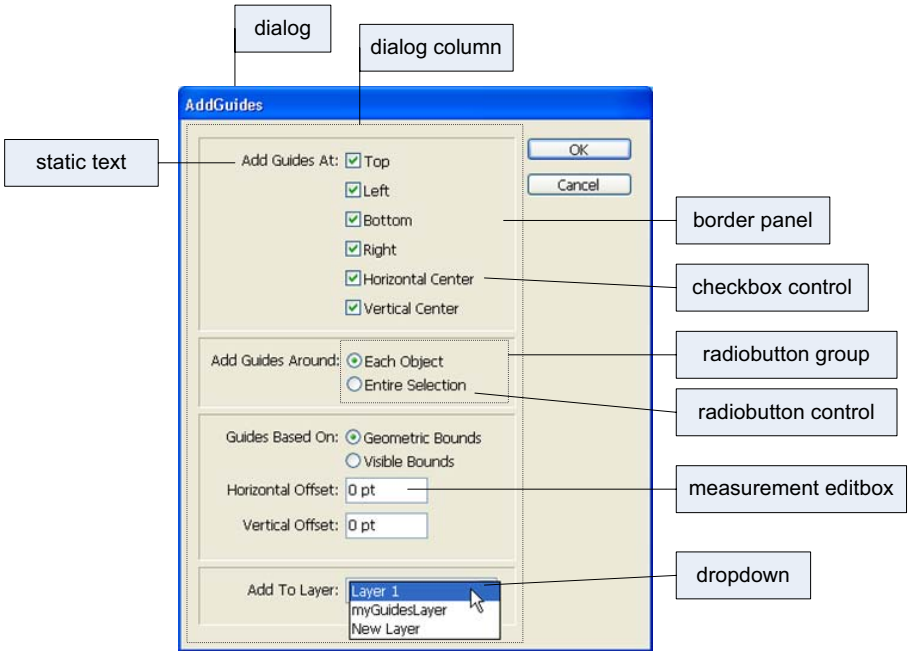
This chapter shows how to work with InDesign dialog scripting. The sample scripts in this chapter are presented in order of complexity, starting with very simple scripts and building toward more complex operations.

NOTE: InDesign scripts written in JavaScript also can include user interfaces created using the Adobe *ScriptUI* component. This chapter includes some ScriptUI scripting tutorials; for more information, see *Adobe JavaScript Tools Guide*.

We assume that you have already read *Adobe InDesign Scripting Tutorial* and know how to create and run a script.

Dialog Overview

An InDesign dialog box is an object like any other InDesign scripting object. The dialog box can contain several different types of elements (known collectively as "widgets"), as shown in the following figure. The elements of the figure are described in the table following the figure.



Dialog box element	InDesign name
Text-edit fields	Text editbox control
Numeric-entry fields	Real editbox, integer editbox, measurement editbox, percent editbox, angle editbox
Pop-up menus	Drop-down control
Control that combines a text-edit field with a pop-up menu	Combo-box control
Check box	Check-box control
Radio buttons	Radio-button control

The `dialog` object itself does not directly contain the controls; that is the purpose of the `dialogColumn` object. `dialogColumns` give you a way to control the positioning of controls within a dialog box. Inside `dialogColumns`, you can further subdivide the dialog box into other `dialogColumns` or `borderPanels` (both of which can, if necessary, contain more `dialogColumns` and `borderPanels`).

Like any other InDesign scripting object, each part of a dialog box has its own properties. A `checkboxControl`, for example, has a property for its text (`staticLabel`) and another property for its state (`checkedState`). The `dropdown` control has a property (`stringList`) for setting the list of options that appears on the control's menu.

To use a dialog box in your script, create the `dialog` object, populate it with various controls, display the dialog box, and then gather values from the dialog-box controls to use in your script. Dialog boxes remain in InDesign's memory until they are destroyed. This means you can keep a dialog box in memory and have data stored in its properties used by multiple scripts, but it also means the dialog boxes take up memory and should be disposed of when they are not in use. In general, you should destroy a dialog-box object before your script finishes executing.

Your First InDesign Dialog

The process of creating an InDesign dialog is very simple: add a dialog, add a dialog column to the dialog, and add controls to the dialog column. The following script demonstrates the process (for the complete script, see SimpleDialog):

```
var myDialog = app.dialogs.add({name:"Simple Dialog"});
//Add a dialog column.
with(myDialog.dialogColumns.add()){
    staticTexts.add({staticLabel:"This is a very simple dialog box."});
}
//Show the dialog box.
var myResult = myDialog.show();
//If the user clicked OK, display one message;
//if they clicked Cancel, display a different message.
if(myResult == true){
    alert("You clicked the OK button.");
}
else{
    alert("You clicked the Cancel button.");
}
//Remove the dialog box from memory.
myDialog.destroy();
```

Adding a User Interface to "Hello World"

In this example, we add a simple user interface to the Hello World tutorial script presented in *Adobe InDesign Scripting Tutorial*. The options in the dialog box provide a way for you to specify the sample text and change the point size of the text:

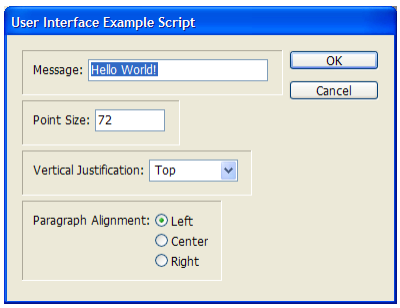
```
var myDialog = app.dialogs.add({name:"Simple User Interface Example
Script",canCancel:true});
with(myDialog){
    //Add a dialog column.
    with(dialogColumns.add()){
        //Create a text edit field.
        var myTextEditField = textEditboxes.add({editContents:"Hello World!",
minWidth:180});
        //Create a number (real) entry field.
        var myPointSizeField = measurementEditboxes.add({editValue:72,
editUnits:MeasurementUnits.points});
    }
}
//Display the dialog box.
var myResult = myDialog.show();
if(myResult == true){
    //Get the values from the dialog box controls.
    var myString = myTextEditField.editContents;
    var myPointSize = myPointSizeField.editValue;
    //Remove the dialog box from memory.
    myDialog.destroy();
    myMakeDocument(myString, myPointSize);
}
else{
    myDialog.destroy();
}
```

Here is the `myMakeDocument` function referred to in the above fragment:

```
function myMakeDocument(myString, myPointSize){
    //Create a new document.
    var myDocument = app.documents.add()
    with(myDocument){
        //Create a text frame.
        var myTextFrame = pages.item(0).textFrames.add();
        //Resize the text frame to the "live" area of the page
        //(using the function "myGetBounds").
        var myBounds = myGetBounds(myDocument, myDocument.pages.item(0));
        myTextFrame.geometricBounds=myBounds;
        //Enter the text from the dialog box in the text frame.
        myTextFrame.contents=myString;
        //Set the size of the text to the size you entered in the dialog box.
        myTextFrame.texts.item(0).pointSize = myPointSize;
    }
}
```

Creating a More Complex User Interface

In the next example, we add more controls and different types of controls to the sample dialog box. The example creates a dialog box that resembles the following:



For the complete script, see ComplexUI.

```
var myDialog = app.dialogs.add({name:"User Interface Example Script",
canCancel:true});
with(myDialog){
    //Add a dialog column.
    with(dialogColumns.add()){
        //Create a border panel.
        with(borderPanels.add()){
            with(dialogColumns.add()){
                //The following line shows how to set a property as you create an object.
                staticTexts.add({staticLabel:"Message:"});
            }
            with(dialogColumns.add()){
                //The following line shows how to set multiple properties
                //as you create an object.
                var myTextEditField = textEditboxes.add
                ({editContents:"Hello World!", minWidth:180});
            }
        }
        //Create another border panel.
        with(borderPanels.add()){
            with(dialogColumns.add()){
```

```
                staticTexts.add({staticLabel:"Point Size:"});
            }
            with(dialogColumns.add()){
                //Create a number entry field. Note that this field uses editValue
                //rather than editText (as a textEditBox would).
                var myPointSizeField = measurementEditboxes.add({editValue:72});
            }
        }
        //Create another border panel.
        with(borderPanels.add()){
            with(dialogColumns.add()){
                staticTexts.add({staticLabel:"Vertical Justification:"});
            }
            with(dialogColumns.add()){
                //Create a pop-up menu ("dropdown") control.
                var myVerticalJustificationMenu = dropdowns.add
                ({stringList:["Top", "Center", "Bottom"], selectedIndex:0});
            }
        }
        //Create another border panel.
        with(borderPanels.add()){
            staticTexts.add({staticLabel:"Paragraph Alignment:"});
            var myRadioButtonGroup = radiobuttonGroups.add();
            with(myRadioButtonGroup){
                var myLeftRadioButton = radiobuttonControls.add
                ({staticLabel:"Left", checkedState:true});
                var myCenterRadioButton = radiobuttonControls.add
                ({staticLabel:"Center"});
                var myRightRadioButton = radiobuttonControls.add({staticLabel:"Right"});
            }
        }
    }
}
//Display the dialog box.
if(myDialog.show() == true){
    var myParagraphAlignment, myString, myPointSize, myVerticalJustification;
    //If the user didn't click the Cancel button,
    //then get the values back from the dialog box.
    //Get the example text from the text edit field.
    myString = myTextEditField.editContents
    //Get the point size from the point size field.
    myPointSize = myPointSizeField.editValue;
    //Get the vertical justification setting from the pop-up menu.
    if(myVerticalJustificationMenu.selectedIndex == 0){
        myVerticalJustification = VerticalJustification.topAlign;
    }
    else if(myVerticalJustificationMenu.selectedIndex == 1){
        myVerticalJustification = VerticalJustification.centerAlign;
    }
    else{
        myVerticalJustification = VerticalJustification.bottomAlign;
    }
    //Get the paragraph alignment setting from the radiobutton group.
    if(myRadioButtonGroup.selectedButton == 0){
```

```
        myParagraphAlignment = Justification.leftAlign;
    }
    else if(myRadioButtonGroup.selectedButton == 1){
        myParagraphAlignment = Justification.centerAlign;
    }
    else{
        myParagraphAlignment = Justification.rightAlign;
    }
    myDialog.destroy();
    myMakeDocument(myString, myPointSize, myParagraphAlignment,
myVerticalJustification);
}
else{
    myDialog.destroy()
}
```

Here is the `myMakeDocument` function referred to in the above fragment:

```
function myMakeDocument(myString, myPointSize, myParagraphAlignment,
myVerticalJustification){
    //Create a new document.
    var myDocument = app.documents.add();
    with(myDocument){
        viewPreferences.horizontalMeasurementUnits = MeasurementUnits.points;
        viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
        var myPage = pages[0];
        with(myPage){
            //Create a text frame.
            var myTextFrame = pages.item(0).textFrames.add();
            with(myTextFrame){
                //Set the geometric bounds of the frame using the "myGetBounds" function.
                geometricBounds = myGetBounds(myDocument, myPage);
                //Set the contents of the frame to the string you
                //entered in the dialog box.
                contents = myString;
                //Set the alignment of the paragraph.
                texts.item(0).justification = myParagraphAlignment;
                //Set the point size of the text.
                texts.item(0).pointSize = myPointSize;
                //Set the vertical justification of the text frame.
                textFramePreferences.verticalJustification = myVerticalJustification;
            }
        }
    }
}
```

Working with ScriptUI

JavaScripts can make create and define user-interface elements using an Adobe scripting component named ScriptUI. ScriptUI gives scripters a way to create floating palettes, progress bars, and interactive dialog boxes that are far more complex than InDesign's built-in `dialog` object.

This does not mean, however, that user-interface elements written using Script UI are not accessible to users. InDesign scripts can execute scripts written in other scripting languages using the method.

Creating a progress bar with ScriptUI

The following sample script shows how to create a progress bar using JavaScript and ScriptUI, then use the progress bar from another script (for the complete script, see ProgressBar):

```
#targetengine "session"
//Because these terms are defined in the "session" engine,
//they will be available to any other JavaScript running
//in that instance of the engine.
var myMaximumValue = 300;
var myProgressBarWidth = 300;
var myIncrement = myMaximumValue/myProgressBarWidth;
myCreateProgressPanel(myMaximumValue, myProgressBarWidth);
function myCreateProgressPanel(myMaximumValue, myProgressBarWidth){
    myProgressPanel = new Window('window', 'Progress');
    with(myProgressPanel){
        myProgressPanel.myProgressBar = add('progressbar', [12, 12,
    myProgressBarWidth, 24], 0, myMaximumValue);
    }
}
```

The following script fragment shows how to call the progress bar created in the above script using a separate JavaScript (for the complete script, see CallProgressBar):

```
Rem Create a document and add pages to it--
Rem if you do not do this, the progress bar
Rem will go by too quickly.
Set myDocument = myInDesign.Documents.Add
Rem Note that the JavaScripts must use the "session"
Rem engine for this to work.
myString = "#targetengine ""session""" & vbCr
myString = myString & "myCreateProgressPanel(100, 400);" & vbcr
myString = myString & "myProgressPanel.show();" & vbcr
myInDesign.DoScript myString, idScriptLanguage.idJavascript
For myCounter = 1 to 100
    Rem Add a page to the document.
    myInDesign.Documents.Item(1).Pages.Add
    myString = "#targetengine ""session""" & vbCr
    myString = myString & "myProgressPanel.myProgressBar.value = "
    myString = myString & cstr(myCounter) & "/myIncrement;" & vbcr
    myInDesign.DoScript myString, idScriptLanguage.idJavascript
    If(myCounter = 100) Then
        myString = "#targetengine ""session""" & vbCr
        myString = myString & "myProgressPanel.myProgressBar.value = 0;" & vbcr
        myString = myString & "myProgressPanel.hide();" & vbcr
        myInDesign.DoScript myString, idScriptLanguage.idJavascript
        myDocument.Close idSaveOptions.idNo
    End If
Next
```

8 Events

Chapter Update Status	
CS6	Unchanged

InDesign scripting can respond to common application and document events, such as opening a file, creating a new file, printing, and importing text and graphic files from disk. In InDesign scripting, the `event` object responds to an event that occurs in the application. Scripts can be attached to events using the `eventListener` scripting object. Scripts that use events are the same as other scripts—the only difference is that they run automatically when the corresponding event occurs, rather than being run by the user (from the Scripts palette).

This chapter shows how to work with InDesign event scripting. The sample scripts in this chapter are presented in order of complexity, starting with very simple scripts and building toward more complex operations.

We assume that you have already read *Adobe InDesign Scripting Tutorial* and know how to create, install, and run a script.

For a discussion of events related to menus, see Chapter 9, "Menus."

The InDesign event scripting model is similar to the Worldwide Web Consortium (W3C) recommendation for Document Object Model Events. For more information, see http://www.w3c.org.

Understanding the Event Scripting Model

The InDesign event scripting model consists of a series of objects that correspond to the events that occur as you work with the application. The first object is the `event`, which corresponds to one of a limited series of actions in the InDesign user interface (or corresponding actions triggered by scripts).

To respond to an event, you register an `eventListener` with an object capable of receiving the event. When the specified event reaches the object, the `eventListener` executes the script function defined in its handler function (which can be either a script function or a reference to a script file on disk).

You can view the available events using the Object Model Viewer in the ExtendScript Toolkit. In the ExtendScript Toolkit, select the Event class, then click Class in the Types list to display a list of available event types in the Properties and Methods list.

About event properties and event propagation

When an action—whether initiated by a user or by a script—triggers an event, the event can spread, or *propagate*, through the scripting objects capable of responding to the event. When an event reaches an object that has an `eventListener` registered for that event, the `eventListener` is triggered by the event. An event can be handled by more than one object as it propagates.

There are two types of event propagation:

- **None —** Only the `eventListeners` registered to the event target are triggered by the event. The `beforeDisplay` event is an example of an event that does not propagate.
- **Bubbling** — The event starts propagation at its `target` and triggers any qualifying `eventListeners` registered to the `target`. The event then proceeds upward through the scripting object model, triggering any qualifying `eventListeners` registered to objects above the `target` in the scripting object model hierarchy.

The following table provides more detail on the properties of an `event` and the ways in which they relate to event propagation through the scripting object model.

Property	Description
`Bubbles`	If true, the `event` propagates to scripting objects *above* the object initiating the `event`.
`Cancelable`	If true, the default behavior of the `event` on its `target` can be canceled. To do this, use the `PreventDefault` method .
`CurrentTarget`	The current scripting object processing the `event`. See `target` in this table.
`DefaultPrevented`	If true, the default behavior of the `event` on the current `target` was prevented, thereby canceling the action. See `target` in this table.
`EventPhase`	The current stage of the `event` propagation process.
`EventType`	The type of the `event`, as a string (for example, `"beforeNew"`).
`PropagationStopped`	If true, the `event` has stopped propagating beyond the `current target` (see `target` in this table). To stop event propagation, use the `stopPropagation` method .
`Target`	The object from which the `event` originates. For example, the `target` of a `beforeImport` event is a document; of a `beforeNew` event, the application.
`TimeStamp`	The time and date when the `event` occurred.

Working with Event Listeners

When you create an `eventListener`, you specify the event type and the event handler (as a function or file reference). The following script fragment shows how to add an `eventListener` for a specific event (for the complete script, see AddEventListener).

```
var myEventListener = app.addEventListener("afterNew", myDisplayEventType, false);
```

The preceding script fragment refers to the following function:

```
function myDisplayEventType(myEvent){
    alert("This event is the " + myEvent.eventType + " event.");
}
```

To remove the `eventListener` created by the preceding script, run the following script (from the RemoveEventListener tutorial script):

```
app.removeEventListener("afterNew", myDisplayEventType);
```

When an `eventListener` responds to an event, the event may still be processed by other `eventListeners` that might be monitoring the event (depending on the propagation of the event). For example, the `afterOpen` event can be observed by `eventListeners` associated with both the application and the document.

`eventListeners` do not persist beyond the current InDesign session. To make an `eventListener` available in every InDesign session, add the script to the startup scripts folder. (For more on installing scripts, see "Installing Scripts" in *Adobe InDesign Scripting Tutorial*.) When you add an `eventListener` script to a document, it is not saved with the document or exported to IDML.

NOTE: If you are having trouble with a script that defines an `eventListener`, you can either run a script that removes the `eventListener` or quit and restart InDesign.

`eventListeners` that use handler functions defined inside the script (rather than in an external file) must use `#targetengine "session"`. If the script is run using `#targetengine "main"` (the default), the function is not available when the event occurs, and the script generates an error.

An event can trigger multiple `eventListeners` as it propagates through the scripting object model. The following sample script demonstrates an event triggering `eventListeners` registered to different objects (for the full script, see MultipleEventListeners):

```
#targetengine "session"
main();
function main(){
    var myApplicationEventListener = app.eventListeners.add("beforeImport",
    myEventInfo);
    var myDocumentEventListener = app.documents.item(0).eventListeners.add
    ("beforeImport", myEventInfo);
}
function myEventInfo(myEvent){
    var myString = "Current Target: " + myEvent.currentTarget.name;
    alert(myString);
}
```

When you run the preceding script and place a file, InDesign displays alerts showing, in sequence, the name of the document, then the name of the application. To remove the event listeners added by the preceding script, run the RemoveMultipleEventListeners script.

The following sample script creates an `eventListener` for each document event and displays information about the event in a simple dialog box. For the complete script, see EventListenersOn.

```
main()
function main(){
    app.scriptPreferences.version = 5.0;
    var myEventNames = [
        "beforeQuit", "afterQuit",
        "beforeNew", "afterNew",
        "beforeOpen", "afterOpen",
        "beforeClose", "afterClose",
        "beforeSave", "afterSave",
        "beforeSaveAs", "afterSaveAs",
        "beforeSaveACopy", "afterSaveACopy",
        "beforeRevert", "afterRevert",
        "beforePrint", "afterPrint",
        "beforeExport", "afterExport",
        "beforeImport", "afterImport"
    ] ;
    for (var myCounter = 0; myCounter < myEventNames.length; myCounter ++){
        app.addEventListener(myEventNames[myCounter], myEventInfo);
```

```
    }
}
function myEventInfo(myEvent){
    var myString = "Handling Event: " +myEvent.eventType;
    myString += "\r\rTarget: " + myEvent.target + " " +myEvent.target.name;
    myString += "\rCurrent: " +myEvent.currentTarget + " " +
    myEvent.currentTarget.name;
    myString += "\r\rPhase: " + myGetPhaseName(myEvent.eventPhase );
    myString += "\rBubbles: " + myEvent.bubbles;
    myString += "\r\rCancelable: " +myEvent.cancelable;
    myString += "\rStopped: " +myEvent.propagationStopped;
    myString += "\rCanceled: " +myEvent.defaultPrevented;
    myString += "\r\rTime: " +myEvent.timeStamp;
    alert(myString);
    function myGetPhaseName(myPhase){
        switch(myPhase){
            case EventPhases.atTarget:
                myPhaseName = "At Target";
                break;
            case EventPhases.bubblingPhase:
                myPhaseName = "Bubbling";
                break;
            case EventPhases.done:
                myPhaseName = "Done";
                break;
            case EventPhases.notDispatching:
                myPhaseName = "Not Dispatching";
                break;
        }
        return myPhaseName;
    }
}
```

The following sample script shows how to turn off all `eventListeners` on the application object. For the complete script, see EventListenersOff.

```
#targetengine "session"
app.eventListeners.everyItem().remove();
```

Sample afterNew Event Listener

The `afterNew` event provides a convenient place to add information to the document, such as the user name, the date the document was created, copyright information, and other job-tracking information. The following tutorial script shows how to add this kind of information to a text frame in the slug area of the first master spread in the document (for the complete script, see AfterNew). This script also adds document metadata (also known as file info or XMP information).

```
#targetengine "session"
//Creates an event listener that will run after a new document is created.
main();
function main(){
    var myEventListener = app.eventListeners.add("afterNew", myAfterNewHandler);
}
function myAfterNewHandler(myEvent){
    var myDocument = myEvent.parent;
    myDocument.viewPreferences.horizontalMeasurementUnits =
    MeasurementUnits.points;
    myDocument.viewPreferences.verticalMeasurementUnits =
    MeasurementUnits.points;
    myDocument.viewPreferences.rulerOrigin = RulerOrigin.pageOrigin;
    myCreateSlug(myDocument);
    myAddXMPData(myDocument);
    function myCreateSlug(myDocument){
        //mySlugOffset is the distance from the bottom of the page to
        //the top of the slug.
        var mySlugOffset = 24;
        //mySlugHeight is the height of the slug text frame.
        var mySlugHeight = 72;
            with(myDocument.documentPreferences){
                documentSlugUniformSize = false;
                slugBottomOffset = mySlugOffset + mySlugHeight;
                slugTopOffset = 0;
                slugInsideOrLeftOffset = 0;
                slugRightOrOutsideOffset = 0;
            }
            for(var myCounter = 0; myCounter < myDocument.masterSpreads.length;
            myCounter++){
            var myMasterSpread = myDocument.masterSpreads.item(myCounter);
            for(var myMasterPageCounter = 0; myMasterPageCounter <
            myMasterSpread.pages.length; myMasterPageCounter ++){
                var myPage = myMasterSpread.pages.item(myMasterPageCounter);
                var mySlugBounds = myGetSlugBounds(myDocument, myPage,
                mySlugOffset, mySlugHeight);
                var mySlugFrame = myPage.textFrames.add(
                {geometricBounds:mySlugBounds, contents:"Created: " +
                myEvent.timeStamp + "\rby: " + app.userName});
            }
        }
    }
    function myAddXMPData(myDocument){
        with(myDocument.metadataPreferences){
            author = "Adobe Systems";
            description = "This is a sample document with XMP metadata.";
        }
    }
    function myGetSlugBounds(myDocument, myPage, mySlugOffset, mySlugHeight){
        var myPageWidth = myDocument.documentPreferences.pageWidth;
        var myPageHeight = myDocument.documentPreferences.pageHeight
        //Because "right" and "left" margins become "inside" and "outside"
```

```
        //for pages in a facing pages view, we have to use a special case for
        //left hand pages.
        if(myPage.side == PageSideOptions.leftHand){
            var myX2 = myPageWidth - myPage.marginPreferences.left;
            var myX1 = myPage.marginPreferences.right;
        }
        else{
            var myX1 = myPage.marginPreferences.left;
            var myX2 = myPageWidth - myPage.marginPreferences.right;
        }
        var myY1 = myPageHeight + mySlugOffset;
        var myY2 = myY1 + mySlugHeight;
        return [myY1, myX1, myY2, myX2];
    }
}
```

Sample beforePrint Event Listener

The `beforePrint` event provides a perfect place to execute a script that performs various preflight checks on a document. The following script shows how to add an event listener that checks a document for certain attributes before printing (for the complete script, see BeforePrint):

```
#targetengine "session"
//Adds an event listener that performs a preflight check on a document
//before printing. If the preflight check fails, the script cancels
//the print job.
main();
function main(){
    var myEventListener = app.eventListeners.add("beforePrint",
    myBeforePrintHandler);
}
function myBeforePrintHandler(myEvent){
    //The parent of the event is the document.
    var myDocument = myEvent.parent;
    if(myPreflight(myDocument) == false){
        myEvent.stopPropagation();
        myEvent.preventDefault();
        alert("Document did not pass preflight check.");
    }
    else{alert("Document passed preflight check. Ready to print.");}
    function myPreflight(myDocument){
        var myPreflightCheck = true;
        var myFontCheck = myCheckFonts(myDocument);
        var myGraphicsCheck = myCheckGraphics(myDocument);
        alert("Fonts: " + myFontCheck + "\r" + "Links:" + myGraphicsCheck);
        if((myFontCheck == false)||(myGraphicsCheck == false)){
            myPreflightCheck = false;
        }
        return myPreflightCheck;
    }
    function myCheckFonts(myDocument){
        var myFontCheck = true;
        for(var myCounter = 0; myCounter < myDocument.fonts.length;
            myCounter ++){
            if(myDocument.fonts.item(myCounter).status != FontStatus.installed){
                myFontCheck = false;
            }
```

```
        }
        return myFontCheck;
    }
    function myCheckGraphics(myDocument){
        var myGraphicsCheck = true;
        for(var myCounter = 0; myCounter < myDocument.allGraphics.length;
        myCounter++){
            var myGraphic = myDocument.allGraphics[myCounter];
            if(myGraphic.itemLink.status != LinkStatus.normal){
                myGraphicsCheck = false;
            }
        }
        return myGraphicsCheck;
    }
}
```

Sample Selection Event Listeners

InDesign can respond to events related to selection. When you select or deselect objects in an InDesign document, InDesign generates the `afterSelectionChanged` event. When you change an attribute (the formatting or position) of the selected object or objects, InDesign generates the `afterSelectionAttributeChanged` event. These two events are useful when you want to create a script that responds to user actions.

The following script fragment shows how to get and display the type of an object when the selection changes. For the complete script, see AfterSelectionChanged.

```
var myDocument = app.documents.add();
myDocument.addEventListener("afterSelectionChanged", myDisplaySelectionType);
```

The event handler referred to in the preceding script fragment looks like this:

```
function myDisplaySelectionType(myEvent){
    if(app.documents.length != 0){
        if(app.documents.item(0).selection.length != 0){
            var mySelection = app.documents.item(0).selection;
            var myString = "Selection Contents:\r";
            for(var myCounter = 0; myCounter < mySelection.length; myCounter++){
                myString = myString + mySelection[myCounter].constructor.name + "\r"
            }
            alert(myString);
        }
    }
}
```

To remove the event listener added by the preceding script, run the RemoveAfterSelectionChanged script.

The following script fragment shows how to respond to a change in the attributes of a selection. In this example, the event handler checks the selection to see whether the Registration swatch has been applied. (Accidental application of the Registration swatch can cause problems at your commercial printer.) If the Registration swatch has been applied, the script asks whether the change was intentional. For the complete script, see AfterSelectionAttributeChanged.

```
app.addEvenListener("afterSelectionAttributeChanged", myCheckForRegistration);
```

The event handler referred to in the preceding script fragment looks like this:

```
function myCheckForRegistration(myEvent){
    var myRegistrationSwatchUsed = false;
    if(app.selection.length != 0){
        for(var myCounter = 0; myCounter < app.selection.length; myCounter++){
            if((app.documents.item(0).selection[myCounter].fillColor ==
            app.documents.item(0).swatches.item("Registration"))||
            (app.documents.item(0).selection[myCounter].strokeColor ==
            app.documents.item(0).swatches.item("Registration")){
                myRegistrationSwatchUsed = true;
            }
        }
    }
    if(myRegistrationSwatchUsed == true){
        alert("The Registration swatch is applied to some of the\robjects in the
selection.
        Did you really intend to apply this swatch?");
    }
}
```

To remove the event listener added by the preceding script, run the RemoveAfterSelectionAttributeChanged script.

Sample onIdle Event Listener

InDesign's idle tasks execute when there are no events in the event queue for the application to process. It is easy to run idle tasks by scripting. The `onIdle` event provides a way to run scripting-based idle tasks. It can be used to automatically execute a script when InDesign/InCopy is idle. Its event target is `IdleTask`, and its event object is `IdleEvent`.

The `sleep` property of the idle task is the amount of time that elapses before InDesign calls the task again. It should be obvious that you need to set the sleep time to a value high enough that it does not interfere with your work, though this value will vary depending on what tasks the script performs.

Setting the sleep time to zero deletes the task (though it does not remove the event listener). This is the most convenient way to stop an idle task.

The following script shows how to add an eventListener and show a message box from the idle task (for the complete script, see Reminder):

```
#targetengine "session"
main();
function main()
{
    var myIdleTask = app.idleTasks.add({name:"my_idle_task", sleep:10000});
    var onIdleEventListener = myIdleTask.addEventListener(IdleEvent.ON_IDLE,
onIdleEventHandler, false);
    alert("Created idle task " + myIdleTask.name + "; added event listener on " +
onIdleEventListener.eventType);
}
function onIdleEventHandler(myIdleEvent)
{
    if (app.documents.length == 0)
    {
        var myDoc = app.documents.add();
        alert("Created document " + myDoc.name + " in idle task.");
        return;
    }
```

```
    var myTextFrames = app.activeDocument.pages.item(0).textFrames;
    if (myTextFrames.length == 0)
    {
        var myTextFrame = myTextFrames.add();
        myTextFrame.geometricBounds = ["72pt", "72pt", "288pt", "288pt"];
        myTextFrame.contents = "Text frame created in idle task";
        alert("Created a text frame in idle task.");
        return;
    }

    //Delete idle task by setting its sleep time to zero.
    myIdleEvent.parent.sleep = 0;
    alert("Nothing to do. Delete idle task.");
}
```

To remove the idle task created by preceding script, run the following script (for the complete script, see RemoveIdleTask):

```
#targetengine "session"
main();
function main()
{
    if (app.idleTasks.length == 0)
    {
        alert("There is no idle task.");
    }
    else
    {
        var myIdleTaskName = "my_idle_task";
        var myIdleTask = app.idleTasks.itemByName(myIdleTaskName);
        if (myIdleTask != null)
        {
            myIdleTask.remove();
        }
        else
        {
            alert("There is no idle task " + myIdleTaskName);
        }
    }
}
```

To remove all idle tasks, run the following script (for the complete script, see RemoveAllIdleTasks):

```
#targetengine "session"
main();
function main()
{
    var length = app.idleTasks.length;
    if (length == 0)
    {
        alert("There is no idle task.");
    }
    else
    {
        for (var i = length-1; i >=0; i--)
        {
            app.idleTasks.item(i).remove();
        }
        alert(length + " idle task(s) removed.");
    }
}
```

To list existing idle tasks, run the following script (for the complete script, see ListIdleTasks):

```
#targetengine "session"
main();
function main()
{
    var length = app.idleTasks.length;
    if (length == 0)
    {
        alert("There is no idle task.");
    }
    else
    {
        var str = "";
        for (var i = 0; i < length; i++)
        {
            var myIdleTask = app.idleTasks.item(i);
            str += "idle task " + myIdleTask.id + ": " + myIdleTask.name + "\n";
        }
        alert(str);
    }
}
```

9 Menus

Chapter Update Status	
CS6	Unchanged

InDesign scripting can add menu items, remove menu items, perform any menu command, and attach scripts to menu items.

This chapter shows how to work with InDesign menu scripting. The sample scripts in this chapter are presented in order of complexity, starting with very simple scripts and building toward more complex operations.

We assume that you have already read *Adobe InDesign Scripting Tutorial* and know how to create, install, and run a script.

Understanding the Menu Model

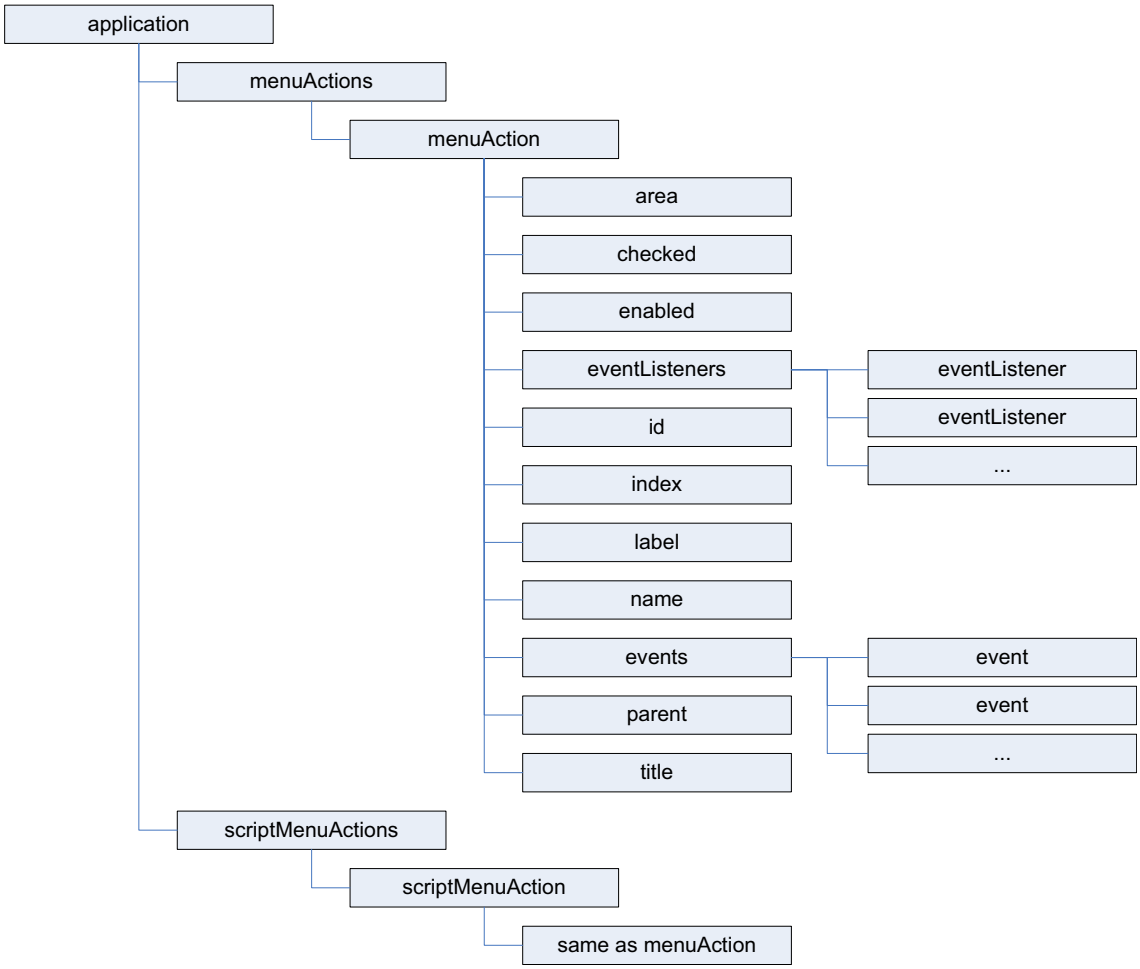
The InDesign menu-scripting model is made up of a series of objects that correspond to the menus you see in the application's user interface, including menus associated with panels as well as those displayed on the main menu bar. A `menu` object contains the following objects:

- `menuItems` — The menu options shown on a menu. This does not include submenus.
- `menuSeparators` — Lines used to separate menu options on a menu.
- `submenus` — Menu options that contain further menu choices.
- `menuElements` — All `menuItems`, `menuSeparators` and `submenus` shown on a menu.
- `eventListeners` — These respond to user (or script) actions related to a menu.
- `events` — The `events` triggered by a menu.

Every `menuItem` is connected to a `menuAction` through the `associatedMenuAction` property. The properties of the `menuAction` define what happens when the menu item is chosen. In addition to the `menuActions` defined by the user interface, InDesign scripters can create their own, `scriptMenuActions`, which associate a script with a menu selection.

A `menuAction` or `scriptMenuAction` can be connected to zero, one, or more `menuItems`.

The following diagram shows how the different menu objects relate to each other:



To create a list (as a text file) of all menu actions, run the following script fragment (from the GetMenuActions tutorial script):

```
var myMenuActionNames = app.menuActions.everyItem().name;
//Open a new text file.
var myTextFile = File.saveDialog("Save Menu Action Names As", undefined);
//If the user clicked the Cancel button, the result is null.
if(myTextFile != null){
    //Open the file with write access.
    myTextFile.open("w");
    for(var myCounter = 0; myCounter < myMenuActionNames.length; myCounter++){
        myTextFile.writeln(myMenuActionNames[myCounter]);
    }
    myTextFile.close();
}
```

To create a list (as a text file) of all available menus, run the following script fragment (for the complete script, see GetMenuNames). These scripts can be very slow, as there are many menu names in InDesign.

```
var myMenu;
//Open a new text file.
var myTextFile = File.saveDialog("Save Menu Action Names As", undefined);
//If the user clicked the Cancel button, the result is null.
if(myTextFile != null){
    //Open the file with write access.
    myTextFile.open("w");
    for(var myMenuCounter = 0;myMenuCounter< app.menus.length; myMenuCounter++){
        myMenu = app.menus.item(myMenuCounter);
        myTextFile.writeln(myMenu.name);
        myProcessMenu(myMenu, myTextFile);
    }
    myTextFile.close();
    alert("done!");
}
function myProcessMenu(myMenu, myTextFile){
    var myMenuElement;
    var myIndent = myGetIndent(myMenu);
    for(var myCounter = 0; myCounter < myMenu.menuElements.length; myCounter++){
        myMenuElement = myMenu.menuElements.item(myCounter);
        if(myMenuElement.getElements()[0].constructor.name != "MenuSeparator"){
            myTextFile.writeln(myIndent + myMenuElement.name);
            if(myMenuElement.getElements()[0].constructor.name == "Submenu"){
                if(myMenuElement.menuElements.length > 0){
                    myProcessMenu(myMenuElement, myTextFile);
                }
            }
        }
    }
}
function myGetIndent(myObject){
    var myString = "\t";
    var myDone = false;
    do{
        if((myObject.parent.constructor.name == "Menu")||
        (myObject.parent.constructor.name == "Application")){
            myDone = true;
        }
        else{
            myString = myString + "\t";
            myObject = myObject.parent;
        }
    }while(myDone == false)
    return myString;
}
```

Localization and menu names

in InDesign scripting, `menuItems`, `menus`, `menuActions`,and `submenus` are all referred to by name. Because of this, scripts need a method of locating these objects that is independent of the installed locale of the application. To do this, you can use an internal database of strings that refer to a specific item, regardless of locale. For example, to get the locale-independent name of a menu action, you can use the following script fragment (for the complete script, see GetKeyStrings):

```
var myString = "";
var myMenuAction = app.menuActions.item("Convert to Note");
var myKeyStrings = app.findKeyStrings(myMenuAction.name);
if(myKeyStrings.constructor.name == "Array"){
    for(var myCounter = 0; myCounter < myKeyStrings.length; myCounter ++){
        myString += myKeyStrings[myCounter] + "\r";
    }
}
else{
    myString = myKeyStrings;
}
alert(myString);
```

NOTE: It is much better to get the locale-independent name of a `menuAction` than of a `menu`, `menuItem`, or `submenu`, because the title of a `menuAction` is more likely to be a single string. Many of the other menu objects return multiple strings when you use the `findKeyStrings` method.

Once you have the locale-independent string you want to use, you can include it in your scripts. Scripts that use these strings will function properly in locales other than that of your version of InDesign.

To translate a locale-independent string into the current locale, use the following script fragment (from the TranslateKeyString tutorial script):

```
var myString = app.translateKeyString("$ID/NotesMenu.ConvertToNote");
alert(myString);
```

Running a Menu Action from a Script

Any of InDesign's built-in `menuActions` can be run from a script. The `menuAction` does not need to be attached to a `menuItem`; however, in every other way, running a `menuItem` from a script is exactly the same as choosing a menu option in the user interface. For example, If selecting the menu option displays a dialog box, running the corresponding `menuAction` from a script also displays a dialog box.

The following script shows how to run a `menuAction` from a script (for the complete script, see InvokeMenuAction):

```
//Get a reference to a menu action.
var myMenuAction = app.menuActions.item("$ID/NotesMenu.ConvertToNote");
//Run the menu action. This example action will fail if you do not
//have text selected.
myMenuAction.invoke();
```

NOTE: In general, you should not try to automate InDesign processes by scripting menu actions and user-interface selections; InDesign's scripting object model provides a much more robust and powerful way to work. Menu actions depend on a variety of user-interface conditions, like the selection and the state of the window. Scripts using the object model work with the objects in an InDesign document directly, which means they do not depend on the user interface; this, in turn, makes them faster and more consistent.

Adding Menus and Menu Items

Scripts also can create new menus and menu items or remove menus and menu items, just as you can in the InDesign user interface. The following sample script shows how to duplicate the contents of a submenu to a new menu in another menu location (for the complete script, see CustomizeMenu):

```
var myMainMenu = app.menus.item("Main");
var myTypeMenu = myMainMenu.menuElements.item("Type");
var myFontMenu = myTypeMenu.menuElements.item("Font");
var myKozukaMenu = myFontMenu.submenus.item("Kozuka Mincho Pro ");
var mySpecialFontMenu = myMainMenu.submenus.add("Kozuka Mincho Pro");
for(myCounter = 0;myCounter < myKozukaMenu.menuItems.length; myCounter++){
    var myAssociatedMenuAction =
myKozukaMenu.menuItems.item(myCounter).associatedMenuAction;
    mySpecialFontMenu.menuItems.add(myAssociatedMenuAction);
}
```

To remove the custom menu item created by the above script, use RemoveCustomMenu.

```
var myMainMenu = app.menus.item("$ID/Main");
try{
    var mySpecialFontMenu = myMainMenu.submenus.item("Kozuka Mincho Pro");
    mySpecialFontMenu.remove();
}catch(myError){}
```

Menus and Events

Menus and submenus generate events as they are chosen in the user interface, and `menuActions` and `scriptMenuActions` generate events as they are used. Scripts can install `eventListeners` to respond to these events. The following table shows the events for the different menu scripting components:

Object	Event	Description
`menu`	`beforeDisplay`	Runs the attached script before the contents of the menu is shown.
`menuAction`	`afterInvoke`	Runs the attached script when the associated `menuItem` is selected, but after the `onInvoke` event.
	`beforeInvoke`	Runs the attached script when the associated `menuItem` is selected, but before the `onInvoke` event.
`scriptMenuAction`	`afterInvoke`	Runs the attached script when the associated `menuItem` is selected, but after the `onInvoke` event.
	`beforeInvoke`	Runs the attached script when the associated `menuItem` is selected, but before the `onInvoke` event.
	`beforeDisplay`	Runs the attached script before an internal request for the enabled/checked status of the `scriptMenuActionscriptMenuAction`.
	`onInvoke`	Runs the attached script when the `scriptMenuAction` is invoked.
`submenu`	`beforeDisplay`	Runs the attached script before the contents of the `submenu` are shown.

For more about `events` and `eventListeners`, see Chapter 8, "Events."

To change the items displayed in a menu, add an `eventListener` for the `beforeDisplay event`. When the menu is selected, the `eventListener` can then run a script that enables or disables menu items, changes

the wording of menu item, or performs other tasks related to the menu. This mechanism is used internally to change the menu listing of available fonts, recent documents, or open windows.

Working with scriptMenuActions

You can use `scriptMenuAction` to create a new `menuAction` whose behavior is implemented through the script registered to run when the `onInvoke event` is triggered.

The following script shows how to create a `scriptMenuAction` and attach it to a menu item (for the complete script, see MakeScriptMenuAction). This script simply displays an alert when the menu item is selected.

```
var mySampleScriptAction = app.scriptMenuActions.add("Display Message");
var myEventListener = mySampleScriptAction.eventListeners.add("onInvoke",
function(){alert("This menu item was added by a script.");});
//If the submenu "Script Menu Action" does not already exist, create it.
try{
    var mySampleScriptMenu = app.menus.item("$ID/Main").submenus.item(
    "Script Menu Action");
    mySampleScriptMenu.title;
}
catch (myError){
    var mySampleScriptMenu = app.menus.item("$ID/Main").submenus.add
    ("Script Menu Action");
}
var mySampleScriptMenuItem = mySampleScriptMenu.menuItems.add(mySampleScriptAction);
```

To remove the `menu`, `submenu`, `menuItem`, and `scriptMenuAction` created by the above script, run the following script fragment (from the RemoveScriptMenuAction tutorial script):

```
#targetengine "session"
var mySampleScriptAction = app.scriptMenuActions.item("Display Message");
mySampleScriptAction.remove();
var mySampleScriptMenu = app.menus.item("$ID/Main").submenus.item
("Script Menu Action");
mySampleScriptMenu.remove();
```

You also can remove all `scriptMenuAction`, as shown in the following script fragment (from the RemoveAllScriptMenuActions tutorial script). This script also removes the menu listings of the `scriptMenuAction`, but it does not delete any menus or submenus you might have created.

```
#targetengine "session"
app.scriptMenuActions.everyItem().remove();
```

You can create a list of all current `scriptMenuActions`, as shown in the following script fragment (from the ListScriptMenuActions tutorial script):

```
var myScriptMenuActionNames = app.scriptMenuActions.everyItem().name;
//Open a new text file.
var myTextFile = File.saveDialog("Save Script Menu Action Names As", undefined);
//If the user clicked the Cancel button, the result is null.
if(myTextFile != null){
    //Open the file with write access.
    myTextFile.open("w");
    for(var myCounter = 0; myCounter < myScriptMenuActionNames.length; myCounter++){
        myTextFile.writeln(myScriptMenuActionNames[myCounter]);
    }
    myTextFile.close();
}
```

`scriptMenuAction` also can run scripts during their `beforeDisplay event`, in which case they are executed before an internal request for the state of the `scriptMenuAction` (e.g., when the menu item is about to be displayed). Among other things, the script can then change the menu names and/or set the enabled/checked status.

In the following sample script, we add an `eventListener` to the `beforeDisplay event` that checks the current selection. If there is no selection, the script in the `eventListener` disables the menu item. If an item is selected, the menu item is enabled, and choosing the menu item displays the type of the first item in the selection. (For the complete script, see BeforeDisplay.)

```
var mySampleScriptAction = app.scriptMenuActions.add("Display Message");
var myEventListener = mySampleScriptAction.eventListeners.add("onInvoke", function()
    {
        //JavaScript function to run when the menu item is selected.
        myString = app.selection[0].constructor.name;
        alert("The first item in the selection is a " + myString + ".");
    });
var mySampleScriptMenu = app.menus.item("$ID/Main").submenus.add("Script Menu
Action");
var mySampleScriptMenuItem = mySampleScriptMenu.menuItems.add(mySampleScriptAction);
mySampleScriptMenu.eventListeners.add("beforeDisplay", function()
    {
        //JavaScript function to run before the menu item is drawns.
        var mySampleScriptAction = app.scriptMenuActions.item("Display Message");
        if(app.selection.length > 0){
            mySampleScriptAction.enabled = true;
        }
        else{
            mySampleScriptAction.enabled = false;
        }
    });
```

A More Complex Menu-scripting Example

You have probably noticed that selecting different items in the InDesign user interface changes the contents of the context menus. The following sample script shows how to modify the context menu based on the properties of the object you select. Fragments of the script are shown below; for the complete script, see LayoutContextMenu.

The following snippet shows how to create a new menu item on the Layout context menu (the context menu that appears when you have a page item selected). The following snippet adds a `beforeDisplay eventListener` which checks for the existence of a `menuItem` and removes it if it already exists. We do this to ensure the `menuItem` does not appear on the context menu when the selection does not contain a

graphic, and to avoid adding multiple menu choices to the context menu. The `eventListener` then checks the selection to see if it contains a graphic; if so, it creates a new `scriptMenuItem`.

```
//The locale-independent name (aka "key string") for the
//Layout context menu is "$ID/RtMouseLayout".
var myLayoutContextMenu = app.menus.item("$ID/RtMouseLayout");
//Create the event handler for the "beforeDisplay"
//event of the Layout context menu.
var myBeforeDisplayListener = myLayoutContextMenu.addEventListener
("beforeDisplay", myBeforeDisplayHandler, false);
//This event handler checks the type of the selection.
//If a graphic is selected, the event handler adds the script menu
//action to the menu.
function myBeforeDisplayHandler(myEvent){
    if(app.documents.length != 0){
        if(app.selection.length > 0){
            var myObjectList = new Array;
            //Does the selection contain any graphics?
            for(var myCounter = 0; myCounter < app.selection.length;
            myCounter ++){
                switch(app.selection[myCounter].constructor.name){
                    case "PDF":
                    case "EPS":
                    case "Image":
                        myObjectList.push(app.selection[myCounter]);
                        break;
                    case "Rectangle":
                    case "Oval":
                    case "Polygon":
                        if(app.selection[myCounter].graphics.length != 0){
                            myObjectList.push(app.selection[myCounter].
                            graphics.item(0));
                        }
                        break;
                    default:
                }
            }
            if(myObjectList.length > 0){
                //Add the menu item if it does not already exist.
                if(myCheckForMenuItem(myLayoutContextMenu,
                "Create Graphic Label") == false){
                    myMakeLabelGraphicMenuItem();
                }
            }
            else{
                //Remove the menu item, if it exists.
                if(myCheckForMenuItem(myLayoutContextMenu,
                "Create Graphic Label") == true){
                    myLayoutContextMenu.menuItems.item("Create Graphic
                    Label").remove();
                }
            }
        }
    }
    function myMakeLabelGraphicMenuItem(){
        //alert("Got to the myMakeLabelGraphicMenuItem function!");
        if(myCheckForScriptMenuItem("Create Graphic Label") == false){
            //alert("Making a new script menu action!");
            var myLabelGraphicMenuAction = app.scriptMenuActions.add("Create
            Graphic Label");
```

```
    var myLabelGraphicEventListener = myLabelGraphicMenuAction.
    eventListeners.add("onInvoke", myLabelGraphicEventHandler, false);
}
var myLabelGraphicMenuItem = app.menus.item("$ID/RtMouseLayout").
menuItems.add(app.scriptMenuActions.item("Create Graphic Label"));
function myLabelGraphicEventHandler(myEvent){
    //alert("Got to myLabelGraphicEventListener!");
    if(app.selection.length > 0){
        var myObjectList = new Array;
        //Does the selection contain any graphics?
        for(var myCounter = 0; myCounter < app.selection.length;
        myCounter ++){
               switch(app.selection[myCounter].constructor.name){
                   case "PDF":
                   case "EPS":
                   case "Image":
                          myObjectList.push(app.selection[myCounter]);
                      break;
                   case "Rectangle":
                   case "Oval":
                   case "Polygon":
                      if(app.selection[myCounter].graphics.length != 0){
                         myObjectList.push(app.selection[myCounter].
                         graphics.item(0));
                      }
                      break;
                   default:
               }
           }
           if(myObjectList.length > 0){
               myDisplayDialog(myObjectList);
           }
        }
        //Function that adds the label.
        function myAddLabel(myGraphic, myLabelType,
        myLabelHeight, myLabelOffset, myLabelStyleName, myLayerName){
           var myLabelLayer;
           var myDocument = app.documents.item(0);
           var myLabel;
           myLabelStyle = myDocument.paragraphStyles.item
           (myLabelStyleName);
           var myLink = myGraphic.itemLink;
           try{
               myLabelLayer = myDocument.layers.item(myLayerName);
               //if the layer does not exist, trying to get
               //the layer name will cause an error.
               myLabelLayer.name;
           }
           catch (myError){
               myLabelLayer = myDocument.layers.add(myLayerName);
           }
           //Label type defines the text that goes in the label.
           switch(myLabelType){
               //File name
               case 0:
                   myLabel = myLink.name;
                   break;
               //File path
               case 1:
                   myLabel = myLink.filePath;
```

```
                break;
            //XMP description
            case 2:
                try{
                    myLabel = myLink.linkXmp.description;
                }
                catch(myError){
                    myLabel = "No description available.";
                }
                break;
            //XMP author
            case 3:
                try{
                    myLabel = myLink.linkXmp.author
                }
                catch(myError){
                    myLabel = "No author available.";
                }
                break;
        }
        var myFrame = myGraphic.parent;
        var myX1 = myFrame.geometricBounds[1];
        var myY1 = myFrame.geometricBounds[2] + myLabelOffset;
        var myX2 = myFrame.geometricBounds[3];
        var myY2 = myY1 + myLabelHeight;
        var myTextFrame = myFrame.parent.textFrames.add(myLabelLayer,
        undefined, undefined,{geometricBounds:[myY1, myX1, myY2,
        myX2],contents:myLabel});
        myTextFrame.textFramePreferences.firstBaselineOffset =
        FirstBaseline.leadingOffset;
        myTextFrame.paragraphs.item(0).appliedParagraphStyle =
        myLabelStyle;
    }
    function myDisplayDialog(myObjectList){
        var myLabelWidth = 100;
        var myStyleNames = myGetParagraphStyleNames
        (app.documents.item(0));
        var myLayerNames = myGetLayerNames(app.documents.item(0));
        var myDialog = app.dialogs.add({name:"LabelGraphics"});
        with(myDialog.dialogColumns.add()){
            //Label type
            with(dialogRows.add()){
                with(dialogColumns.add()){
                    staticTexts.add({staticLabel:"Label Type",
                    minWidth:myLabelWidth});
                }
                with(dialogColumns.add()){
                    var myLabelTypeDropdown = dropdowns.add(
                    {stringList:["File name", "File path", "XMP
                    description", "XMP author"], selectedIndex:0});
                }
            }
            //Text frame height
            with(dialogRows.add()){
                with(dialogColumns.add()){
                    staticTexts.add({staticLabel:"Label Height",
                    minWidth:myLabelWidth});
                }
                with(dialogColumns.add()){
                    var myLabelHeightField = measurementEditboxes.add
```

```
                ({editValue:24, editUnits:MeasurementUnits.points});
            }
        }
        //Text frame offset
        with(dialogRows.add()){
            with(dialogColumns.add()){
                staticTexts.add({staticLabel:"Label Offset",
                minWidth:myLabelWidth});
            }
            with(dialogColumns.add()){
                var myLabelOffsetField = measurementEditboxes.add
                ({editValue:0, editUnits:MeasurementUnits.points});
            }
        }
        //Style to apply
        with(dialogRows.add()){
            with(dialogColumns.add()){
                staticTexts.add({staticLabel:"Label Style",
                minWidth:myLabelWidth});
            }
            with(dialogColumns.add()){
                var myLabelStyleDropdown = dropdowns.add
                ({stringList:myStyleNames, selectedIndex:0});
            }
        }
        //Layer
        with(dialogRows.add()){
            with(dialogColumns.add()){
                staticTexts.add({staticLabel:"Layer:",
                minWidth:myLabelWidth});
            }
            with(dialogColumns.add()){
                var myLayerDropdown = dropdowns.add
                ({stringList:myLayerNames, selectedIndex:0});
            }
        }
    }
    var myResult = myDialog.show();
    if(myResult == true){
        var myLabelType = myLabelTypeDropdown.selectedIndex;
        var myLabelHeight = myLabelHeightField.editValue;
        var myLabelOffset = myLabelOffsetField.editValue;
        var myLabelStyle = myStyleNames[myLabelStyleDropdown.
        selectedIndex];
        var myLayerName = myLayerNames[myLayerDropdown.
        selectedIndex];
        myDialog.destroy();
        var myOldXUnits = app.documents.item(0).viewPreferences.
        horizontalMeasurementUnits;
        var myOldYUnits = app.documents.item(0).viewPreferences.
        verticalMeasurementUnits;
        app.documents.item(0).viewPreferences.
        horizontalMeasurementUnits = MeasurementUnits.points;
        app.documents.item(0).viewPreferences.
        verticalMeasurementUnits = MeasurementUnits.points;
        for(var myCounter = 0; myCounter < myObjectList.length;
        myCounter++){
            var myGraphic = myObjectList[myCounter];
```

```
                        myAddLabel(myGraphic, myLabelType, myLabelHeight,
                        myLabelOffset, myLabelStyle, myLayerName);
                    }
                    app.documents.item(0).viewPreferences.
                    horizontalMeasurementUnits = myOldXUnits;
                    app.documents.item(0).viewPreferences.
                    verticalMeasurementUnits = myOldYUnits;
                }
                else{
                    myDialog.destroy();
                }
            }
        }
    }
}
```

10 Working with Preflight

Chapter Update Status	
CS6	Unchanged

Preflight is a way to verify that you have all required files, fonts, assets (e.g., placed images and PDF files), printer settings, trapping styles, etc., before you send a publication to an output device. For example, if you placed an image as a low-resolution proxy but do not have the high-resolution original image accessible on your hard disk (or workgroup server), that may result in an error during the printing process. Preflight checks for this sort of problem. It can be run in the background as you work.

This chapter demonstrates how to interact with the preflight system using scripting. For illustration purposes, we show how to configure preflight to raise an error if the page size is something other than letter size (8.5" x 11"). We briefly highlight how it is done in the user interface, then show how to achieve the same results through scripting.

We assume that you have already read *Adobe InDesign Scripting Tutorial* and know how to create, install, and run a script.

Exploring Preflight Profiles

InDesign's preflight feature is profile based, rule driven, and parameterized. There might be one or more preflight profiles. Initially, there is one profile, [Basic], which is read-only; you cannot modify or delete it.

A preflight profile contains many preflight rules. Each rule has a name and multiple data objects. Each data object has a name, data type, and data value. The data value can be changed. Each rule can be configured as follows:

- Disabled — The preflight rule is disabled.
- Return as error — The preflight rule returns error-level feedback.
- Return as warning — The preflight rule returns warning-level feedback.
- Return as informational — The preflight rule returns informational-level feedback.

To check the profile in InDesign, choose Preflight Panel > Define Profiles. You also can get profile information by scripting.

Listing preflight profiles

This script fragment shows how to list all preflight profiles. For the complete script, see ListPreflightProfiles.

```
var profiles = app.preflightProfiles;
var profileCount = profiles.length;|
var str = "Preflight profiles: ";
for (var i = 0; i < profileCount; i++)
{
    if (i != 0)
    {
        str += ", ";
    }
    str += profiles.item(i).name;
}
alert(str);
```

Listing preflight rules

This script fragment shows how to list all preflight rules in a profile. For the complete script, see ListPreflightRules.

```
// Assume the [Basic] profile exists
var myProfile = app.preflightProfiles.item(0);
var myRules = myProfile.preflightProfileRules;
var ruleCount = myRules.length;
var str = "Preflight rules of " + myProfile.name + ": ";
for (var i = 0; i < ruleCount; i++)
{
    if (i > 0)
    {
        str += ", ";
    }
    str += myRules.item(i).name;
}
alert(str);
```

Listing preflight data objects

This script fragment shows how to list all preflight data objects in a profile rule. For the complete script, see ListPreflightDataObjects.

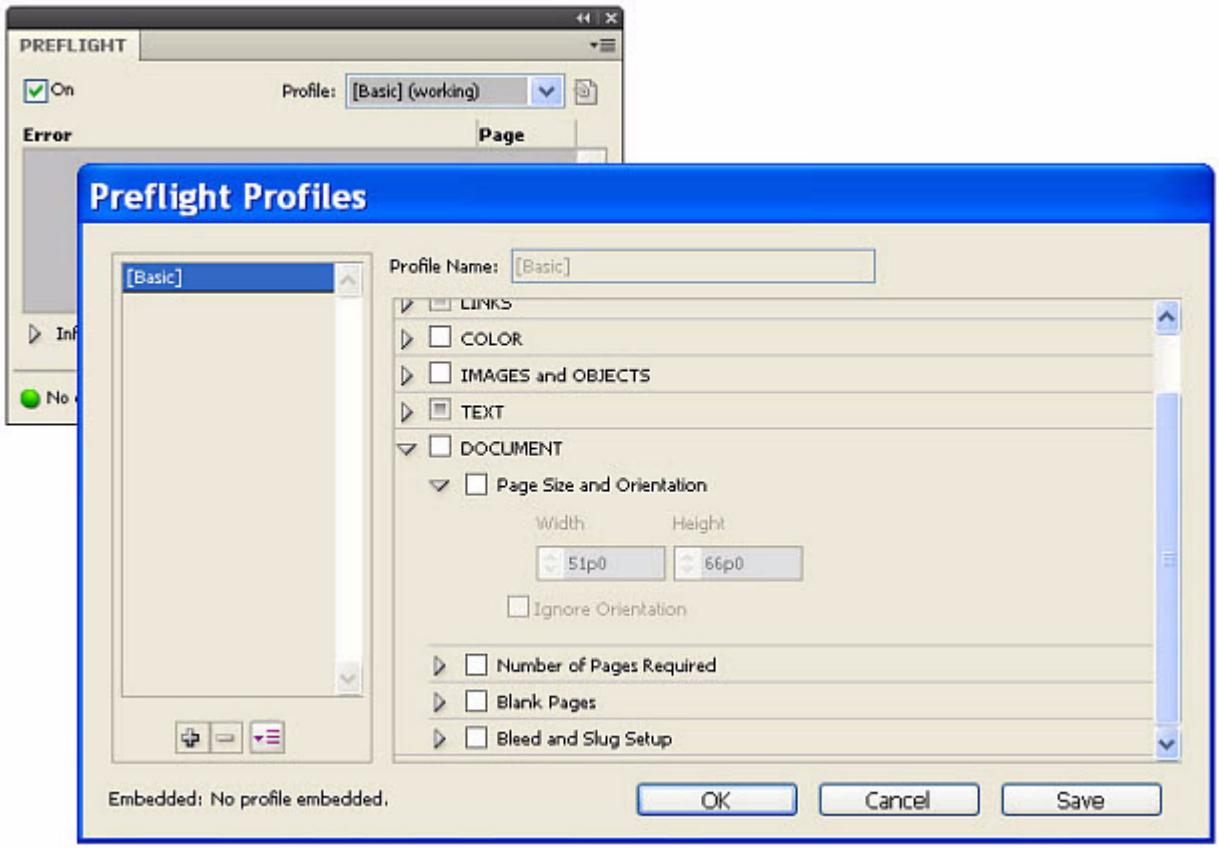
```
// Assume the [Basic] profile exists.
var myProfile = app.preflightProfiles.item(0);
// rule ADBE_BlankPages
var myRule = myProfile.preflightProfileRules.item(0);
var dataObjects = myRule.ruleDataObjects;
var dataObjectCount = dataObjects.length;
var str = "Preflight rule data objects of " + myProfile.name + "." + myRule.name + ": ";
for (var i = 0; i < dataObjectCount; i++)
{
    if (i > 0)
    {
        str += "; ";
    }
    str += dataObjects.item(i).name + ", ";
    str += getDataObjectDataType(dataObjects.item(i).dataType) + ", ";
    str += dataObjects.item(i).dataValue;
}
alert(str);

function getDataObjectDataType(type)
```

```
{
    if (type == RuleDataType.BOOLEAN_DATA_TYPE)
    {
        return "Boolean";
    }
    else if (type == RuleDataType.INTEGER_DATA_TYPE)
    {
        return "Integer";
    }
    else if (type == RuleDataType.LIST_DATA_TYPE)
    {
        return "List";
    }
    else if (type == RuleDataType.OBJECT_DATA_TYPE)
    {
        return "Object";
    }
    else if (type == RuleDataType.REAL_DATA_TYPE)
    {
        return "Real";
    }
    else if (type == RuleDataType.SHORT_INTEGER_DATA_TYPE)
    {
        return "Short Integer";
    }
    else if (type == RuleDataType.STRING_DATA_TYPE)
    {
        return "String";
    }
    else
    {
        return type;
    }
}
```

Importing a Preflight Profile

To import a preflight profile from the Preflight panel, choose Preflight Panel > Define Profiles, then choose Load Profile from the drop-down menu in the Preflight Profiles window.



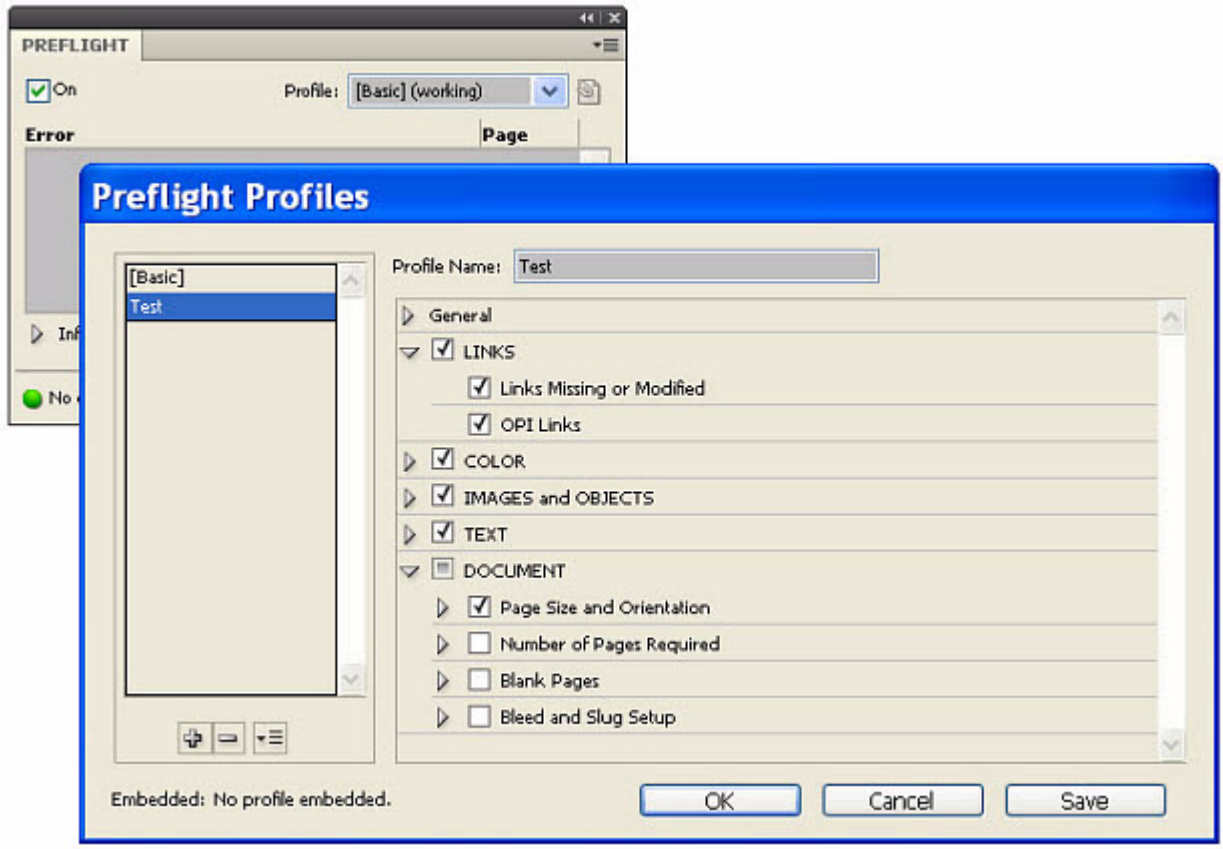
You also can load a profile with scripting. The following script fragment imports a profile called Test. For the complete script, see ImportPreflightProfile.

```
var myProfile = app.loadPreflightProfile(File("/c/Test.idpp"));
if (myProfile == null)
{
    alert("The profile did not load successfully");
}
else
{
    alert("Preflight profile " + myProfile.name + " is loaded.")
}
```

It is easier to create profiles using the Preflight panel than with scripting. One workflow would be to create all profiles in the user interface, export them to files, and import them using scripting. This approach avoids the challenges involved with manually adding rules via scripting.

Creating a Preflight Profile

To create a preflight profile from the Preflight panel, choose Preflight Panel > Define Profiles, then choose the plus sign (+) to add a new preflight profile. Name the profile and fill in all data values for the available rules.



You also can create a profile with scripting. The following script fragment adds a single profile called Test. For the complete script, see CreatePreflightProfile.

```
//Add a new preflight profile
var myProfile = app.preflightProfiles.add();
if (myProfile == null)
{
    alert("The profile did not add successfully");
}
else
{
    alert("Preflight profile " + myProfile.name + " is added.")
}
//Rename the profile.
var oldName = myProfile.name
myProfile.name = "Test";
myProfile.description = "Test description";
alert("Preflight profile " + oldName + " is renamed to " + myProfile.name + ".")
```

Preflight-profile names must be unique. If the script above is executed more than once within the same InDesign instance, an error is raised, indicating that a profile with that name already exists. To avoid this, either access the existing profile using `app.preflightProfiles.itemByName()`, or check to see if a profile exists and remove it; see the following script fragment. For the complete script, see DeletePreflightProfile.

```
function removeProfile(name)
{
    //Lookup the existing preflight profile by name
    var oldProfile = app.preflightProfiles.itemByName(name);
    //If a profile with that name was found
    if (oldProfile != null)
    {
        oldProfile.remove();
    }
}
```

Adding Rules

A preflight profile contains a mutually exclusive set of rules. To add a rule to a profile, follow these steps:

1. Add a rule to a profile by name.

   Rules are added by name. For information on rule names, see "Available Rules" on page 163. The following adds the ADBE_PageSizeOrientation rule to the profile.

   ```
   //Add a rule that requires a specific page size and orientation
   //(portrait or landscape).
   const RULE_NAME = "ADBE_PageSizeOrientation";
   var myRule = myProfile.preflightProfileRules.add(RULE_NAME);
   ```

2. Set the rule's data values.

   Many, but not all, rules have data properties. For a complete specification of the rules available with InDesign, see "Available Rules" on page 163. The ADBE_PageSizeOrientation rule contains particular data properties that allow you to specify a page size. The following sets the acceptable page height and width, a tolerance (fudge factor), and an option for handling page orientation.

   ```
   //Requires the page size to be 8.5in x 11in (Letter Size)
   //enters a value for tolerance
   myRule.ruleDataObjects.add("tolerance", RuleDataType.realDataType, 0.01);
   //Sets the width  to the point equivalent of 8.5 inches
   myRule.ruleDataObjects.add("width", RuleDataType.realDataType, 612);
   //Sets the width  to the point equivalent of 11 inches
   myRule.ruleDataObjects.add("height", RuleDataType.realDataType, 792);
   //true = ignore orientation is checked
   myRule.ruleDataObjects.add("ignore_orientation", RuleDataType.booleanDataType,
   true);
   ```

3. Set the rule's reporting state.

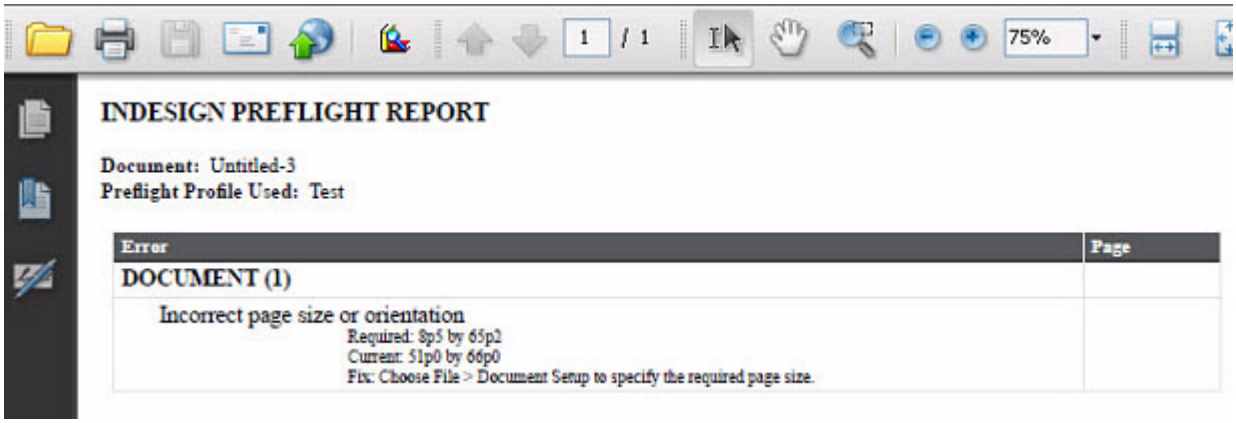
   This is done using the rule's `flag` property. There are several choices (disabled, information, warning, and error), controlled by the `PreflightRuleFlag` enumeration.

   ```
   //set the rule to return an error
   myRule.flag = PreflightRuleFlag.returnAsError;
   ```

Processing a Profile

In the desktop version of InDesign, preflight errors are reported in the user interface. In scripting (especially for InDesign Server), the errors are generated on demand. The following script processes an InDesign document. (For the complete script, see ProcessPreflightProfile.) If there are errors, it writes the results to a new PDF file. For an example of the output, see the figure below the script.

```
// Assume there is a document.
var myDoc = app.documents.item(0);
// Assume the Test preflight profile exists.
var myProfile = app.preflightProfiles.itemByName("Test");
//Process the myDoc with the rule
var myProcess = app.preflightProcesses.add(myDoc, myProfile);
myProcess.waitForProcess();
var myResults = myProcess.processResults;
//If errors were found
if (myResults != "None")
{
    //Export the file to PDF. The "true" value selects to open the file after export.
    myProcess.saveReport(File("C:\PreflightResults.pdf"), true);
}
//Cleanup
myProcess.remove();
```



If you would rather produce a text file, simply name your output file with a `.txt` extension.

Alternately, you may prefer to iterate the errors yourself. The following demonstrates how to access the errors array. For the complete script, see ProcessPreflightProfileShowErrors.

```
//If errors were found
if (myResults != "None")
{
    // array containing detailed results
    var errors = myProcess.aggregatedResults
    // Show the errors in a message box.
    var str = "Document: " + errors[0] + ", Profile: " + errors[1] + ", Results: ["
    var errorResults = errors[2]
    for (var i = 0; i < errorResults.length; i++)
    {
        if (i > 1)
        {
            str += ", "
        }
        str += errorResults[i][1]
    }
    str = str + "]"
    alert(str)
}
```

Custom Rules

It is not possible to create custom rules through the Preflight panel or scripting; however, this can be done through a C++ plug-in. The InDesign Products SDK contains a sample, PreflightRule, that demonstrates how to add custom rules with a plug-in.

Available Rules

One of the hardest aspects of scripting rules is discovering rule names and properties. Due to the dynamic nature of rules (they really are just strings), specific rule names and properties do not appear in the Extend Script Tool Kit's Object Model Viewer. To discover this information, see "Exploring Preflight Profiles" on page 156. For your convenience, the `DumpPreflightRules.jsx` script is provided in the SDK to produce the following output as an HTML file (*SDK*\docs\references\PreflightRules.html). If you use a plug-in that adds custom rules, you can run the script to extract the new names and properties.

Rule name	Rule properties
ADBE_BlankPages	"ADBE_BlankPages" on page 164
ADBE_BleedSlug	"ADBE_BleedSlug" on page 165
ADBE_BleedTrimHazard	"ADBE_BleedTrimHazard" on page 166
ADBE_CMYPlates	no
ADBE_Colorspace	"ADBE_Colorspace" on page 166
ADBE_ConditionIndicators	no
ADBE_CrossReferences	"ADBE_CrossReferences" on page 166yes
ADBE_FontUsage	"ADBE_FontUsage" on page 166
ADBE_ImageColorManagement	"ADBE_ImageColorManagement" on page 167
ADBE_ImageResolution	"ADBE_ImageResolution" on page 167

Rule name	Rule properties
ADBE_InteractiveContent	no
ADBE_LayerVisibility	no
ADBE_MissingFonts	no
ADBE_MissingGlyph	no
ADBE_MissingModifiedGraphics	no
ADBE_OPI	no
ADBE_Overprint	no
ADBE_OversetText	no
ADBE_PageCount	"ADBE_PageCount" on page 168
ADBE_PageSizeOrientation	"ADBE_PageSizeOrientation" on page 168
ADBE_Registration	no
ADBE_ScaledGraphics	"ADBE_ScaledGraphics" on page 168
ADBE_ScaledType	"ADBE_ScaledType" on page 168
ADBE_SmallText	"ADBE_SmallText" on page 168
ADBE_SpellCheck	no
ADBE_SpotColorSetup	"ADBE_SpotColorSetup" on page 169
ADBE_StrokeRequirements	"ADBE_StrokeRequirements" on page 169
ADBE_TextOverrides	"ADBE_TextOverrides" on page 169
ADBE_TransparencyBlending	"ADBE_TransparencyBlending" on page 169
ADBE_TransparencyUsage	no
ADBE_WhiteOverprint	no

ADBE_BlankPages

Data Type	Name	Default value
Boolean	ignore_master	true
Boolean	ignore_nonprinting	true

ADBE_BleedSlug

Data Type	Name	Default value
Real	bleed_b	9
Real	bleed_b_aux	9
Integer	bleed_comparison_type	3
Boolean	bleed_enabled	true
Real	bleed_l	9
Real	bleed_l_aux	9
Real	bleed_r	9
Real	bleed_r_aux	9
Real	bleed_t	9
Real	bleed_t_aux	9
Real	slug_b	18
Real	slug_b_aux	18
Integer	slug_comparison_type	3
Boolean	slug_enabled	false
Real	slug_l	18
Real	slug_l_aux	18
Real	slug_r	18
Real	slug_r_aux	18
Real	slug_t	18
Real	slug_t_aux	18
Real	tolerance	0.01

ADBE_BleedTrimHazard

Data Type	Name	Default value
Boolean	binding_enabled	false
Real	binding_width	1
Real	live_b	18
Real	live_l	18
Real	live_r	18
Real	live_t	18
Real	tolerance	0.01

ADBE_Colorspace

Data Type	Name	Default value
Boolean	no_cmyk	false
Boolean	no_gray	false
Boolean	no_lab	false
Boolean	no_rgb	false
Boolean	no_spot	false

ADBE_CrossReferences

Data Type	Name	Default value
Boolean	xrefs_out_of_date	true
Boolean	xrefs_unresolved	true

ADBE_FontUsage

Data Type	Name	Default value
Boolean	no_ATC	false
Boolean	no_Bitmap	false
Boolean	no_CID	false
Boolean	no_MultipleMaster	false

Data Type	Name	Default value
Boolean	no_OpenTypeCFF	false
Boolean	no_OpenTypeCID	false
Boolean	no_OpenTypeTT	false
Boolean	no_TrueType	false
Boolean	no_Type1	false
Boolean	no_protected	true

ADBE_ImageColorManagement

Data Type	Name	Default value
Boolean	no_cmyk_profiles	true
Boolean	no_image_overrides	true
Boolean	overrides_exclude_uncal	true

ADBE_ImageResolution

Data Type	Name	Default value
Boolean	bw_max_enabled	false
Real	bw_max_res	2400
Boolean	bw_min_enabled	true
Real	bw_min_res	800
Boolean	color_max_enabled	false
Real	color_max_res	1200
Boolean	color_min_enabled	true
Real	color_min_res	250
Boolean	gray_max_enabled	false
Real	gray_max_res	1200
Boolean	gray_min_enabled	true
Real	gray_min_res	250
Real	tolerance	0.5

ADBE_PageCount

Data Type	Name	Default value
Integer	comparison_type	2
Integer	comparison_value	1
Integer	comparison_value_aux	1

ADBE_PageSizeOrientation

Data Type	Name	Default value
Real	height	792
Boolean	ignore_orientation	false
Real	tolerance	0.01
Real	width	612

ADBE_ScaledGraphics

Data Type	Name	Default value
Real	max_scale	100.5

ADBE_ScaledType

Data Type	Name	Default value
Boolean	ignore_justification	true
Real	max_scale	100.5

ADBE_SmallText

Data Type	Name	Default value
Real	minSize	4
Boolean	minSize_trap_safe_only	false

ADBE_SpotColorSetup

Data Type	Name	Default value
Boolean	lab_spots	true
Boolean	lab_spots_enabled	false
Integer	max_spots	1
Boolean	max_spots_enabled	true

ADBE_StrokeRequirements

Data Type	Name	Default value
Real	min_width	0.125
Boolean	min_width_trap_safe_only	false

ADBE_TextOverrides

Data Type	Name	Default value
Boolean	ignore_color_overrides	false
Boolean	ignore_font_overrides	false
Boolean	ignore_kerning_tracking_overrides	false
Boolean	ignore_language_overrides	false

ADBE_TransparencyBlending

Data Type	Name	Default value
Integer	space	3

11 Creating Dynamic Documents

Chapter Update Status	
CS6	Unchanged

InDesign can create documents for web and online use, also known as Rich Interactive Documents (RID). Dynamic documents contain sounds, animations, hyperlinks, and other interactive content. InDesign documents can be exported to SWF, XFL, or PDF. For SWF and XFL files, documents can include animations, buttons, multistate objects, movies, and sound clips. You can use the Preview panel in InDesign to test some types of dynamic content before exporting.

This chapter shows how to create dynamic documents using scripting. For more on exporting as PDF, SWF, and XFL, refer to the "Working with Documents" chapter.

Importing Movies and Sounds

InDesign can import movie and sound files that can then be viewed or listened to in exported PDF, SWF, or XFL documents. Movies and sounds in an InDesign document are very similar to graphics in that they exist inside container objects on an InDesign page. Unlike graphics, however, you cannot see (or hear) the content of the imported multimedia files on an InDesign page. For that, you'll need to either view the page in the Preview panel, or export the file, then open the file in a viewer capable of displaying the content (such as Acrobat Reader or a web browser).

Scripts can control the playback properties of a sound or movie in an exported dynamic document. You can also add a preview, or "poster," image to the page item containing the sound or movie.

The following script fragment shows how to import a movie and control the way that the movie is shown and played in an exported document (for the complete script, refer to PlaceMovie).

```
//Given a page "myPage"...
var myFrame = myPage.rectangles.add({geometricBounds:[72, 72, 288, 288]});
//Import a movie file (you'll have to provide a valid file path on your system);
var myMovie = myFrame.place(File("/c/movie.flv"))[0];
//Set movie properties.
myMovie.embedInPDF = true;
myMovie.showControls = true;
//Add a preview image. You'll have to provide a valid path on your system.
myMovie.posterFile = File("/c/movie poster.jpg");
```

The following script fragment shows how to import a sound file and control the playback and display of the sound in an exported document (for the complete script, refer to PlaceSound).

```
//Given a page "myPage"...
//Import a sound file (you'll have to provide a valid file path on your system);
var mySound = myPage.place(File("/c/sound.mp3"), [72, 72])[0];
//Set sound properties.
mySound.embedInPDF = true;
mySound.doNotPrintPoster = true;
mySound.soundLoop = true;
mySound.stopOnPageTurn = true;
//Add a preview image. You'll have to provide a valid path on your system.
mySound.posterFile = File("/c/sound poster.jpg");
```

Buttons can be used to control the playback of sounds and movies. For information on how to script buttons, see the next section.

Creating Buttons

Buttons are often used for navigation in dynamic documents. Buttons contain three states, known as "Normal," "Rollover," and "Click," which, in turn, can contain page items such as rectangles, ovals, text frames, or images. The button can display only one state at a time; the other states are displayed when triggered by mouse actions.

Behaviors control what the button does when you perform a specific mouse action. Behaviors correspond to the Actions shown in the Buttons panel in InDesign's user interface. Buttons can contain multiple behaviors.

The following script fragment shows how to create a simple button that displays the next page in an exported PDF or SWF (for the complete script, refer to SimpleButton). This button makes use of only the Normal state.

```
//Given a page "myPage" and a document containing the color "Red"...
//Make a button by converting a page item.
var myRightArrow = myPage.polygons.add({fillColor:myDocument.colors.item("Red"),
name:"GoToNextPageButton"});
myRightArrow.paths.item(0).entirePath = [[72, 72],[144,108],[72, 144]];
var myButton = myPage.buttons.add({geometricBounds:[72, 72, 144, 144]});
myButton.states.item(0).addItemsToState(myRightArrow);
var myGoToNextPageBehavior =
myButton.gotoNextPageBehaviors.add({behaviorEvent:BehaviorEvents.mouseUp});
```

The following script fragment shows how to create a somewhat more complicated button, containing page items that change the appearance of each of the three button states. For the complete script, refer to ButtonStates.

```
//Given a page "myPage" in a document "myDocument," containing the colors
//"Blue" and "Red"...
//Make a button "from scratch."
var myButton = myPage.buttons.add({geometricBounds:[72, 72, 144, 144],
name:"GoToNextPageButton"});
var myRightArrow =
myButton.states.item(0).polygons.add({fillColor:myDocument.colors.item("Red")});
myRightArrow.paths.item(0).entirePath = [[72, 72],[144,108],[72, 144]];
//Add the Rollover state.
var myRolloverState = myButton.states.add();
//Add a shadow to the polygon in the Rollover state.
var myRolloverArrow =
myRolloverState.polygons.add({fillColor:myDocument.colors.item("Red")});
myRolloverArrow.paths.item(0).entirePath = [[72, 72],[144,108],[72, 144]];
var myFillTransparencySettings = myRolloverArrow.fillTransparencySettings;
myFillTransparencySettings.dropShadowSettings.mode = ShadowMode.drop;
myFillTransparencySettings.dropShadowSettings.angle = 90;
myFillTransparencySettings.dropShadowSettings.xOffset = 0;
myFillTransparencySettings.dropShadowSettings.yOffset = 0;
myFillTransparencySettings.dropShadowSettings.size = 6;
//Add a shadow to the polygon in the Click state.
var myClickState = myButton.states.add();
var myClickArrow =
myClickState.polygons.add({fillColor:myDocument.colors.item("Blue")});
myClickArrow.paths.item(0).entirePath = [[72, 72],[144,108],[72, 144]];
//Set the behavior for the button.
var myGoToNextPageBehavior =
myButton.gotoNextPageBehaviors.add({behaviorEvent:BehaviorEvents.mouseUp});
```

Buttons can be used to control the playback of movie and sound files. The following script fragment shows an example of using a set of buttons to control the playback of a moving file (for the complete script, refer to MovieControl).

```
//Given a page "myPage" in a document "myDocument,"
//containing the colors "Gray" and "Red"...
var myFrame = myPage.rectangles.add({geometricBounds:[72, 72, 288, 288]});
//Import a movie file (you'll have to provide a valid file path on your system);
myFrame.place(File("/c/movie.flv"));
//Create the movie "Start" button.
var myPlayButton = myPage.buttons.add({geometricBounds:[294,186,354,282],
name:"PlayMovieButton"});
var myRightArrow =
myPlayButton.states.item(0).polygons.add({fillColor:myDocument.colors.item("Gray")});
myRightArrow.paths.item(0).entirePath = [[186, 294],[186,354],[282, 324]];
//Add the Rollover state.
var myRolloverState = myPlayButton.states.add();
//Add a shadow to the polygon in the Rollover state.
var myRolloverArrow =
myRolloverState.polygons.add({fillColor:myDocument.colors.item("Gray")});
myRolloverArrow.paths.item(0).entirePath = [[186, 294],[186,354],[282, 324]];
var myFillTransparencySettings = myRolloverArrow.fillTransparencySettings;
myFillTransparencySettings.dropShadowSettings.mode = ShadowMode.drop;
myFillTransparencySettings.dropShadowSettings.angle = 90;
myFillTransparencySettings.dropShadowSettings.xOffset = 0;
myFillTransparencySettings.dropShadowSettings.yOffset = 0;
myFillTransparencySettings.dropShadowSettings.size = 6;
//Add a shadow to the polygon in the Click state.
var myClickState = myPlayButton.states.add();
var myClickArrow =
myClickState.polygons.add({fillColor:myDocument.colors.item("Red")});
```

```
myClickArrow.paths.item(0).entirePath = [[186, 294],[186,354],[282, 324]];
//Set the behavior for the button.
var myMovieStartBehavior =
myPlayButton.movieBehaviors.add({movieItem:myFrame.movies.item(0),
behaviorEvent:BehaviorEvents.mouseUp, operation:MoviePlayOperations.play});
//Create the movie "Stop" button.
var myStopButton = myPage.buttons.add({geometricBounds:[294,78,354,174],
name:"StopMovieButton"});
var myNormalRectangle =
myStopButton.states.item(0).rectangles.add({geometricBounds:[294,78,354,174],
fillColor:myDocument.colors.item("Gray")});
myRolloverState = myStopButton.states.add();
var myRolloverRectangle =
myRolloverState.rectangles.add({geometricBounds:[294,78,354,174],
fillColor:myDocument.colors.item("Gray")});
var myFillTransparencySettings = myRolloverRectangle.fillTransparencySettings;
myFillTransparencySettings.dropShadowSettings.mode = ShadowMode.drop;
myFillTransparencySettings.dropShadowSettings.angle = 90;
myFillTransparencySettings.dropShadowSettings.xOffset = 0;
myFillTransparencySettings.dropShadowSettings.yOffset = 0;
myFillTransparencySettings.dropShadowSettings.size = 6;
myClickState = myStopButton.states.add();
var myClickRectangle = myClickState.rectangles.add({geometricBounds:[294,78,354,174],
fillColor:myDocument.colors.item("Red")});
var myMovieStopBehavior =
myStopButton.movieBehaviors.add({movieItem:myFrame.movies.item(0),
behaviorEvent:BehaviorEvents.mouseUp, operation:MoviePlayOperations.stop});
```

Buttons are also important in controlling the appearance of multistate objects, as we'll demonstrate in the next section.

Creating Multistate Objects

Multistate objects (or MSOs) are similar to buttons in that they contains states, and that only one state can be visible at a time. They are unlike buttons in that they can contain any number of states; buttons can contain three states, at most. Multistate objects rely on buttons to change the way they display their states.

The following script fragment shows how to create a simple multistate object and add a button to control the display of the states in the object (for the complete script, refer to MakeMultiStateObject).

```
//Given a document "myDocument" and a page "myPage" and
//four colors "myColorA," "myColorB," "myColorC," and "myColorD"...
var myMSO = myPage.multiStateObjects.add({name:"Spinner", geometricBounds:[72, 72,
144, 144]});
//New multistate objects contain two states when they're created. Add two more.
myMSO.states.item(0).name = "Up";
myMSO.states.item(1).name = "Right";
//Add two more states.
myMSO.states.add({name:"Down"});
myMSO.states.add({name:"Left"});
//Add page items to the states.
var myPolygon = myMSO.states.item(0).polygons.add({fillColor:myColorA,
strokeColor:myDocument.swatches.item("None")});
myPolygon.paths.item(0).entirePath = [[72, 144], [144, 144], [108, 72]];
myPolygon = myMSO.states.item(1).polygons.add({fillColor:myColorB,
strokeColor:myDocument.swatches.item("None")});
myPolygon.paths.item(0).entirePath = [[72, 72], [72, 144], [144, 108]];
myPolygon = myMSO.states.item(2).polygons.add({fillColor:myColorC,
strokeColor:myDocument.swatches.item("None")});
myPolygon.paths.item(0).entirePath = [[72, 72], [108, 144], [144, 72]];
myPolygon = myMSO.states.item(3).polygons.add({fillColor:myColorD,
strokeColor:myDocument.swatches.item("None")});
myPolygon.paths.item(0).entirePath = [[144, 72], [72, 108], [144, 144]];
```

Typically, you'll control the display of the states in a multistate object using a button. The following script fragment shows how to do this (for the complete script, refer to MultiStateObjectControl).

```
//Given a document "myDocument" and a page "myPage" and
//four colors "myColorA," "myColorB," "myColorC," and "myColorD"...
var myMSO = myPage.multiStateObjects.add({name:"Spinner", geometricBounds:[72, 72,
144, 144]});
//New multistate objects contain two states when they're created. Add two more.
myMSO.states.item(0).name = "Up";
myMSO.states.item(1).name = "Right";
//Add two more states.
myMSO.states.add({name:"Down"});
myMSO.states.add({name:"Left"});
//Add page items to the states.
var myPolygon = myMSO.states.item(0).polygons.add({fillColor:myColorA,
strokeColor:myDocument.swatches.item("None")});
myPolygon.paths.item(0).entirePath = [[72, 144], [144, 144], [108, 72]];
myPolygon = myMSO.states.item(1).polygons.add({fillColor:myColorB,
strokeColor:myDocument.swatches.item("None")});
myPolygon.paths.item(0).entirePath = [[72, 72], [72, 144], [144, 108]];
myPolygon = myMSO.states.item(2).polygons.add({fillColor:myColorC,
strokeColor:myDocument.swatches.item("None")});
myPolygon.paths.item(0).entirePath = [[72, 72], [108, 144], [144, 72]];
```

```
myPolygon = myMSO.states.item(3).polygons.add({fillColor:myColorD,
strokeColor:myDocument.swatches.item("None")});
myPolygon.paths.item(0).entirePath = [[144, 72], [72, 108], [144, 144]];
var myButton = myPage.buttons.add({geometricBounds:[72, 72, 144, 144]});
myRolloverState = myButton.states.add();
var myRolloverRectangle = myRolloverState.rectangles.add({geometricBounds:[72, 72,
144, 144]});
var myFillTransparencySettings = myRolloverRectangle.strokeTransparencySettings;
myFillTransparencySettings.dropShadowSettings.mode = ShadowMode.drop;
myFillTransparencySettings.dropShadowSettings.angle = 90;
myFillTransparencySettings.dropShadowSettings.xOffset = 0;
myFillTransparencySettings.dropShadowSettings.yOffset = 0;
myFillTransparencySettings.dropShadowSettings.size = 6;
var myClickState = myButton.states.add();
var myNextStateBehavior =
myButton.gotoNextStateBehaviors.add({associatedMultiStateObject:myMSO,
behaviorEvent:BehaviorEvents.mouseDown, enableBehavior:true,
loopsToNextOrPrevious:true});
```

Working with Animation

Page items can be animated, adding motion to the dynamic documents you create using InDesign. You apply animation to objects using motion presets, define the movement of animated objects using motion paths, and control the duration of the animation using timing settings, timing lists, and timing groups.

The `animationSettings` of an object control the animation that will be applied to the object. When animation settings have been applied to an object, InDesign sets the `hasCustomSettings` property of the object to true; if the object is not to be animated, this property is false.

The point at which an animation begins to play, relative to the event that triggers the animation, is controlled by the objects and properties of the `timingSettings` object attached to the page item or to one of its parent containers (usually the spread).

Basic animation

The following script fragment shows how to create a simple animation (for the complete script, refer to SimpleAnimation). The most basic forms of animation can be applied without using timing settings.

```
//Given a document "myDocument" and a page "myPage" and a color "myColorA"...
//Add a page items to animate.
var myPolygon = myPage.polygons.add({fillColor:myColorA,
strokeColor:myDocument.swatches.item("None")});
myPolygon.paths.item(0).entirePath = [[72, 72], [72, 144], [144, 108]];
//Create a motion path.
var myMotionPathPoints = [[[[108,108],[108,108],[108,108]],[[516, 108],[516,
108],[516, 108]]],true];
//Set animation preferences for the polygon. We havent' set a dynamic trigger
//for the animation, so the polygon's animation will be triggered by
//DynamicTriggerEvents.onPageLoad (the default).
myPolygon.animationSettings.duration = 2;
myPolygon.animationSettings.motionPathPoints = myMotionPathPoints;
```

TimingSettings

The `timingSettings` objects of spreads, pages, and page items control the timing of the animation(s) applied to the object and to any objects contained by the object. `timingSettings` contain:

- `timingLists`, which define the trigger event (page load, page click, and so on) that start the animation.
- `timingGroups`, which associate a page item or series of page items with a specific timing and define the sequence in which animations are shown.

`timingGroups` contain `timingTargets`, which define the objects associated with a given `timingGroup`. `timingTargets` also specify a delay value for the animation applied to the page item, relative to the start of the animation of the `timingGroup` (for the first item in the `timingGroup`), or from the start of the previous item in the `timingGroup` (for other items in the `timingGroup`).

The following script fragment shows how to control the timing of the animation of an object using the various timing objects (for the complete script, refer to TimingSettings). Note that the parameters used to create a `timingGroup` specify the properties of the first `timingTarget` in the `timingGroup`; subsequent `timingTargets`, if any, can be added separately.

```
//Given a document "myDocument" and a page "myPage" and the color "myColorA",
//"myColorB", and "myColorC"...
//Add a page items to animate.
var myPolygonA = myPage.polygons.add({fillColor:myColorA,
strokeColor:myDocument.swatches.item("None")});
myPolygonA.paths.item(0).entirePath = [[72, 72], [72, 144], [144, 108]];
var myPolygonB = myPage.polygons.add({fillColor:myColorB,
strokeColor:myDocument.swatches.item("None")});
myPolygonB.paths.item(0).entirePath = [[72, 72], [72, 144], [144, 108]];
var myPolygonC = myPage.polygons.add({fillColor:myColorC,
strokeColor:myDocument.swatches.item("None")});
myPolygonC.paths.item(0).entirePath = [[72, 72], [72, 144], [144, 108]];
//Create a motion path.
var myMotionPathPoints = [[[[108,108],[108,108],[108,108]],[[516, 108],[516,
108],[516, 108]]],true];
//Set animation preferences for the polygons.
myPolygonA.animationSettings.duration = 2;
myPolygonA.animationSettings.motionPathPoints = myMotionPathPoints;
myPolygonB.animationSettings.duration = 2;
myPolygonB.animationSettings.motionPathPoints = myMotionPathPoints;
myPolygonC.animationSettings.duration = 2;
myPolygonC.animationSettings.motionPathPoints = myMotionPathPoints;
var myTimingSettings = myPage.parent.timingSettings;
//Remove the default timing list.
myTimingSettings.timingLists.item(0).remove();
//Add a new timing list that triggers when the page is clicked.
var myTimingList = myTimingSettings.timingLists.add(DynamicTriggerEvents.onPageClick);
//Add the polygons to a single timing group.
var myTimingGroup = myTimingList.timingGroups.add(myPolygonA, 0);
myTimingGroup.timingTargets.add(myPolygonB, 2);
myTimingGroup.timingTargets.add(myPolygonC, 2);
```

Note that attempting to add a page item whose `hasCustomSettings` property (in the `animationSettings` object of the page item) is false to a `timingTarget` generates an error.

The following script fragment shows how to control the sequence of animations applied to objects on a page (for the complete script, refer to MultipleTimingGroups). Note that the order in which `timingGroups`

are added to a `timingList` determines the order in which the animations play when the trigger event specified in the `timingList` occurs. Some trigger events, such as `DynamicTriggerEvents.onPageLoad`, trigger the animations in the `timingList` (in sequence); others, such as `DynamicTriggerEvents.onPageClick`, trigger the animations one by one, in sequence, with each instance of the event. For example, a `timingList` containing five `timingGroups`, each containing a single `timingTarget`, and having the trigger event `DynamicTriggerEvents.onPageClick` requires five mouse clicks to process all the animations.

```
//Given a document "myDocument" and a page "myPage" and the color "myColorA",
//"myColorB", and "myColorC"...
//Add a page items to animate.
var myPolygonA = myPage.polygons.add({fillColor:myColorA,
strokeColor:myDocument.swatches.item("None")});
myPolygonA.paths.item(0).entirePath = [[72, 72], [72, 144], [144, 108]];
var myPolygonB = myPage.polygons.add({fillColor:myColorB,
strokeColor:myDocument.swatches.item("None")});
myPolygonB.paths.item(0).entirePath = [[72, 72], [72, 144], [144, 108]];
var myPolygonC = myPage.polygons.add({fillColor:myColorC,
strokeColor:myDocument.swatches.item("None")});
myPolygonC.paths.item(0).entirePath = [[72, 72], [72, 144], [144, 108]];
var myPolygonD = myPage.polygons.add({fillColor:myColorA,
strokeColor:myDocument.swatches.item("None")});
myPolygonD.paths.item(0).entirePath = [[72, 144], [72, 216], [144, 180]];
var myPolygonE = myPage.polygons.add({fillColor:myColorB,
strokeColor:myDocument.swatches.item("None")});
myPolygonE.paths.item(0).entirePath = [[72, 144], [72, 216], [144, 180]]
var myPolygonF = myPage.polygons.add({fillColor:myColorC,
strokeColor:myDocument.swatches.item("None")});
myPolygonF.paths.item(0).entirePath = [[72, 144], [72, 216], [144, 180]];
//Create a motion path.
var myMotionPathPointsA = [[[[108,108],[108,108],[108,108]],[[516, 108],[516,
108],[516, 108]]],true];
var myMotionPathPointsB = [[[[108,180],[108,180],[108,180]],[[516, 180],[516,
180],[516, 180]]],true];
//Set animation preferences for the polygons.
//DynamicTriggerEvents.onPageLoad (the default).
myPolygonA.animationSettings.duration = 2;
myPolygonA.animationSettings.motionPathPoints = myMotionPathPointsA;
myPolygonB.animationSettings.duration = 2;
myPolygonB.animationSettings.motionPathPoints = myMotionPathPointsA;
myPolygonC.animationSettings.duration = 2;
myPolygonC.animationSettings.motionPathPoints = myMotionPathPointsA;
myPolygonD.animationSettings.duration = 2;
myPolygonD.animationSettings.motionPathPoints = myMotionPathPointsB;
myPolygonE.animationSettings.duration = 2;
myPolygonE.animationSettings.motionPathPoints = myMotionPathPointsB;
```

```
myPolygonF.animationSettings.duration = 2;
myPolygonF.animationSettings.motionPathPoints = myMotionPathPointsB;
var myTimingSettings = myPage.parent.timingSettings;
//Remove the default timing list.
myTimingSettings.timingLists.item(0).remove();
//Add a new timing list that triggers when the page is clicked.
var myTimingList = myTimingSettings.timingLists.add(DynamicTriggerEvents.onPageClick);
//Add the polygons to a single timing group.
var myTimingGroupA = myTimingList.timingGroups.add(myPolygonA, 0);
myTimingGroupA.timingTargets.add(myPolygonB, 2);
myTimingGroupA.timingTargets.add(myPolygonC, 2);
//myTimingGroupB will play on the second page click.
var myTimingGroupB = myTimingList.timingGroups.add(myPolygonD, 0);
myTimingGroupB.timingTargets.add(myPolygonE, 2);
myTimingGroupB.timingTargets.add(myPolygonF, 2);
```

A given `timingSettings` object can contain multiple `timingList` objects, each of which responds to a different trigger event. The following script fragment shows a series of animations triggered by `DynamicTriggerEvents.onPageLoad`, by `DynamicTriggerEvents.onPageClick` (for the complete script, refer to MultipleTimingLists).

```
//Given a document "myDocument" and a page "myPage" containg 6 polygons:
//"myPolygonA", "myPolygonB", "myPolygonC", "myPolygonD", "myPolygonE", "myPolygonF".
//Add a page items to animate.
var myTimingSettings = myPage.parent.timingSettings;
//At this point, all of the polygons have already been added as timing targets
//of the default timing list. Change the delay of myPolygonB and myPolygonC.
var myTimingListA = myTimingSettings.timingLists.item(0);
var myTimingGroup = myTimingListA.timingGroups.item(1);
myTimingGroup.timingTargets.item(0).delaySeconds = 2;
myTimingGroup = myTimingListA.timingGroups.item(2);
myTimingGroup.timingTargets.item(0).delaySeconds = 2;
//Remove the last three timing groups in the timing list.
myTimingListA.timingGroups.item(-1).remove();
myTimingListA.timingGroups.item(-1).remove();
myTimingListA.timingGroups.item(-1).remove();
//Add a new timing list that triggers when the page is clicked.
var myTimingListB =
myTimingSettings.timingLists.add(DynamicTriggerEvents.onPageClick);
var myTimingGroupB = myTimingListB.timingGroups.add(myPolygonD, 0);
myTimingGroupB.timingTargets.add(myPolygonE, 2);
myTimingGroupB.timingTargets.add(myPolygonF, 2);
```

In the previous examples, we've worked with the `timingSettings` of the spread containing the page items we want to animate. When you want to animate a page item when a user clicks the item, you'll need to use the `timingSettings` of the page item itself, as shown in the following script fragment (for the complete script, refer to PageItemTimingSettings).

```
//Given a document "myDocument" and a page "myPage" containg a polygon "myPolygonA"...
//Remove the default timing list in the timing settings for the spread.
//Set animation preferences for the polygon.
myPolygonA.animationSettings.duration = 2;
myPolygonA.animationSettings.motionPathPoints = myMotionPathPointsA;
myPage.parent.timingSettings.timingLists.item(0).remove();
var myTimingSettings = myPolygonA.timingSettings;
var myTimingList = myTimingSettings.timingLists.add(DynamicTriggerEvents.onClick);
var myTimingGroup = myTimingList.timingGroups.add(myPolygonA, 0);
```

Animating transformations

Page items can change size, rotation or skewing angles, opacity, and visibility as their animation plays. The `animationSettings` of the page item contain properties (such as `rotationArray` or `hiddenAfter`) that define the transformations that are applied during animation. The following script fragment shows how to make a page item rotate as it follows a motion path (for the complete script, refer to AnimateRotation).

```
//Given a document "myDocument" and a page "myPage" and the color "myColorA"...
//Add a page items to animate.
var myPolygonA = myPage.polygons.add({fillColor:myColorA,
strokeColor:myDocument.swatches.item("None")});
myPolygonA.paths.item(0).entirePath = [[72, 72], [72, 144], [144, 108]];
//Create a motion path.
var myMotionPathPoints = [[[[108,108],[108,108],[108,108]],[[516, 108],[516,
108],[516, 108]]],true];
//Set animation preferences for the polygon.
myPolygonA.animationSettings.duration = 2;
myPolygonA.animationSettings.motionPathPoints = myMotionPathPoints;
//Assuming 24 Frames Per Second (FPS)
//23 = 1 second, 47 = 2 seconds, 71 = 3 seconds, 95 = 4 seconds, 119 = 5 seconds, 143 = 6
seconds
//Since the duration of our animation is 2 seconds, the following line will
//make the polygon rotate 360 degrees from the start to the end
//of the animation.
myPolygonA.animationSettings.rotationArray = [[0, 0], [47, 360]];
var myTimingSettings = myPage.parent.timingSettings;
//Remove the default timing list.
myTimingSettings.timingLists.item(0).remove();
//Add a new timing list that triggers when the page is clicked.
var myTimingList = myTimingSettings.timingLists.add(DynamicTriggerEvents.onPageClick);
//Add the polygons to a single timing group.
var myTimingGroup = myTimingList.timingGroups.add(myPolygonA, 0);
```

Scripting offers more control over animation than can be achieved with InDesign's user interface. A scripted animation can, for example, apply transformations at each key frame of a given motion path. For more on this topic, see "Key frames" later in this chapter.

Motion presets

In the preceding examples, we've constructed motion paths and specified animation settings as if we were creating animations from the basic level in InDesign's user interface. But InDesign can also use motion presets to define the animation of page items in a layout. A motion preset can apply a number of animation properties at once, as seen in the following script fragment (for the complete script, refer to MotionPreset). InDesign comes with a large number of motion presets, and you can add new presets using either the user interface or scripting.

```
//Given a page containing the ovals "myOvalA"...
var myMotionPreset = app.motionPresets.item("move-right-grow");
myOvalA.animationSettings.duration = 2;
myOvalA.animationSettings.playsLoop = true;
myOvalA.animationSettings.preset = myMotionPreset;
```

Design options

Design options affect the way that an animated object appears, relative to the motion specified in the object's animation settings. The following script fragment shows how the design options for an animated shape can affect the playback of the animation (for the complete script, refer to DesignOptions).

```
//Given a page containing the ovals "myOvalA" and "myOvalB"...
var myMotionPreset = app.motionPresets.item("move-right-grow");
myOvalA.animationSettings.duration = 2;
myOvalA.animationSettings.playsLoop = true;
myOvalA.animationSettings.preset = myMotionPreset;
myOvalA.animationSettings.designOption = DesignOptions.fromCurrentAppearance;
myOvalB.animationSettings.duration = 2;
myOvalB.animationSettings.playsLoop = true;
myOvalB.animationSettings.preset = myMotionPreset;
myOvalB.animationSettings.designOption = DesignOptions.toCurrentAppearance;
```

Key frames

Key frames are points in the timeline of an animation. With InDesign scripting, you can add key frames at any time in the animation, which gives you the ability to apply changes to objects as they are animated. Key frames are part of the motion path applied to an animated page item, and are specified relative to the duration and speed of the animation. For example, for an animation with a duration of two seconds, playing at 24 frames per second, the last frame in the animation is frame 48.

The following script fragment shows how to add key frames to a motion path, and how to change the transformations applied to an animated page item at each key frame. For the complete script, refer to TransformAnimation.

```
//Given a page containing ovals "myOvalA," "myOvalB," and "myOvalC"...
//The motion path is constructed relative to the center of the object, and key frames
//are based on the duration of the animation divided by the number of frames per second
//(usually 24). The following array sets key frames at the start, midpoint, and end
//of a path.
var myMotionPath = [[0,[[0,0], [0,0], [0,0]]], [23,[[234,0], [234,0], [234,0]]],
[47,[[468,0], [468,0], [468,0]]]];
myOvalA.animationSettings.duration = 2;
myOvalA.animationSettings.motionPath = myMotionPath;
//The transformation changes at each key frame.
//scaleXArray in the form [[keyframe, scale], [keyframe, scale], ...]
myOvalA.animationSettings.scaleXArray = [[0, 100], [23,200], [47, 100]];
//scaleYArray in the form [[keyframe, scale], [keyframe, scale], ...]
myOvalA.animationSettings.scaleYArray = [[0, 100], [23,200], [47, 100]];
//opacityArray in the form [[keyframe, opacity], [keyframe, opacity],...];
myOvalA.animationSettings.opacityArray = [[0, 100], [23, 20], [47, 100]];
myOvalA.animationSettings.playsLoop = true;
myOvalB.animationSettings.duration = 2;
myOvalB.animationSettings.motionPath = myMotionPath;
myOvalB.animationSettings.scaleXArray = [[0, 200], [23,300], [47, 50]];
myOvalB.animationSettings.scaleYArray = [[0, 200], [23,300], [47, 50]];
myOvalB.animationSettings.opacityArray = [[0, 10], [23, 80], [47, 60]];
myOvalB.animationSettings.playsLoop = true;
myOvalC.animationSettings.duration = 2;
myOvalC.animationSettings.motionPath = myMotionPath;
myOvalC.animationSettings.scaleXArray = [[0, 50], [23,200], [47, 400]];
myOvalC.animationSettings.scaleYArray = [[0, 50], [23,200], [47, 400]];
myOvalC.animationSettings.opacityArray = [[0, 100], [23, 40], [47, 80]];
myOvalC.animationSettings.playsLoop = true;
```

Adding Page Transitions

Page transitions are special effects that appear when you change pages in an exported dynamic document. Adding page transitions using scripting is easy, as shown in the following script fragment (for the complete script, refer to PageTransitions).

```
//Given a document "myDocument" containing at least two spreads...
for(var myCounter = 0; myCounter < myDocument.spreads.length; myCounter++){
    myDocument.spreads.item(myCounter).pageTransitionType =
    PageTransitionTypeOptions.pageTurnTransition;
    //This page transition option does not support the pageTransitionDirection
    //or pageTransitionDuration properties. If you chose
    //PageTransitionTypeOptions.wipeTransition (for example), you would be
    //able to set those options, as shown in the next two lines:
    //myDocument.spreads.item(myCounter).pageTransitionDirection =
    PageTransitionDirectionOptions.leftToRight;
    //myDocument.spreads.item(myCounter).pageTransitionDuration =
    PageTransitionDurationOptions.medium;
}
//Export the document to SWF, and you'll see the page transitions.
```

12 XML

Chapter Update Status	
CS6	Unchanged

Extensible Markup Language, or XML, is a text-based mark-up system created and managed by the World Wide Web Consortium (www.w3.org). Like Hypertext Markup Language (HTML), XML uses angle brackets to indicate markup tags (for example, `<article>` or `<para>`). While HTML has a predefined set of tags, XML allows you to describe content more precisely by creating custom tags.

Because of its flexibility, XML increasingly is used as a format for storing data. InDesign includes a complete set of features for importing XML data into page layouts, and these features can be controlled using scripting.

We assume that you have already read *Adobe InDesign Scripting Tutorial* and know how to create and run a script. We also assume that you have some knowledge of XML, DTDs, and XSLT.

Overview

Because XML is entirely concerned with content and explicitly *not* concerned with formatting, making XML work in a page-layout context is challenging. InDesign's approach to XML is quite complete and flexible, but it has a few limitations:

- Once XML elements are imported into an InDesign document, they become InDesign elements that correspond to the XML structure. *The InDesign representations of the XML elements are not the same thing as the XML elements themselves.*
- Each XML element can appear only once in a layout. If you want to duplicate the information of the XML element in the layout, you must duplicate the XML element itself.
- The order in which XML elements appear in a layout largely depends on the order in which they appear in the XML structure.
- Any text that appears in a story associated with an XML element becomes part of that element's data.

The Best Approach to Scripting XML in InDesign

You might want to do most of the work on an XML file outside InDesign, before you import the file into an InDesign layout. Working with XML outside InDesign, you can use a wide variety of excellent tools, such as XML editors and parsers.

When you need to rearrange or duplicate elements in a large XML data structure, the best approach is to transform the XML using XSLT. You can do this as you import the XML file.

If the XML data is already formatted in an InDesign document, you probably will want to use XML rules if you are doing more than the simplest of operations. XML rules can search the XML structure in a document and process matching XML elements much faster than a script that does not use XML rules.

For more on working with XML rules, see Chapter 13, "XML Rules."

Scripting XML Elements

This section shows how to set XML preferences and XML import preferences, import XML, create XML elements, and add XML attributes. The scripts in this section demonstrate techniques for working with the XML content itself; for scripts that apply formatting to XML elements, see "Adding XML Elements to a Layout" on page 188.

Setting XML preferences

You can control the appearance of the InDesign structure panel using the XML view-preferences object, as shown in the following script fragment (from the XMLViewPreferences tutorial script):

```
var myDocument = app.documents.add();
var myXMLViewPreferences = myDocument.xmlViewPreferences;
myXMLViewPreferences.showAttributes = true;
myXMLViewPreferences.showStructure = true;
myXMLViewPreferences.showTaggedFrames = true;
myXMLViewPreferences.showTagMarkers = true;
myXMLViewPreferences.showTextSnippets = true;
```

You also can specify XML tagging preset preferences (the default tag names and user-interface colors for tables and stories) using the XML preferences object., as shown in the following script fragment (from the XMLPreferences tutorial script):

```
var myDocument = app.documents.add();
var myXMLPreferences = myDocument.xmlPreferences;
myXMLPreferences.defaultCellTagColor = UIColors.blue;
myXMLPreferences.defaultCellTagName = "cell";
myXMLPreferences.defaultImageTagColor = UIColors.brickRed;
myXMLPreferences.defaultImageTagName = "image";
myXMLPreferences.defaultStoryTagColor = UIColors.charcoal;
myXMLPreferences.defaultStoryTagName = "text";
myXMLPreferences.defaultTableTagColor = UIColors.cuteTeal;
myXMLPreferences.defaultTableTagName = "table";
```

Setting XML import preferences

Before importing an XML file, you can set XML import preferences that can apply an XSLT transform, govern the way white space in the XML file is handled, or create repeating text elements. You do this using the XML import-preferences object, as shown in the following script fragment (from the XMLImportPreferences tutorial script):

```
var myDocument = app.documents.add();
var myXMLImportPreferences = myDocument.xmlImportPreferences;
myXMLImportPreferences.allowTransform = false;
myXMLImportPreferences.createLinkToXML = false;
myXMLImportPreferences.ignoreUnmatchedIncoming = true;
myXMLImportPreferences.ignoreWhitespace = true;
myXMLImportPreferences.importCALSTables = true;
myXMLImportPreferences.importStyle = XMLImportStyles.mergeImport;
myXMLImportPreferences.importTextIntoTables = false;
myXMLImportPreferences.importToSelected = false;
myXMLImportPreferences.removeUnmatchedExisting = false;
myXMLImportPreferences.repeatTextElements = true;
//The following properties are only used when the
//AllowTransform property is set to True.
//myXMLImportPreferences.transformFilename = "c:\myTransform.xsl"
//If you have defined parameters in your XSL file, then you can pass
//parameters to the file during the XML import process. For each parameter,
//enter an array containing two strings. The first string is the name of the
//parameter, the second is the value of the parameter.
//myXMLImportPreferences.transformParameters = [["format", "1"]];
```

Importing XML

Once you set the XML import preferences the way you want them, you can import an XML file, as shown in the following script fragment (from the ImportXML tutorial script):

```
myDocument.importXML(File("/c/xml_test.xml"));
```

When you need to import the contents of an XML file into a specific XML element, use the `importXML` method of the XML element, rather than the corresponding method of the document. See the following script fragment (from the ImportXMLIntoElement tutorial script):

```
myXMLElement.importXML(File("/c/xml_test.xml"));
```

You also can set the `importToSelected` property of the `xmlImportPreferences` object to true, then select the XML element, and then import the XML file, as shown in the following script fragment (from the ImportXMLIntoSelectedElement tutorial script):

```
var myXMLTag = myDocument.xmlTags.add("xml_element");
var myXMLElement = myDocument.xmlElements.item(0).xmlElements.add(myXMLTag);
myDocument.select(myXMLElement);
myDocument.xmlImportPreferences.importToSelected = true;
//Import into the selected XML element.
myDocument.importXML(File("/c/xml_test.xml"));
```

Creating an XML tag

XML tags are the names of the XML elements you want to create in a document. When you import XML, the element names in the XML file are added to the list of XML tags in the document. You also can create XML tags directly, as shown in the following script fragment (from the MakeXMLTags tutorial script):

```
//You can create an XML tag without specifying a color for the tag.
var myXMLTagA = myDocument.xmlTags.add("XML_tag_A");
//You can define the highlight color of the XML tag using the UIColors enumeration...
var myXMLTagB = myDocument.xmlTags.add("XML_tag_B", UIColors.gray);
//...or you can provide an RGB array to set the color of the tag.
var myXMLTagC = myDocument.xmlTags.add("XML_tag_C", [0, 92, 128]);
```

Loading XML tags

You can import XML tags from an XML file without importing the XML contents of the file. You might want to do this to work out a tag-to-style or style-to-tag mapping before you import the XML data., as shown in the following script fragment (from the LoadXMLTags tutorial script):

```
myDocument.loadXMLTags(File("/c/test.xml"));
```

Saving XML tags

Just as you can load XML tags from a file, you can save XML tags to a file, as shown in the following script. When you do this, only the tags themselves are saved in the XML file; document data is not included. As you would expect, this process is much faster than exporting XML, and the resulting file is much smaller. The following sample script shows how to save XML tags (for the complete script, see SaveXMLTags):

```
myDocument.saveXMLTags(File("/c/xml_tags.xml"), "Tag set created October 5, 2006");
```

Creating an XML element

Ordinarily, you create XML elements by importing an XML file, but you also can create an XML element using InDesign scripting, as shown in the following script fragment (from the CreateXMLElement tutorial script):

```
var myDocument = myInDesign.documents.add();
var myXMLTagA = myDocument.xmlTags.add("XML_tag_A");
var myXMLElementA = myDocument.xmlElements.item(0).xmlElements.add(myXMLTagA);
myXMLElementA.contents = "This is an XML element containing text.";
```

Moving an XML element

You can move XML elements within the XML structure using the `move` method, as shown in the following script fragment (from the MoveXMLElement tutorial script):

```
var myRootXMLElement = myDocument.xmlElements.item(0);
var myXMLElementA = myRootXMLElement.xmlElements.item(0);
myXMLElementA.move(LocationOptions.after, myRootXMLElement.xmlElements.item(2));
myRootXMLElement.xmlElements.item(-1).move(LocationOptions.atBeginning);
```

Deleting an XML element

Deleting an XML element removes it from both the layout and the XML structure, as shown in the following script fragment (from the DeleteXMLElement tutorial script).

```
myRootXMLElement.xmlElements.item(0).remove();
```

Duplicating an XML element

When you duplicate an XML element, the new XML element appears immediately after the original XML element in the XML structure, as shown in the following script fragment (from the DuplicateXMLElement tutorial script):

```
var myDocument = app.documents.item(0);
var myRootXMLElement = myDocument.xmlElements.item(0);
//Duplicate the XML element containing "A"
var myNewXMLElement = myRootXMLElement.xmlElements.item(0).duplicate();
//Change the content of the duplicated XML element.
myNewXMLElement.contents = myNewXMLElement.contents + " duplicate";
```

Removing items from the XML structure

To break the association between a page item or text and an XML element, use the `untag` method, as shown in the following script. The objects are not deleted, but they are no longer tied to an XML element (which is deleted). Any content of the deleted XML element becomes associated with the parent XML element. If the XML element is the root XML element, any layout objects (text or page items) associated with the XML element remain in the document. (For the complete script, see UntagElement.)

```
var myXMLElement = myDocument.xmlElements.item(0).xmlElements.item(0);
myXMLElement.untag();
```

Creating an XML comment

XML comments are used to make notes in XML data structures. You can add an XML comment using something like the following script fragment (from the MakeXMLComment tutorial script):

```
var myRootXMLElement = myDocument.xmlElements.item(0);
var myXMLElementB = myRootXMLElement.xmlElements.item(1);
myXMLElementB.xmlComments.add("This is an XML comment.");
```

Creating an XML processing instruction

A processing instruction (PI) is an XML element that contains directions for the application reading the XML document. XML processing instructions are ignored by InDesign but can be inserted in an InDesign XML structure for export to other applications. An XML document can contain multiple processing instructions.

An XML processing instruction has two parts, target and value. The following is an example:

```
<?xml-stylesheet type="text/css" href="generic.css"?>
```

The following script fragment shows how to add an XML processing instruction (for the complete script, see MakeProcessingInstruction):

```
var myRootXMLElement = myDocument.xmlElements.item(0);
var myXMLProcessingInstruction = myRootXMLElement.xmlInstructions.add("xml-stylesheet
type=\"text/css\" ", "href=\"generic.css\"");
```

Working with XML attributes

XML attributes are “metadata” that can be associated with an XML element. To add an XML attribute to an XML element, use something like the following script fragment (from the MakeXMLAttribute tutorial script). An XML element can have any number of XML attributes, but each attribute name must be unique within the element (that is, you cannot have two attributes named “id”).

```
var myDocument = app.documents.item(0);
var myRootXMLElement = myDocument.xmlElements.item(0);
var myXMLElementB = myRootXMLElement.xmlElements.item(1);
myXMLElementB.xmlAttributes.add("example_attribute", "This is an XML attribute. It
will not appear in the layout!");
```

In addition to creating attributes directly using scripting, you can convert XML elements to attributes. When you do this, the text contents of the XML element become the value of an XML attribute added to the parent of the XML element. Because the name of the XML element becomes the name of the attribute, this method can fail when an attribute with that name already exists in the parent of the XML element. If the XML element contains page items, those page items are deleted from the layout.

When you convert an XML attribute to an XML element, you can specify the location where the new XML element is added. The new XML element can be added to the beginning or end of the parent of the XML attribute. By default, the new element is added at the beginning of the parent element.

You also can specify am XML mark-up tag for the new XML element. If you omit this parameter, the new XML element is created with the same XML tag as XML element containing the XML attribute.

The following script shows how to convert an XML element to an XML attribute (for the complete script, see ConvertElementToAttribute):

```
var myRootXMLElement = myDocument.xmlElements.item(0);
myRootXMLElement.xmlElements.item(-1).convertToAttribute();
```

You also can convert an XML attribute to an XML element, as shown in the following script fragment (from the ConvertAttributeToElement tutorial script):

```
var myRootXMLElement = myDocument.xmlElements.item(0);
var myXMLElementB = myRootXMLElement.xmlElements.item(1);
//The "at" parameter can be either LocationOptions.atEnd or
LocationOptions.atBeginning, but cannot
//be LocationOptions.after or LocationOptions.before.
myXMLElementB.xmlAttributes.item(0).convertToElement(LocationOptions.atEnd,
myDocument.xmlTags.item("xml_element"));
```

Working with XML stories

When you import XML elements that were not associated with a layout element (a story or page item), they are stored in an XML story. You can work with text in unplaced XML elements just as you would work with the text in a text frame. The following script fragment shows how this works (for the complete script, see XMLStory):

```
var myXMLStory = myDocument.xmlStories.item(0);
//Though the text has not yet been placed in the layout, all text
//properties are available.
myXMLStory.paragraphs.item(0).pointSize = 72;
//Place the XML element in the layout to see the result.
myDocument.xmlElements.item(0).xmlElements.item(0).placeXML(myDocument.pages.item(0).
textFrames.item(0));
```

Exporting XML

To export XML from an InDesign document, export either the entire XML structure in the document or one XML element (including any child XML elements it contains). The following script fragment shows how to do this (for the complete script, see ExportXML):

```
//Export the entire XML structure in the document.
myDocument.exportFile(ExportFormat.xml, File("/c/completeDocumentXML.xml"));
//Export a specific XML element and its child XML elements.
var myXMLElement = myDocument.xmlElements.item(0).xmlElements.item(-1);
myXMLElement.exportFile(ExportFormat.xml, File("/c/partialDocumentXML.xml"));
```

In addition, you can use the `exportFromSelected` property of the `xmlExportPreferences` object to export an XML element selected in the user interface. The following script fragment shows how to do this (for the complete script, see ExportSelectedXMLElement):

```
myDocument.select(myDocument.xmlElements.item(0).xmlElements.item(1));
myDocument.xmlExportPreferences.exportFromSelected = true;
//Export the entire XML structure in the document.
myDocument.exportFile(ExportFormat.xml, File("/c/selectedXMLElement.xml"));
myDocument.xmlExportPreferences.exportFromSelected = false;
```

Adding XML Elements to a Layout

Previously, we covered the process of getting XML data into InDesign documents and working with the XML structure in a document. In this section, we discuss techniques for getting XML information into a page layout and applying formatting to it.

Associating XML elements with page items and text

To associate a page item or text with an existing XML element, use the `placeXML` method. This replaces the content of the page item with the content of the XML element, as shown in the following script fragment (from the PlaceXML tutorial script):

```
myDocument.xmlElements.item(0).placeXML(myDocument.pages.item(0).textFrames.item(0));
```

To associate an existing page item or text object with an existing XML element, use the `markup` method. This merges the content of the page item or text with the content of the XML element (if any). The following script fragment shows how to use the `markup` method (for the complete script, see Markup):

```
myDocument.xmlElements.item(0).xmlElements.item(0).markup(myDocument.pages.item(0).te
xtFrames.item(0));
```

Placing XML into page items

Another way to associate an XML element with a page item is to use the `placeIntoFrame` method. With this method, you can create a frame as you place the XML, as shown in the following script fragment (for the complete script, see PlaceIntoFrame):

```
myDocument.viewPreferences.horizontalMeasurementUnits = MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
myDocument.viewPreferences.rulerOrigin = RulerOrigin.pageOrigin;
//PlaceIntoFrame has two parameters:
//On: The page, spread, or master spread on which to create the frame
//GeometricBounds: The bounds of the new frame (in page coordinates).
myDocument.xmlElements.item(0).xmlElements.item(0).placeIntoFrame(myDocument.pages.it
em(0), [72, 72, 288, 288]);
```

To associate an XML element with an inline page item (i.e., an anchored object), use the `placeIntoCopy` method, as shown in the following script fragment (from the PlaceIntoCopy tutorial script):

```
var myPage = myDocument.pages.item(0);
var myXMLElement = myDocument.xmlElements.item(0);
myXMLElement.placeIntoCopy(myPage, [288, 72], myPage.textFrames.item(0), true);
```

To associate an existing page item (or a copy of an existing page item) with an XML element and insert the page item into the XML structure at the location of the element, use the `placeIntoInlineCopy` method, as shown in the following script fragment (from the PlaceIntoInlineCopy tutorial script):

```
var myPage = myDocument.pages.item(0);
var myTextFrame = myDocument.textFrames.add({geometricBounds:[72, 72, 96, 144]});
var myXMLElement = myDocument.xmlElements.item(0).xmlElements.item(2);
myXMLElement.placeIntoInlineCopy(myTextFrame, false);
```

To associate an XML element with a new inline frame, use the `placeIntoInlineFrame` method, as shown in the following script fragment (from the PlaceIntoInlineFrame tutorial script):

```
var myXMLElement = myDocument.xmlElements.item(0).xmlElements.item(2);
//Specify width and height as you create the inline frame.
myXMLElement.placeIntoInlineFrame([72, 24]);
```

Inserting text in and around XML text elements

When you place XML data into an InDesign layout, you often need to add white space (for example, return and tab characters) and static text (labels like "name" or "address") to the text of your XML elements. The following sample script shows how to add text in and around XML elements (for the complete script, see InsertTextAsContent):

```
var myXMLElement = myDocument.xmlElements.item(0).xmlElements.item(0);
//By inserting the return character after the XML element, the character
//becomes part of the content of the parent XML element, not of the element itself.
myXMLElement.insertTextAsContent("\r", LocationOptions.after);
myXMLElement = myDocument.xmlElements.item(0).xmlElements.item(1);
myXMLElement.insertTextAsContent("Static text: ", LocationOptions.before);
myXMLElement.insertTextAsContent("\r", LocationOptions.after);
//To add text inside the element, set the location option to beginning or end.
myXMLElement = myDocument.xmlElements.item(0).xmlElements.item(2);
myXMLElement.insertTextAsContent("Text at the start of the element: ",
LocationOptions.atBeginning);
myXMLElement.insertTextAsContent(" Text at the end of the element.",
LocationOptions.atEnd);
myXMLElement.insertTextAsContent("\r", LocationOptions.after);
//Add static text outside the element.
myXMLElement = myDocument.xmlElements.item(0).xmlElements.item(3);
myXMLElement.insertTextAsContent("Text before the element: ", LocationOptions.before);
myXMLElement.insertTextAsContent(" Text after the element.", LocationOptions.after);
//To insert text inside the text of an element, work with the text objects contained by
the element.
myXMLElement.words.item(2).insertionPoints.item(0).contents = "(the third word of) ";
```

Marking up existing layouts

In some cases, an XML publishing project does not start with an XML file—especially when you need to convert an existing page layout to XML. For this type of project, you can mark up existing page-layout content and add it to an XML structure. You can then export this structure for further processing by XML tools outside InDesign.

Mapping tags to styles

One of the quickest ways to apply formatting to XML text elements is to use `XMLImportMaps`, also known as tag-to-style mapping. When you do this, you can associate a specific XML tag with a paragraph or character style. When you use the `mapXMLTagsToStyles` method of the document, InDesign applies the style to the text, as shown in the following script fragment (from the MapTagsToStyles tutorial script):

```
var myDocument = app.documents.item(0);
//Create a tag to style mapping.
myDocument.xmlImportMaps.add(myDocument.xmlTags.item("heading_1"),
myDocument.paragraphStyles.item("heading 1"));
myDocument.xmlImportMaps.add(myDocument.xmlTags.item("heading_2"),
myDocument.paragraphStyles.item("heading 2"));
myDocument.xmlImportMaps.add(myDocument.xmlTags.item("para_1"),
myDocument.paragraphStyles.item("para 1"));
myDocument.xmlImportMaps.add(myDocument.xmlTags.item("body_text"),
myDocument.paragraphStyles.item("body text"));
//Map the XML tags to the defined styles.
myDocument.mapXMLTagsToStyles();
//Place the XML element in the layout to see the result.
var myPage = myDocument.pages.item(0);
var myTextFrame = myPage.textFrames.add({geometricBounds:myGetBounds(myDocument,
myPage)});
var myStory = myTextFrame.parentStory;
myStory.placeXML(myDocument.xmlElements.item(0));
```

Mapping styles to tags

When you have formatted text that is not associated with any XML elements, and you want to move that text into an XML structure, use style-to-tag mapping, which associates paragraph and character styles with XML tags. To do this, use `xmlExportMap` objects to create the links between XML tags and styles, then use the `mapStylesToXMLTags` method to create the corresponding XML elements, as shown in the following script fragment (from the MapStylesToTags tutorial script):

```
var myDocument = app.documents.item(0);
//Create a style to tag mapping.
myDocument.xmlExportMaps.add(myDocument.paragraphStyles.item("heading 1"),
myDocument.xmlTags.item("heading_1"));
myDocument.xmlExportMaps.add(myDocument.paragraphStyles.item("heading 2"),
myDocument.xmlTags.item("heading_2"));
myDocument.xmlExportMaps.add(myDocument.paragraphStyles.item("para 1"),
myDocument.xmlTags.item("para_1"));
myDocument.xmlExportMaps.add(myDocument.paragraphStyles.item("body text"),
myDocument.xmlTags.item("body_text"));
//Apply the style to tag mapping.
myDocument.mapStylesToXMLTags();
```

Another approach is simply to have your script create a new XML tag for each paragraph or character style in the document, and then apply the style to tag mapping, as shown in the following script fragment (from the MapAllStylesToTags tutorial script):

```
var myDocument = app.documents.item(0);
//Create tags that match the style names in the document,
//creating an XMLExportMap for each tag/style pair.
for(var myCounter = 0; myCounter<myDocument.paragraphStyles.length; myCounter++){
    var myParagraphStyle = myDocument.paragraphStyles.item(myCounter);
    var myParagraphStyleName = myParagraphStyle.name;
    var myXMLTagName = myParagraphStyleName.replace(/\ /gi, "_")
    myXMLTagName = myXMLTagName.replace(/\[/gi, "")
    myXMLTagName = myXMLTagName.replace(/\]/gi, "")
    var myXMLTag = myDocument.xmlTags.add(myXMLTagName);
    myDocument.xmlExportMaps.add(myParagraphStyle, myXMLTag);
}
//Apply the style to tag mapping.
myDocument.mapStylesToXMLTags();
```

Marking up graphics

The following script fragment shows how to associate an XML element with a graphic (for the complete script, see MarkingUpGraphics):

```
var myXMLTag = myDocument.xmlTags.add("graphic");
var myGraphic = myDocument.pages.item(0).place(File(/c/test.tif"));
//Associate the graphic with a new XML element as you create the element.
var myXMLElement = myDocument.xmlElements.Item(0).xmlElements.add(myXMLTag,
myGraphic);
```

Applying styles to XML elements

In addition to using tag-to-style and style-to-tag mappings or applying styles to the text and page items associated with XML elements, you also can apply styles to XML elements directly. The following script fragment shows how to use three methods: `applyParagraphStyle`, `applyCharacterStyle`, and `applyObjectStyle`. (For the complete script, see ApplyStylesToXMLElements.)

```
var myDocument = app.documents.add();
myDocument.viewPreferences.horizontalMeasurementUnits = MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
//Create a series of XML tags.
var myHeading1XMLTag = myDocument.xmlTags.add("heading_1");
var myHeading2XMLTag = myDocument.xmlTags.add("heading_2");
var myPara1XMLTag = myDocument.xmlTags.add("para_1");
var myBodyTextXMLTag = myDocument.xmlTags.add("body_text");
//Create a series of paragraph styles.
var myHeading1Style = myDocument.paragraphStyles.add();
myHeading1Style.name = "heading 1";
myHeading1Style.pointSize = 24;
var myHeading2Style = myDocument.paragraphStyles.add();
myHeading2Style.name = "heading 2";
myHeading2Style.pointSize = 14;
myHeading2Style.spaceBefore = 12;
var myPara1Style = myDocument.paragraphStyles.add();
myPara1Style.name = "para 1";
myPara1Style.pointSize = 12;
myPara1Style.firstLineIndent = 0;
var myBodyTextStyle = myDocument.paragraphStyles.add();
myBodyTextStyle.name = "body text";
myBodyTextStyle.pointSize = 12;
myBodyTextStyle.firstLineIndent = 24;
```

```
//Create a character style.
var myCharacterStyle = myDocument.characterStyles.add();
myCharacterStyle.name = "Emphasis";
myCharacterStyle.fontStyle = "Italic";
//Add XML elements and apply paragraph styles.
var myRootXMLElement = myDocument.xmlElements.item(0);
var myXMLElementA = myRootXMLElement.xmlElements.add(myHeading1XMLTag);
myXMLElementA.contents = "Heading 1";
myXMLElementA.insertTextAsContent("\r", XMLElementPosition.afterElement);
myXMLElementA.applyParagraphStyle(myHeading1Style, true);
var myXMLElementB = myRootXMLElement.xmlElements.add(myPara1XMLTag);
myXMLElementB.contents = "This is the first paragraph in the article.";
myXMLElementB.insertTextAsContent("\r", XMLElementPosition.afterElement);
myXMLElementB.applyParagraphStyle(myPara1Style, true);
var myXMLElementC = myRootXMLElement.xmlElements.add(myBodyTextXMLTag);
myXMLElementC.contents = "This is the second paragraph in the article.";
myXMLElementC.insertTextAsContent("\r", XMLElementPosition.afterElement);
myXMLElementC.applyParagraphStyle(myBodyTextStyle, true);
var myXMLElementD = myRootXMLElement.xmlElements.add(myHeading2XMLTag);
myXMLElementD.contents = "Heading 2";
myXMLElementD.insertTextAsContent("\r", XMLElementPosition.afterElement);
myXMLElementD.applyParagraphStyle(myHeading2Style, true);
var myXMLElementE = myRootXMLElement.xmlElements.add(myPara1XMLTag);
myXMLElementE.contents = "This is the first paragraph following the subhead.";
myXMLElementE.insertTextAsContent("\r", XMLElementPosition.afterElement);
myXMLElementE.applyParagraphStyle(myPara1Style, true);
var myXMLElementF = myRootXMLElement.xmlElements.add(myBodyTextXMLTag);
myXMLElementF.contents = "This is the second paragraph following the subhead.";
myXMLElementF.insertTextAsContent("\r", XMLElementPosition.afterElement);
myXMLElementF.applyParagraphStyle(myBodyTextStyle, true);
var myXMLElementG = myRootXMLElement.xmlElements.add(myBodyTextXMLTag);
myXMLElementG.contents = "Note:";
myXMLElementG = myXMLElementG.move(LocationOptions.atBeginning, myXMLElementF)
myXMLElementG.insertTextAsContent(" ", XMLElementPosition.afterElement);
myXMLElementG.applyCharacterStyle(myCharacterStyle, true);
// Add text elements.
var myPage = myDocument.pages.item(0);
var myTextFrame = myPage.textFrames.add({geometricBounds:myGetBounds(myDocument,
myPage)});
var myStory = myTextFrame.parentStory;
myStory.placeXML(myRootXMLElement);
```

Working with XML tables

InDesign automatically imports XML data into table cells when the data is marked up using HTML standard table tags. If you cannot use the default table mark-up or prefer not to use it, InDesign can convert XML elements to a table using the `convertElementToTable` method.

To use this method, the XML elements to be converted to a table must conform to a specific structure. Each row of the table must correspond to a specific XML element, and that element must contain a series of XML elements corresponding to the cells in the row. The following script fragment shows how to use this method (for the complete script, see ConvertXMLElementToTable). The XML element used to denote the table row is consumed by this process.

```
var myDocument = app.documents.add();
//Create a series of XML tags.
var myRowTag = myDocument.xmlTags.add("row");
var myCellTag = myDocument.xmlTags.add("cell");
var myTableTag = myDocument.xmlTags.add("table");
//Add XML elements.
var myRootXMLElement = myDocument.xmlElements.item(0);
with(myRootXMLElement){
    var myTableXMLElement = xmlElements.add(myTableTag);
    with(myTableXMLElement){
        for(var myRowCounter = 1;myRowCounter < 7;myRowCounter++){
            with(xmlElements.add(myRowTag)){
                myString = "Row " + myRowCounter;
                for(var myCellCounter = 1; myCellCounter < 5; myCellCounter++){
                    with(xmlElements.add(myCellTag)){
                        contents = myString + ":Cell " + myCellCounter;
                    }
                }
            }
        }
    }
}
var myTable = myTableXMLElement.convertElementToTable(myRowTag, myCellTag);
// Add text elements.
var myPage = myDocument.pages.item(0);
var myTextFrame = myPage.textFrames.add({geometricBounds:myGetBounds(myDocument,
myPage)});
var myStory = myTextFrame.parentStory;
myStory.placeXML(myRootXMLElement);
```

Once you are working with a table containing XML elements, you can apply table styles and cell styles to the XML elements directly, rather than having to apply the styles to the tables or cells associated with the XML elements. To do this, use the `applyTableStyle` and `applyCellStyle` methods, as shown in the following script fragment (from the ApplyTableStyles tutorial script):

```
var myDocument = app.documents.add();
//Create a series of XML tags.
var myRowTag = myDocument.xmlTags.add("row");
var myCellTag = myDocument.xmlTags.add("cell");
var myTableTag = myDocument.xmlTags.add("table");
//Create a table style and a cell style.
var myTableStyle = myDocument.tableStyles.add({name:"myTableStyle"});
myTableStyle.startRowFillColor = myDocument.colors.item("Black");
myTableStyle.startRowFillTint = 25;
myTableStyle.endRowFillColor = myDocument.colors.item("Black");
myTableStyle.endRowFillTint = 10;
var myCellStyle = myDocument.cellStyles.add();
myCellStyle.fillColor = myDocument.colors.item("Black");
myCellStyle.fillTint = 45
//Add XML elements.
var myRootXMLElement = myDocument.xmlElements.item(0);
with(myRootXMLElement){
    var myTableXMLElement = xmlElements.add(myTableTag);
    with(myTableXMLElement){
        for(var myRowCounter = 1;myRowCounter < 7;myRowCounter++){
            with(xmlElements.add(myRowTag)){
                myString = "Row " + myRowCounter;
                for(var myCellCounter = 1; myCellCounter < 5; myCellCounter++){
                    with(xmlElements.add(myCellTag)){
                        contents = myString + ":Cell " + myCellCounter;
```

```
                    }
                }
            }
        }
    }
}
var myTable = myTableXMLElement.convertElementToTable(myRowTag, myCellTag);
var myTableXMLElement = myDocument.xmlElements.item(0).xmlElements.item(0);
myTableXMLElement.applyTableStyle(myTableStyle);
myTableXMLElement.xmlElements.item(0).applyCellStyle(myCellStyle);
myTableXMLElement.xmlElements.item(5).applyCellStyle(myCellStyle);
myTableXMLElement.xmlElements.item(10).applyCellStyle(myCellStyle);
myTableXMLElement.xmlElements.item(15).applyCellStyle(myCellStyle);
myTableXMLElement.xmlElements.item(16).applyCellStyle(myCellStyle);
myTableXMLElement.xmlElements.item(21).applyCellStyle(myCellStyle);
// Add text elements.
var myPage = myDocument.pages.item(0);
var myTextFrame = myPage.textFrames.add({geometricBounds:myGetBounds(myDocument,
myPage)});
var myStory = myTextFrame.parentStory;
myStory.placeXML(myRootXMLElement);
myTable.alternatingFills = AlternatingFillsTypes.alternatingRows;
```

13 XML Rules

Chapter Update Status	
CS6	Unchanged

The InDesign XML- rules feature provides a powerful set of scripting tools for working with the XML content of your documents. XML rules also greatly simplify the process of writing scripts to work with XML elements and dramatically improve performance of finding, changing, and formatting XML elements.

While XML rules can be triggered by application events, like open, place, and close, typically you will run XML rules after importing XML into a document. (For more information on attaching scripts to events, see Chapter 8, "Events.")

This chapter gives an overview of the structure and operation of XML rules, and shows how to do the following:

- Define an XML rule.
- Apply XML rules.
- Find XML elements using XML rules.
- Format XML data using XML rules.
- Create page items based on XML rules.
- Restructure data using XML rules.
- Use the XML-rules processor.

We assume that you have already read *Adobe InDesign Scripting Tutorial* and know how to create and run a script. We also assume that you have some knowledge of XML and have read Chapter 12, "XML."

Overview

InDesign's XML rules feature has three parts:

- **XML rules processor (a scripting object)** — Locates XML elements in an XML structure using XPath and applies the appropriate XML rule(s). It is important to note that a script can contain multiple XML rule processor objects, and each rule-processor object is associated with a given XML rule set.
- **Glue code** — A set of routines provided by Adobe to make the process of writing XML rules and interacting with the XML rules-processor easier.
- **XML rules** — The XML actions you add to a script. XML rules are written in scripting code. A rule combines an XPath-based condition and a function to apply when the condition is met. The "apply" function can perform any set of operations that can be defined in InDesign scripting, including changing the XML structure; applying formatting; and creating new pages, page items, or documents.

A script can define any number of rules and apply them to the entire XML structure of an InDesign document or any subset of elements within the XML structure. When an XML rule is triggered by an XML

rule processor, the rule can apply changes to the matching XML element or any other object in the document.

You can think of the XML rules feature as being something like XSLT. Just as XSLT uses XPath to locate XML elements in an XML structure, then transforms the XML elements in some way, XML rules use XPath to locate and act on XML elements inside InDesign. Just as an XSLT template uses an XML parser outside InDesign to apply transformations to XML data, InDesign's XML Rules Processor uses XML rules to apply transformations to XML data inside InDesign.

Why use XML rules?

In prior releases of InDesign, you could not use XPath to navigate the XML structure in your InDesign files. Instead, you needed to write recursive script functions to iterate through the XML structure, examining each element in turn. This was difficult and slow.

XML rules makes it easy to find XML elements in the structure, by using XPath and relying on InDesign's XML-rules processors to find XML elements. An XML-rule processor handles the work of iterating through the XML elements in your document, and it can do so much faster than a script.

XML-rules programming model

An XML rule contains three things:

1. A name (as a string).
2. An `XPath` statement (as a string).
3. An `apply` function.

The XPath statement defines the location in the XML structure; when the XML rules processor finds a matching element, it executes the apply function defined in the rule.

Here is a sample XML rule:

```
function RuleName() {
    this.name = "RuleNameAsString";
    this.xpath = "ValidXPathSpecifier";
    this.apply = function (element, ruleSet, ruleProcessor){
        //Do something here.
        //Return true to stop further processing of the XML element
        return true;
    }; // end of Apply function
}
```

In the above example, `RuleNameAsString` is the name of the rule and matches the `RuleName`; `ValidXPathSpecifier` is an XPath expression. Later in this chapter, we present a series of functioning XML-rule examples.

NOTE: XML rules support a limited subset of XPath 1.0. See "XPath limitations" on page 200."

XML-rule sets

An XML-rule set is an array of one or more XML rules to be applied by an XML-rules processor. The rules are applied in the order in which they appear in the array. Here is a sample XML-rule set:

```
var myRuleSet = new Array (new SortByName,
                           new AddStaticText,
                           new LayoutElements,
                           new FormatElements
                           );
```

In the above example, the rules listed in the `myRuleSet` array are defined elsewhere in the script. Later in this chapter, we present several functioning scripts containing XML-rule sets.

"Glue" code

In addition to the XML-rules processor object built into InDesign's scripting model, Adobe provides a set of functions intended to make the process of writing XML rules much easier. These functions are defined within the `glue code.jsx` file:

- **`__processRuleSet(root, ruleSet)`** — To execute a set of XML rules, your script must call the `__processRuleSet` function and provide an XML element and an XML rule set. The XML element defines the point in the XML structure at which to begin processing the rules.
- **`__processChildren(ruleProcessor)`** — This function directs the XML-rules processor to apply matching XML rules to child elements of the matched XML element. This allows the rule applied to a parent XML element to execute code after the child XML elements are processed. By default, when an XML-rules processor applies a rule to the children of an XML element, control does not return to the rule. You can use the `__processChildren` function to return control to the `apply` function of the rule after the child XML elements are processed.
- **`__skipChildren(ruleProcessor)`** — This function tells the processor not to process any descendants of the current XML element using the XML rule. Use this function when you want to move or delete the current XML element or improve performance by skipping irrelevant parts of an XML structure.

Iterating through an XML structure

The XML-rules processor iterates through the XML structure of a document by processing each XML element in the order in which it appears in the XML hierarchy of the document. The XML-rules processor uses a forward-only traversal of the XML structure, and it visits each XML element in the structure twice (in the order parent-child-parent, just like the normal ordering of nested tags in an XML file). For any XML element, the XML-rules processor tries to apply all matching XML rules in the order in which they are added to the current XML rule set.

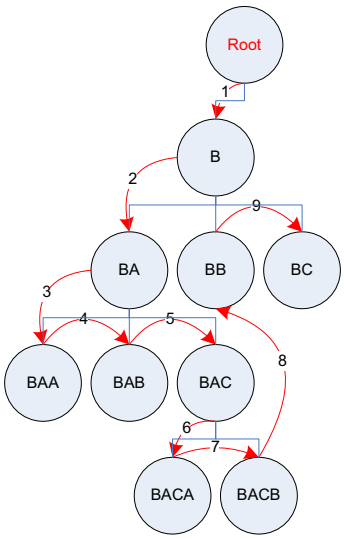
The `__processRuleSet` function applies rules to XML elements in "depth first" order; that is, XML elements and their child elements are processed in the order in which they appear in the XML structure. For each "branch" of the XML structure, the XML-rules processor visits each XML element before moving on to the next branch.

After an XML rule is applied to an XML element, the XML-rules processor continues searching for rules to apply to the descendents of that XML element. An XML rule can alter this behavior by using the `__skipChildren` or `__processChildren` function, or by changing the operation of other rules.

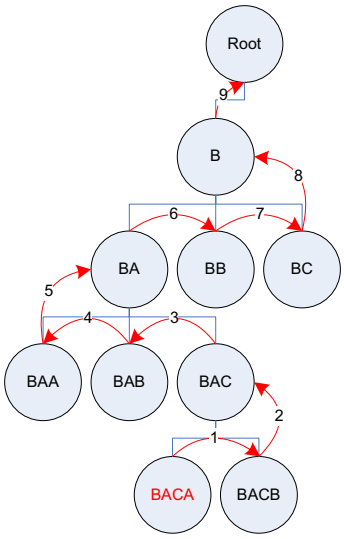
To see how all these functions work together, import the `DepthFirstProcessingOrder.xml` file into a new document, then run the `DepthFirstProcessingOrder.jsx` script. InDesign creates a text frame, that lists the attribute names of each element in the sample XML file in the order in which they were visited by each rule. You can use this script in conjunction with the AddAttribute tutorial script to troubleshoot

XML traversal problems in your own XML documents (you must edit the AddAttribute script to suit your XML structure).

Normal iteration (assuming a rule that matches every XML element in the structure) is shown in the following figure:

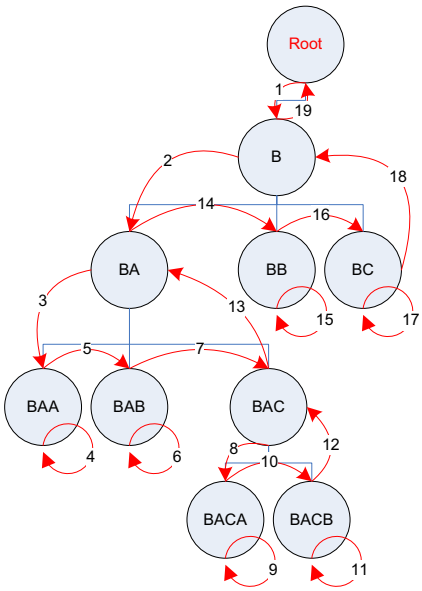


Iteration with `__processChildren` (assuming a rule that matches every XML element in the structure) is shown in the following figure:



Iteration given the following rule set is shown in the figure after the script fragment. The rule set includes two rules that match every element, including one that uses `__processChildren`. Every element is processed twice. (For the complete script, see ProcessChildren.)

```
function NormalRule(){
    this.name = "NormalRule";
    //XPath will match on every part_number XML element in the XML structure.
    this.xpath = "//XMLElement";
    // Define the apply function.
    this.apply = function(myElement, myRuleProcessor){
        app.documents.item(0).stories.item(0).insertionPoints.item(-1).contents =
        myElement.xmlAttributes.item(0).value + "\r";
        return false;
    } //End of apply function
}
function ProcessChildrenRule(){
    this.name = "ProcessChildrenRule";
    //XPath will match on every part_number XML element in the XML structure.
    this.xpath = "//XMLElement";
    // Define the apply function.
    this.apply = function(myElement, myRuleProcessor){
        __processChildren(myRuleProcessor);
        app.documents.item(0).stories.item(0).insertionPoints.item(-1).contents =
        myElement.xmlAttributes.item(0).value + "\r";
        return false;
    } //End of apply function
}
```



Changing structure during iteration

When an XML-rules processor finds a matching XML element and applies an XML rule, the rule can change the XML structure of the document. This can conflict with the process of applying other rules, if the affected XML elements in the structure are part of the current path of the XML-rules processor. To prevent errors that might cause the XML-rules processor to become invalid, the following limitations are placed on XML structure changes you might make within an XML rule:

- **Deleting an ancestor XML element** — To delete an ancestor XML element of the matched XML element, create a separate rule that matches and processes the ancestor XML element.

- **Inserting a parent XML element** — To add an ancestor XML element to the matched XML element, do so after processing the current XML element. The ancestor XML element you add is not processed by the XML-rules processor during this rule iteration (as it appears "above" the current element in the hierarchy).
- **Deleting the current XML element** — You cannot delete or move the matched XML element until any child XML elements contained by the element are processed. To make this sort of change, use the `__skipChildren` function before making the change.
- **No repetitive processing —** Changes to nodes that were already processed will not cause the XML rule to be evaluated again.

Handling multiple matching rules

When multiple rules match an XML element, the XML-rules processor can apply some or all of the matching rules. XML rules are applied in the order in which they appear in the rule set, up to the point that one of the rule `apply` functions returns `true`. In essence, returning `true` means the element was processed. Once a rule returns `true`, any other XML rules matching the XML element are ignored. You can alter this behavior and allow the next matching rule to be applied, by having the XML rule `apply` function return `false`.

When an `apply` function returns `false`, you can control the matching behavior of the XML rule based on a condition other than the `XPath` property defined in the XML rule, like the state of another variable in the script.

XPath limitations

InDesign's XML rules support a limited subset of the XPath 1.0 specification, specifically including the following capabilities:

- Find an element by name, specifying a path from the root; for example, `/doc/title`.
- Find paths with wildcards and node matches; for example, `/doc/*/subtree/node()`.
- Find an element with a specified attribute that matches a specified value; for example, `/doc/para[@font='Courier']`.
- Find an element with a specified attribute that does not match a specified value; for example, `/doc/para[@font !='Courier']`.
- Find a child element by numeric position (but not `last()`); for example, `/doc/para[3]`.
- Find self or any descendent; for example, `//para`.
- Find comment as a terminal; for example, `/doc/comment()`.
- Find PI by target or any; for example, `/doc/processing-instruction('foo')`.
- Find multiple predicates; for example, `/doc/para[@font='Courier'][@size=5][2]`.
- Find along following-sibling axes; for example, `/doc/note/following-sibling::*`.

Due to the one-pass nature of this implementation, the following XPath expressions are specifically excluded:

- No ancestor or preceding-sibling axes, including `..`, `ancestor::`, `preceding-sibling::`.

- No path specifications in predicates; for example, `foo[bar/c]`.
- No `last()` function.
- No `text()` function or text comparisons; however, you can use InDesign scripting to examine the text content of an XML element matched by a given XML rule.
- No compound Boolean predicates; for example, `foo[@bar=font or @c=size]`.
- No relational predicates; for example, `foo[@bar < font or @c > 3]`.
- No relative paths; for example, `doc/chapter`.

Error handling

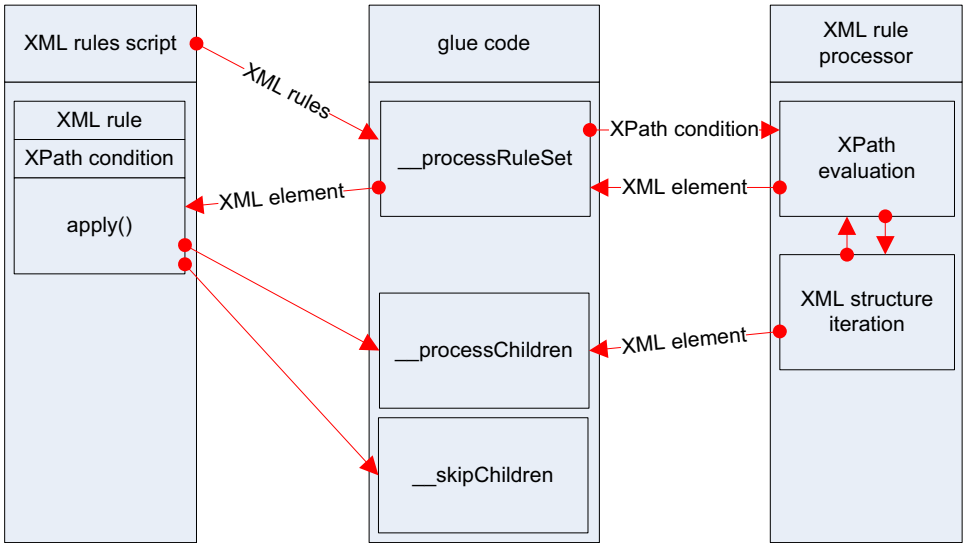
Because XML rules are part of the InDesign scripting model, scripts that use rules do not differ in nature from ordinary scripts, and they benefit from the same error-handling mechanism. When InDesign generates an error, an XML-rules script behaves no differently than any other script. InDesign errors can be captured in the script using whatever tools the scripting language provides to achieve that; for example, `try...catch` blocks.

InDesign does include a series of errors specific to XML-rules processing. An InDesign error can occur at XML-rules processor initialization, when a rule uses a non-conforming XPath specifier (see "XPath limitations" on page 200). An InDesign error also can be caused by a model change that invalidates the state of an XML-rules processor. XML structure changes caused by the operation of XML rules can invalidate the XML-rules processor. These changes to the XML structure can be caused by the script containing the XML-rules processor, another concurrently executing script, or a user action initiated from the user interface.

XML structure changes that invalidate an XML-rules processor lead to errors when the XML-rules processor's iteration resumes. The error message indicates which XML structural change caused the error.

XML rules flow of control

As a script containing XML rules executes, the flow of control passes from the script function containing the XML rules to each XML rule, and from each rule to the functions defined in the glue code. Those functions pass control to the XML-rules processor which, in turn, iterates through the XML elements in the structure. Results and errors are passed back up the chain until they are handled by a function or cause a scripting error. The following diagram provides a simplified overview of the flow of control in an XML-rules script:



XML Rules Examples

Because XML rules rely on XPath statements to find qualifying XML elements, XML rules are closely tied to the structure of the XML in a document. This means it is almost impossible to demonstrate a functional XML-rules script without having an XML structure to test it against. In the remainder of this chapter, we present a series of XML-rules exercises based on a sample XML data file. For our example, we use the product list of an imaginary integrated-circuit manufacturer. Each record in the XML data file has the following structure:

```
<device>
    <name></name>
    <type></type>
    <part_number></part_number>
    <supply_voltage>
        <minimum></minimum>
        <maximum></maximum>
    </supply_voltage>
    <package>
        <type></type>
        <pins></pins>
    </package>
    <price></price>
    <description></description>
</device>
```

The scripts are presented in order of complexity, starting with a very simple script and building toward more complex operations.

Setting up a sample document

Before you run each script in this chapter, import the `XMLRulesExampleData.xml` data file into a document. When you import the XML, turn on the Do Not Import Contents of Whitespace-Only Elements option in the XML Import Options dialog box. Save the file, then choose File > Revert before running each sample script in this section. Alternately, run the following script before you run each sample XML-rule script (see the `XMLRulesExampleSetup.jsx` script file):

```
//XMLRuleExampleSetup.jsx
//
main();
function main(){
    var myDocument = app.documents.add();
    myDocument.xmlImportPreferences.allowTransform = false;
    myDocument.xmlImportPreferences.ignoreWhitespace = true;
    var myScriptPath = myGetScriptPath();
    var myFilePath = myScriptPath.path + "/XMLRulesExampleData.xml"
    myDocument.importXML(File(myFilePath));
    var myBounds = myGetBounds(myDocument, myDocument.pages.item(0));
    myDocument.xmlElements.item(0).placeIntoFrame(myDocument.pages.item(0), myBounds);
    function myGetBounds(myDocument, myPage){
        var myWidth = myDocument.documentPreferences.pageWidth;
        var myHeight = myDocument.documentPreferences.pageHeight;
        var myX1 = myPage.marginPreferences.left;
        var myY1 = myPage.marginPreferences.top;
        var myX2 = myWidth - myPage.marginPreferences.right;
        var myY2 = myHeight - myPage.marginPreferences.bottom;
        return [myY1, myX1, myY2, myX2];
    }
    function myGetScriptPath() {
        try {
            return app.activeScript;
        }
        catch(myError){
            return File(myError.fileName);
        }
    }
}
```

Getting started with XML rules

Here is a very simple XML rule—it does nothing more than add a return character after every XML element in the document. The XML-rule set contains one rule. For the complete script, see AddReturns.

```
main();
function main(){
    if (app.documents.length != 0){
        var myDocument = app.documents.item(0);
        //This rule set contains a single rule.
        var myRuleSet = new Array (new AddReturns);
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
    }
    else{
        alert("No open document");
    }
    //Adds a return character at the end of every XML element.
    function AddReturns(){
        this.name = "AddReturns";
        //XPath will match on every XML element in the XML structure.
        this.xpath = "//*";
        // Define the apply function.
        this.apply = function(myElement, myRuleProcessor){
            with(myElement){
                //Add a return character at the end of the XML element.
                insertTextAsContent("\r", XMLElementPosition.ELEMENT_END);
            }
            return true;// Succeeded
        } //End of apply function
    }
}
```

Adding white space and static text

The following XML rule script is similar to the previous script, in that it adds white space and static text. It is somewhat more complex, however, in that it treats some XML elements differently based on their element names. For the complete script, see AddReturnsAndStaticText.

```
main();
function main(){
    if (app.documents.length != 0){
        var myDocument = app.documents.item(0);
        //This rule set contains a single rule.
        var myRuleSet = new Array (new ProcessDevice,
                                   new ProcessName,
                                   new ProcessType,
                                   new ProcessPartNumber,
                                   new ProcessSupplyVoltage,
                                   new ProcessPackageType,
                                   new ProcessPackageOne,
                                   new ProcessPackages,
                                   new ProcessPrice);
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
    }
    else{
        alert("No open document");
    }
    //Adds a return character at the end of the "device" XML element.
```

```
function ProcessDevice(){
    this.name = "ProcessDevice";
    this.xpath = "/devices/device";
    // Define the apply function.
    this.apply = function(myElement, myRuleProcessor){
        with(myElement){
            //Add a return character at the end of the XML element.
            insertTextAsContent("\r", XMLElementPosition.afterElement);
        }
        return true;// Succeeded
    } //End of apply function
}
//Adds a return character at the end of the "name" XML element.
function ProcessName(){
    this.name = "ProcessName";
    this.xpath = "/devices/device/name";
    // Define the apply function.
    this.apply = function(myElement, myRuleProcessor){
        with(myElement){
            //Add static text at the beginning of the XML element.
            insertTextAsContent("Device Name: ",
            XMLElementPosition.beforeElement);
            //Add a return character at the end of the XML element.
            insertTextAsContent("\r", XMLElementPosition.afterElement);
        }
        return true;// Succeeded
    } //End of apply function
}
//Adds a return character at the end of the "type" XML element.
function ProcessType(){
    this.name = "ProcessType";
    this.xpath = "/devices/device/type";
    // Define the apply function.
    this.apply = function(myElement, myRuleProcessor){
        with(myElement){
            //Add static text at the beginning of the XML element.
            insertTextAsContent("Circuit Type: ",
            XMLElementPosition.beforeElement);
            //Add a return character at the end of the XML element.
            insertTextAsContent("\r", XMLElementPosition.beforeElement);
        }
        return true;// Succeeded
    } //End of apply function
}
//Adds a return character at the end of the "part_number" XML element.
function ProcessPartNumber(){
    this.name = "ProcessPartNumber";
    this.xpath = "/devices/device/part_number";
    // Define the apply function.
    this.apply = function(myElement, myRuleProcessor){
        with(myElement){
            //Add static text at the beginning of the XML element.
            insertTextAsContent("Part Number: ",
            XMLElementPosition.beforeElement);
            //Add a return character at the end of the XML element.
            insertTextAsContent("\r", XMLElementPosition.afterElement);
        }
        return true;// Succeeded
    } //End of apply function
}
```

```
//Adds static text around the "minimum" and "maximum"
//XML elements of the "supply_voltage" XML element.
function ProcessSupplyVoltage(){
    this.name = "ProcessSupplyVoltage";
    this.xpath = "/devices/device/supply_voltage";
    // Define the apply function.
    this.apply = function(myElement, myRuleProcessor){
        //Note the positions at which we insert the static text. If we use
        //XMLElementPosition.elementEnd, the static text will appear
        //inside the XML element. If we use XMLElementPosition.afterElement,
        //the static text appears outside the XML elment (as a text element of
        //the parent element).
        with(myElement){
            //Add static text to the beginning of the voltage range.
            insertTextAsContent("Supply Voltage: From ",
            XMLElementPosition.elementStart);
            with(myElement.xmlElements.item(0)){
                insertTextAsContent(" to ", XMLElementPosition.afterElement);
            }
            with(myElement.xmlElements.item(-1)){
                //Add static text to the beginning of the voltage range.
                insertTextAsContent(" volts", XMLElementPosition.afterElement);
            }
            //Add a return at the end of the XML element.
            insertTextAsContent("\r", XMLElementPosition.afterElement);
        }
        return true;// Succeeded
    } //End of apply function
}
function ProcessPackageType(){
    this.name = "ProcessPackageType";
    this.xpath = "/devices/device/package/type";
    // Define the apply function.
    this.apply = function(myElement, myRuleProcessor){
        with(myElement){
            insertTextAsContent("-", XMLElementPosition.afterElement);
        }
        return true;
    } //End of apply function
}
//Add the text "Package:" before the list of packages.
function ProcessPackageOne(){
    this.name = "ProcessPackageOne";
    this.xpath = "/devices/device/package[1]";
    this.apply = function(myElement, myRuleProcessor){
        with(myElement){
            insertTextAsContent("Package: ", XMLElementPosition.elementStart);
        }
        return false; //Return false to let other XML rules process the element.
    } //End of apply function
}
//Add commas between the package types.
function ProcessPackages(){
    this.name = "ProcessPackages";
    this.xpath = "/devices/device/package";
    this.apply = function(myElement, myRuleProcessor){
        with(myElement){
            if(myElement.parent.xmlElements.nextItem(myElement).markupTag.name ==
            "package"){
                insertTextAsContent(", ", XMLElementPosition.elementEnd);
```

```
            }
            else{
                insertTextAsContent("\r", XMLElementPosition.afterElement);
            }
        }
        return true;
    } //End of apply function
}
function ProcessPrice(){
    this.name = "ProcessPrice";
    this.xpath = "/devices/device/price";
    // Define the apply function.
    this.apply = function(myElement, myRuleProcessor){
        with(myElement){
            insertTextAsContent("Price: $", XMLElementPosition.beforeElement);
            //Add a return at the end of the XML element.
            insertTextAsContent("\r", XMLElementPosition.afterElement);
        }
        return true;// Succeeded
    } //End of apply function
}
}
```

NOTE: The above script uses scripting logic to add commas between repeating elements (in the `ProcessPackages` XML rule). If you have a sequence of similar elements at the same level, you can use forward-axis matching to do the same thing. Given the following example XML structure:

```
<xmlElement><item>1</item><item>2</item><item>3</item><item>4</item>
</xmlElement>
```

To add commas between each `item` XML element in a layout, you could use an XML rule like the following (from the ListProcessing tutorial script):

```
var myRuleSet = new Array (new ListItems);
var myDocument = app.documents.item(0);
with(myDocument){
    var elements = xmlElements;
    __processRuleSet(elements.item(0), myRuleSet);
}
//Add commas between each "item" element.
function ListItems(){
    this.name = "ListItems";
    //Match all following sibling XML elements
    //of the first "item" XML element.
    this.xpath = "/xmlElement/item[1]/following-sibling::*";
    this.apply = function(myElement, myRuleProcessor){
        with(myElement){
            insertTextAsContent(", ", XMLElementPosition.beforeElement);
        }
        return false; //Let other XML rules process the element.
    }
}
```

Changing the XML structure using XML rules

Because the order of XML elements is significant in InDesign's XML implementation, you might need to use XML rules to change the sequence of elements in the structure. In general, large-scale changes to the

structure of an XML document are best done using an XSLT file to transform the document before or during XML import into InDesign.

The following XML rule script shows how to use the `move` method to accomplish this. Note the use of the `__skipChildren` function from the glue code to prevent the XML-rules processor from becoming invalid. For the complete script, see MoveXMLElement.

```
main();
function main(){
    if (app.documents.length != 0){
        var myDocument = app.documents.item(0);
        //This rule set contains a single rule.
        var myRuleSet = new Array (new MoveElement);
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
    }
    else{
        alert("No open document");
    }
    //Adds a return character at the end of every XML element.
    function MoveElement(){
        this.name = "MoveElement";
        //XPath will match on every part_number XML element in the XML structure.
        this.xpath = "/devices/device/part_number";
        // Define the apply function.
        this.apply = function(myElement, myRuleProcessor){
            //Moves the part_number XML element to the start of
            //the device XML element (the parent).
            __skipChildren(myRuleProcessor);
            myElement.move(LocationOptions.before,
            myElement.parent.xmlElements.item(0));
            return true;// Succeeded
        } //End of apply function
    }
}
```

Duplicating XML elements with XML rules

As discussed in Chapter 12, "XML," XML elements have a one-to-one relationship with their expression in a layout. If you want the content of an XML element to appear more than once in a layout, you need to duplicate the element. The following script shows how to duplicate elements using XML rules. For the complete script, see DuplicateXMLElement. Again, this rule uses `__skipChildren` to avoid invalid XML object references.

```
#include "glue code.jsx"
main();
function main(){
    if (app.documents.length != 0){
        var myDocument = app.documents.item(0);
        //This rule set contains a single rule.
        var myRuleSet = new Array (new DuplicateElement);
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
    }
    else{
        alert("No open document");
    }
    //Duplicates the part number element in each XML element.
    function DuplicateElement(){
        this.name = "DuplicateElement";
        this.xpath = "/devices/device/part_number";
        this.apply = function(myElement, myRuleProcessor){
            //Duplicates the part_number XML element.
            __skipChildren(myRuleProcessor);
            myElement.duplicate();
            return true;
        }
    }
}
```

XML rules and XML attributes

The following XML rule adds attributes to XML elements based on the content of their "name" element. When you need to find an element by its text contents, copying or moving XML element contents to XML attributes attached to their parent XML element can be very useful in XML-rule scripting. While the subset of XPath supported by XML rules cannot search the text of an element, it can find elements by a specified attribute value. For the complete script, see AddAttribute.

```
main();
function main(){
    if (app.documents.length != 0){
        var myDocument = app.documents.item(0);
        var myRuleSet = new Array (new AddAttribute);
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
    }
    else{
        alert("No open document");
    }
    function AddAttribute(){
        this.name = "AddAttribute";
        this.xpath = "/devices/device/part_number";
        this.apply = function(myElement, myRuleProcessor){
            myElement.parent.xmlAttributes.add("part_number",
            myElement.texts.item(0).contents);
            return true;
        }
    }
}
```

In the previous XML rule, we copied the data from an XML element into an XML attribute attached to its parent XML element. Instead, what if we want to move the XML element data into an attribute and remove the XML element itself? Use the `convertToAttribute` method, as shown in the following script (from the ConvertToAttribute tutorial script):

```
main();
function main(){
    if (app.documents.length != 0){
        var myDocument = app.documents.item(0);
        var myRuleSet = new Array (new ConvertToAttribute);
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
    }
    else{
        alert("No open document");
    }
    //Converts all part_number XML elements to XML attributes.
    function ConvertToAttribute(){
        this.name = "ConvertToAttribute";
        this.xpath = "/devices/device/part_number";
        // Define the apply function.
        this.apply = function(myElement, myRuleProcessor){
            //Use __skipChildren to prevent the XML rule processor from becoming
            //invalid when we convert the XML element to an attribute.
            __skipChildren(myRuleProcessor);
            //Converts the XML element to an XML attribute of its parent XML element.
            myElement.convertToAttribute("PartNumber");
            return true;
        }
    }
}
```

To move data from an XML attribute to an XML element, use the `convertToElement` method, as described in Chapter 12, "XML."

Applying multiple matching rules

When the `apply` function of an XML rule returns true, the XML-rules processor does not apply any further XML rules to the matched XML element. When the `apply` function returns false, however, the XML-rules processor can apply other rules to the XML element. The following script shows an example of an XML-rule `apply` function that returns false. This script contains two rules that will match every XML element in the document. The only difference between them is that the first rule applies a color and returns false, while the second rule applies a different color to every other XML element (based on the state of a variable, `myCounter`). For the complete script, see ReturningFalse.

```
main();
function main(){
    myCounter = 0;
    if (app.documents.length != 0){
        var myDocument = app.documents.item(0);
        //Define two colors.
        var myColorA = myDocument.colors.add({model:ColorModel.process,
        colorValue:[0, 100, 80, 0], name:"ColorA"});
        var myColorB = myDocument.colors.add({model:ColorModel.process,
        colorValue:[100, 0, 80, 0], name:"ColorB"})
        var myRuleSet = new Array (new ReturnFalse,
                            new ReturnTrue);
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
    }
    else{
        alert("No open document");
    }
    //Adds a color to the text of every element in the structure.
    function ReturnFalse(){
        this.name = "ReturnFalse";
        //XPath will match on every XML element in the XML structure.
        this.xpath = "//*";
        this.apply = function(myElement, myRuleProcessor){
            with(myElement){
                myElement.texts.item(0).fillColor =
                app.documents.item(0).colors.item("ColorA");
            }
            // Leaves the XML element available to further processing.
            return false;
        }
    }
    //Adds a color to the text of every other element in the structure.
    function ReturnTrue(){
        this.name = "ReturnTrue";
```

```
        //XPath will match on every XML element in the XML structure.
        this.xpath = "//*";
        this.apply = function(myElement, myRuleProcessor){
            with(myElement){
                if(myCounter % 2 == 0){
                    myElement.texts.item(0).fillColor =
                    app.documents.item(0).colors.item("ColorB");
                }
                myCounter++;
            }
            //Do not process the element with any further matching rules.
            return true;
        }
    }
}
```

Finding XML elements

As noted earlier, the subset of XPath supported by XML rules does not allow for searching the text contents of XML elements. To get around this limitation, you can either use attributes to find the XML elements you want or search the text of the matching XML elements. The following script shows how to match XML elements using attributes. This script applies a color to the text of elements it finds, but a practical script would do more. For the complete script, see FindXMLElementByAttribute.

```
main();
function main(){
    if (app.documents.length != 0){
        var myDocument = app.documents.item(0);
        var myRuleSet = new Array(new AddAttribute);
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
        //Now that the attributes have been added, find and format
        //the XML element whose attribute content matches a specific string.
        var myRuleSet = new Array(new FindAttribute);
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
    }
    else{
        alert("No open document");
    }
    function AddAttribute(){
        this.name = "AddAttribute";
        this.xpath = "/devices/device/part_number";
        this.apply = function(myElement, myRuleProcessor){
```

```
            myElement.parent.xmlAttributes.add("part_number",
            myElement.texts.item(0).contents);
            return true;
        }
    }
    function FindAttribute(){
        this.name = "FindAttribute";
        this.xpath = "/devices/device[@part_number = 'DS001']";
        this.apply = function(myElement, myRuleProcessor){
            myElement.xmlElements.item(0).texts.item(0).fillColor =
            app.documents.item(0).swatches.item(-1);
            return true;
        }
    }
}
```

The following script shows how to use a JavaScript regular expression (RegExp) to find and format XML elements by their content (for the complete script, see FindXMLElementByRegExp):

```
main();
function main(){
    if (app.documents.length != 0){
        var myDocument = app.documents.item(0);
        var myRuleSet = new Array (new FindByContent);
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
    }
    else{
        alert("No open document");
    }
    function FindByContent(){
        //Find descriptions that contain both "triangle" and "pulse".
        var myRegExp = /triangle.*?pulse|pulse.*?triangle/i
        this.name = "FindByContent";
        //XPath will match on every description in the XML structure.
        this.xpath = "/devices/device/description";
        this.apply = function(myElement, myRuleProcessor){
            if(myRegExp.test(myElement.texts.item(0).contents) == true){
                myElement.texts.item(0).fillColor =
app.documents.item(0).swatches.item(-1);
            }
            return true;
        }
    }
    function myResetFindChangeGrep(){
        app.findGrepPreferences = NothingEnum.nothing;
        app.changeGrepPreferences = NothingEnum.nothing;
    }
}
```

The following script shows how to use the `findText` method to find and format XML content (for the complete script, see FindXMLElementByFindText):

```
main();
function main(){
    if (app.documents.length != 0){
        var myDocument = app.documents.item(0);
        var myRuleSet = new Array (new FindByFindText);
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
    }
    else{
        alert("No open document");
    }
    function FindByFindText(){
        this.name = "FindByFindText";
        this.xpath = "/devices/device/description";
        this.apply = function(myElement, myRuleProcessor){
            if(myElement.texts.item(0).contents != ""){
                //Clear the find text preferences.
                myResetFindText();
                //Search for the word "triangle" in the content of the element.
                app.findTextPreferences.findWhat = "triangle";
                var myFoundItems = myElement.texts.item(0).findText();
                if(myFoundItems.length != 0){
                    myElement.texts.item(0).fillColor =
app.documents.item(0).swatches.item(-1);
                }
                myResetFindText();
            }
                return true;
        }
    }
    function myResetFindText(){
        app.findTextPreferences = NothingEnum.nothing;
        app.changeTextPreferences = NothingEnum.nothing;
    }
}
```

The following script shows how to use the `findGrep` method to find and format XML content (for the complete script, see FindXMLElementByFindGrep):

```
main();
function main(){
    if (app.documents.length != 0){
        var myDocument = app.documents.item(0);
        var myRuleSet = new Array (new FindByContent);
        myResetFindChangeGrep();
        app.findGrepPreferences.findWhat = "(?i)pulse.*?triangle|triangle.*?pulse";
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
        myResetFindChangeGrep();
    }
    else{
        alert("No open document");
    }
    function FindByContent(){
        //Find descriptions that contain both "triangle" and "pulse".
        this.name = "FindByContent";
```

```
        //XPath will match on every description in the XML structure.
        this.xpath = "/devices/device/description";
        // Define the apply function.
        this.apply = function(myElement, myRuleProcessor){
            var myFoundItems = myElement.texts.item(0).findGrep();
            if(myFoundItems.length != 0){
                myElement.texts.item(0).fillColor =
app.documents.item(0).swatches.item(-1);
            }
            return true;
        }
    }
    function myResetFindChangeGrep(){
        app.findGrepPreferences = NothingEnum.nothing;
        app.changeGrepPreferences = NothingEnum.nothing;
    }
}
```

Extracting XML elements with XML rules

XSLT often is used to extract a specific subset of data from an XML file. You can accomplish the same thing using XML rules. The following sample script shows how to duplicate a set of sample XML elements and move them to another position in the XML element hierarchy. Note that you must add the duplicated XML elements at a point in the XML structure that will not be matched by the XML rule, or you run the risk of creating an endless loop. For the complete script, see ExtractSubset.

```
main();
function main(){
    if (app.documents.length != 0){
        var myDocument = app.documents.item(0);
        var myRuleSet = new Array (new ExtractVCO);
        var myMarkupTag = myDocument.xmlTags.add("VCOs");
        var myContainerElement =
        myDocument.xmlElements.item(0).xmlElements.add(myMarkupTag);
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
    }
    else{
        alert("No open document");
    }
    function ExtractVCO(){
        var myNewElement;
        this.name = "ExtractVCO";
        this.xpath = "/devices/device/type";
        this.apply = function(myElement, myRuleProcessor){
            with(myElement){
                if(myElement.texts.item(0).contents == "VCO"){
                    myNewElement = myElement.parent.duplicate();
                    myNewElement.move(LocationOptions.atEnd,
app.documents.item(0).xmlElements.item(0).xmlElements.item(-1));
                }
            }
            return true;
        }
    }
}
```

Applying formatting with XML rules

The previous XML-rule examples have shown basic techniques for finding XML elements, rearranging the order of XML elements, and adding text to XML elements. Because XML rules are part of scripts, they can perform almost any action—from applying text formatting to creating entirely new page items, pages, and documents. The following XML-rule examples show how to apply formatting to XML elements using XML rules and how to create new page items based on XML-rule matching.

The following script adds static text and applies formatting to the example XML data (for the complete script, see XMLRulesApplyFormatting):

```
main();
function main(){
    if (app.documents.length != 0){
        var myDocument = app.documents.item(0);
        //Document set-up.
        myDocument.viewPreferences.horizontalMeasurementUnits =
        MeasurementUnits.points;
        myDocument.viewPreferences.verticalMeasurementUnits =
        MeasurementUnits.points;
        myDocument.colors.add({model:ColorModel.process,
        colorValue:[0, 100, 100, 0], name:"Red"});
        myDocument.paragraphStyles.add({name:"DeviceName", pointSize:24,
        leading:24, spaceBefore:24, fillColor:"Red", paragraphRuleAbove:true});
        myDocument.paragraphStyles.add({name:"DeviceType", pointSize:12,
        fontStyle:"Bold", leading:12});
        myDocument.paragraphStyles.add({name:"PartNumber", pointSize:12,
        fontStyle:"Bold", leading:12});
        myDocument.paragraphStyles.add({name:"Voltage", pointSize:10, leading:12});
        myDocument.paragraphStyles.add({name:"DevicePackage", pointSize:10,
        leading:12});
        myDocument.paragraphStyles.add({name:"Price", pointSize:10, leading:12,
        fontStyle:"Bold"});
        var myRuleSet = new Array (new ProcessDevice,
                                   new ProcessName,
                                   new ProcessType,
                                   new ProcessPartNumber,
                                   new ProcessSupplyVoltage,
                                   new ProcessPackageType,
                                   new ProcessPackageOne,
                                   new ProcessPackages,
                                   new ProcessPrice);
        with(myDocument){
            var elements = xmlElements;
            __processRuleSet(elements.item(0), myRuleSet);
        }
    }
    else{
        alert("No open document");
    }
    function ProcessDevice(){
        this.name = "ProcessDevice";
        this.xpath = "/devices/device";
        this.apply = function(myElement, myRuleProcessor){
            with(myElement){
                insertTextAsContent("\r", XMLElementPosition.afterElement);
            }
            return true;
        }
```

```
}
function ProcessName(){
    this.name = "ProcessName";
    this.xpath = "/devices/device/name";
    this.apply = function(myElement, myRuleProcessor){
        var myDocument = app.documents.item(0);
        with(myElement){
            insertTextAsContent("\r", XMLElementPosition.afterElement);
            applyParagraphStyle(myDocument.paragraphStyles.
            item("DeviceName"));
        }
        return true;
    }
}
function ProcessType(){
    this.name = "ProcessType";
    this.xpath = "/devices/device/type";
    this.apply = function(myElement, myRuleProcessor){
        var myDocument = app.documents.item(0);
        with(myElement){
            insertTextAsContent("Circuit Type: ", XMLElementPosition.beforeElement);
            insertTextAsContent("\r", XMLElementPosition.afterElement);
            applyParagraphStyle(myDocument.paragraphStyles.
            item("DeviceType"));
        }
        return true;
    }
}
function ProcessPartNumber(){
    this.name = "ProcessPartNumber";
    this.xpath = "/devices/device/part_number";
    this.apply = function(myElement, myRuleProcessor){
        var myDocument = app.documents.item(0);
        with(myElement){
            //Add static text at the beginning of the XML element.
            insertTextAsContent("Part Number: ", XMLElementPosition.beforeElement);
            //Add a return character at the end of the XML element.
            insertTextAsContent("\r", XMLElementPosition.afterElement);
            applyParagraphStyle(myDocument.paragraphStyles.
            item("PartNumber"));
        }
        return true;
    }
}
//Adds static text around the "minimum" and "maximum"
//XML elements of the "supply_voltage" XML element.
function ProcessSupplyVoltage(){
    this.name = "ProcessSupplyVoltage";
    this.xpath = "/devices/device/supply_voltage";
    this.apply = function(myElement, myRuleProcessor){
        var myDocument = app.documents.item(0);
        //Note the positions at which we insert the static text.
        //If we use XMLElementPosition.elementEnd, the static text
        //will appear inside the XML element. If we use
        //XMLElementPosition.afterElement, the static text appears
        //outside the XML elment (as a text element of the parent element).
        with(myElement){
            //Add static text to the beginning of the voltage range.
            insertTextAsContent("Supply Voltage: From ",
            XMLElementPosition.beforeElement);
```

```
            with(myElement.xmlElements.item(0)){
                insertTextAsContent(" to ", XMLElementPosition.afterElement);
            }
            with(myElement.xmlElements.item(-1)){
                //Add static text to the beginning of the voltage range.
                insertTextAsContent(" volts", XMLElementPosition.afterElement);
            }
            //Add a return at the end of the XML element.
            insertTextAsContent("\r", XMLElementPosition.afterElement);
            applyParagraphStyle(myDocument.paragraphStyles.item("Voltage"));
        }
        return true;
    }
}
function ProcessPackageType(){
    this.name = "ProcessPackageType";
    this.xpath = "/devices/device/package/type";
    this.apply = function(myElement, myRuleProcessor){
        var myDocument = app.documents.item(0);
        with(myElement){
            insertTextAsContent("-", XMLElementPosition.afterElement);
        }
        return true;
    }
}
//Add the text "Package:" before the list of packages.
function ProcessPackageOne(){
    this.name = "ProcessPackageOne";
    this.xpath = "/devices/device/package[1]";
    this.apply = function(myElement, myRuleProcessor){
        with(myElement){
            insertTextAsContent("Package: ", XMLElementPosition.beforeElement);
        }
        return false; //Return false to let other XML rules process the element.
    }
}
//Add commas between the package types.
function ProcessPackages(){
    this.name = "ProcessPackages";
    this.xpath = "/devices/device/package";
    this.apply = function(myElement, myRuleProcessor){
        var myDocument = app.documents.item(0);
        with(myElement){
            if(myElement.parent.xmlElements.nextItem(myElement).
            markupTag.name == "package"){
                insertTextAsContent(", ", XMLElementPosition.afterElement);
            }
            else{
                insertTextAsContent("\r", XMLElementPosition.afterElement);
                applyParagraphStyle(myDocument.paragraphStyles.
                item("DevicePackage"));
            }
        }
        return true;
    }
}
```

```
    function ProcessPrice(){
        this.name = "ProcessPrice";
        this.xpath = "/devices/device/price";
        this.apply = function(myElement, myRuleProcessor){
            var myDocument = app.documents.item(0);
            with(myElement){
                insertTextAsContent("Price: $", XMLElementPosition.beforeElement);
                //Add a return at the end of the XML element.
                insertTextAsContent("\r", XMLElementPosition.afterElement);
                applyParagraphStyle(myDocument.paragraphStyles.item("Price"));
            }
            return true;
        }
    }
}
```

Creating page items with XML rules

The following script creates new page items, inserts the content of XML elements in the page items, adds static text, and applies formatting. We include only the relevant XML-rule portions of the script here; for more information, see the complete script (XMLRulesLayout).

The first rule creates a new text frame for each "device" XML element:

```
//Creates a new text frame on each page.
function ProcessDevice(){
    this.name = "ProcessDevice";
    this.xpath = "/devices/device";
    this.apply = function(myElement, myRuleProcessor){
        var myDocument = app.documents.item(0);
        with(myElement){
            insertTextAsContent("\r", XMLElementPosition.afterElement);
            if(myDocument.pages.item(0).textFrames.length == 0){
                myPage = myDocument.pages.item(0);
            }
            else{
                myPage = myDocument.pages.add();
            }
            var myBounds = myGetBounds(myDocument, myPage);
            var myTextFrame = placeIntoFrame(myPage, myBounds);
            myTextFrame.textFramePreferences.firstBaselineOffset =
            FirstBaseline.leadingOffset;
        }
        return true;
    }
}
```

The "ProcessType" rule moves the "type" XML element to a new frame on the page:

```
//Creates a new text frame at the top of the page to contain the "type" XML element.
function ProcessType(){
    this.name = "ProcessType";
    this.xpath = "/devices/device/type";
    this.apply = function(myElement, myRuleProcessor){
        var myDocument = app.documents.item(0);
        with(myElement){
            var myBounds = myGetBounds(myDocument, myDocument.pages.item(-1));
            myBounds = [myBounds[0]-24, myBounds[1], myBounds[0], myBounds[2]];
            var myTextFrame = placeIntoFrame(myPage, myBounds);
            applyParagraphStyle(myDocument.paragraphStyles.item("DeviceType"));
            myTextFrame.textFramePreferences.insetSpacing = [6, 6, 6, 6];
            myTextFrame.fillColor = myDocument.swatches.item("Red")
        }
        return true;
    }
}
```

Creating Tables using XML Rules

You can use the `convertElementToTable` method to turn an XML element into a table. This method has a limitation in that it assumes that all of the XML elements inside the table conform to a very specific set of XML tags—one tag for a row element; another for a cell, or column element. Typically, the XML data we want to put into a table does not conform to this structure: it is likely that the XML elements we want to arrange in columns use heterogeneous XML tags (price, part number, etc.).

To get around this limitation, we can "wrap" each XML element we want to add to a table row using a container XML element, as shown in the following script fragments (see XMLRulesTable). In this example, a specific XML rule creates an XML element for each row.

```
function ProcessDevice(){
    this.name = "ProcessDevice";
    this.xpath = "//device[@type = 'VCO']";
    this.apply = function(myElement, myRuleProcessor){
            var myNewElement = myContainerElement.xmlElements.add(
        app.documents.item(0).xmlTags.item("Row"));
        return true;
    }
}
```

Successive rules move and format their content into container elements inside the row XML element.

```
function ProcessPrice(){
       this.name = "ProcessPrice";
       this.xpath = "//device[@type = 'VCO']/price";
       this.apply = function(myElement, myRuleProcessor){
           with(myElement){
               __skipChildren(myRuleProcessor);
               var myNewElement = myContainerElement.xmlElements.item(-1)
           .xmlElements.add(app.documents.item(0).xmlTags.item("Column"));
               var myElement = myElement.move(LocationOptions.atBeginning,
           myNewElement);
               myElement.insertTextAsContent("$",
           XMLElementPosition.beforeElement);
           }
           return true;
       }
   }
}
```

Once all of the specified XML elements have been "wrapped," we can convert the container element to a table.

```
var myTable = myContainerElement.convertElementToTable(myRowTag, myColumnTag);
```

Scripting the XML-rules Processor Object

While we have provided a set of utility functions in `glue code.jsx`, you also can script the XML-rules processor object directly. You might want do this to develop your own support routines for XML rules or to use the XML-rules processor in other ways.

When you script XML elements outside the context of XML rules, you cannot locate elements using XPath. You can, however, create an XML rule that does nothing more than return matching XML elements, and apply the rule using an XML-rules processor, as shown in the following script. (This script uses the same XML data file as the sample scripts in previous sections.) For the complete script, see XMLRulesProcessor.

```
main();
function main(){
    var myXPath = ["/devices/device"];
    var myXMLMatches = mySimulateXPath(myXPath);
    //At this point, myXMLMatches contains all of the XML elements
    //that matched the XPath expression provided in myXPath.
    function mySimulateXPath(myXPath){
        var myXMLElements = new Array;
        var myRuleProcessor = app.xmlRuleProcessors.add(myXPath);
        try{
            var myMatchData = myRuleProcessor.startProcessingRuleSet(app.documents.
            item(0).xmlElements.item(0));
            while(myMatchData != undefined){
                var myElement = myMatchData.element;
                myXMLElements.push(myElement);
                myMatchData = myRuleProcessor.findNextMatch();
            }
            myRuleProcessor.endProcessingRuleSet();
            myRuleProcessor.remove();
            return myXMLElements;
        }
        catch (myError){
            myRuleProcessor.endProcessingRuleSet();
            myRuleProcessor.remove();
            throw myError;
        }
    }
}
```

14 Track Changes

Chapter Update Status	
CS6	Unchanged

Writers can track, show, hide, accept, and reject changes as a document moves through the writing and editing process. All changes are recorded and visualized to make it easier to review a document.

This tutorial shows how to script the most common operations involving tracking changes.

We assume that you have already read *Adobe InDesign Scripting Tutorial* and know how to create, install, and run a script. We also assume that you have some knowledge of working with text in InDesign and understand basic typesetting terms.

Tracking Changes

This section shows how to navigate tracked changes, accept changes, and reject changes using scripting.

Whenever anyone adds, deletes, or moves text within an existing story, the change is marked in galley and story views.

Navigating tracked changes

If the story contains a record of tracked changes, the user can navigate sequentially through tracked changes. The following script show how to navigate the tracked changes (for the complete script, refer to GetTrackchange).

The script below uses the `nextItem` method to navigate to the change following the insertion point:

```
var myDocument = app.documents.item(0);
var myStory = myDocument.stories.item(0);
//Story.trackChanges   If true, track changes is turned on.
if(myStory.trackChanges==true)
{
    var myChangeCount = myStory.changes.length;
    var myChange = myStory.changes.item(0);
    if(myChangeCount>1)
    {
        var myChange0 = myStory.changes.nextItem(myChange);
    }
}
```

In the script below, we use the `previousItem` method to navigate to the change following the insertion point:

```
var myDocument = app.documents.item(0);
var myStory = myDocument.stories.item(0);
//Story.trackChanges   If true, track changes is turned on.
if(myStory.trackChanges==true)
{
    var myChangeCount = myStory.changes.length;
    var myChange = myStory.changes.lastItem();
    if(myChangeCount>1)
    {
        var myChange0 = myStory.changes.previousItem(myChange);
    }
}
```

Accepting and reject tracked changes

When changes are made to a story, by you or others, the change-tracking feature enables you to review all changes and decide whether to incorporate them into the story. You can accept and reject changes—added, deleted, or moved text—made by any user.

In the following script, the change is accepted (for the complete script, refer to AcceptChange):

```
var myDocument = app.documents.item(0);
var myStory = myDocument.stories.item(0);
var myChange = myStory.changes.item(0);
myChange.accept() ;
```

In the following script, the change is rejected (for the complete script, refer to RejectChange):

```
var myDocument = app.documents.item(0);
var myStory = myDocument.stories.item(0);
var myChange = myStory.changes.item(0);
myChange.reject() ;
```

Information about tracked changes

Change information includes include date and time. The following script shows the information of a tracked change (for the complete script, refer to GetChangeInfo):

```
var myDocument = app.documents.item(0);
var myStory = myDocument.stories.item(0);
var myChange = myStory.changes.item(0);
//ChangeTypes.DELETED_TEXT  (Read Only) Deleted text.
//ChangeTypes.DELETED_TEXT  (Read Only) Deleted text.
//ChangeTypes.MOVED_TEXT  (Read Only) Moved text.
var myChangeTypes = myChange.changeType;
//Characters  A collection of characters.
var myCharacters = myChange.characters;
var myDate = myChange.date;
//InsertionPoints A collection of insertion points.
var myInsertionPoints = myChange.insertionPoints;
//Change.lines  (Read Only) A collection of lines.
var myLines = myChange.lines;
//Change.paragraphs  (Read Only) A collection of paragraphs.
var myParagraphs = myChange.paragraphs;
var myStoryOffset = myChange.storyOffset;
//Change.textColumns  (Read Only) A collection of text columns.
var myTextColumns = myChange.textColumns;
//Change.textStyleRanges  (Read Only) A collection of text style ranges.
var myTextStyleRanges = myChange.textStyleRanges;
//Change.textVariableInstances  (Read Only) A collection of text variable instances.
var myTextVariableInstances = myChange.textVariableInstances;
//Change.texts  (Read Only) A collection of text objects.
var myTexts = myChange.texts;
var myUserName = myChange.userName;
var myWords = myChange.words;
```

Preferences for Tracking Changes

Track-changes preferences are user settings for tracking changes. For example, you can define which changes are tracked (adding, deleting, or moving text). You can specify the appearance of each type of tracked change, and you can have changes identified with colored change bars in the margins. The following script shows how to set and get these preferences (for the complete script, refer to GetChangePreference):

```
var myTrackChangesPreference = app.trackChangesPreferences;
with(myTrackChangesPreference)
{
    addedBackgroundColorChoice =
ChangeBackgroundColorChoices.CHANGE_BACKGROUND_USES_CHANGE_PREF_COLOR;
    addedTextColorChoice = ChangeTextColorChoices.CHANGE_USES_CHANGE_PREF_COLOR;
    backgroundColorForAddedText = UIColors.gray;
    var myColor = backgroundColorForDeletedText;
    backgroundColorForDeletedText = UIColors.red;
    backgroundColorForMovedText = UIColors.pink;
    changeBarColor = UIColors.charcoal;
    deletedBackgroundColorChoice
=ChangeBackgroundColorChoices.CHANGE_BACKGROUND_USES_CHANGE_PREF_COLOR;
    deletedTextColorChoice = ChangeTextColorChoices.CHANGE_USES_CHANGE_PREF_COLOR;
    //ChangebarLocations.LEFT_ALIGN  (Read Only) Change bars are in the left margin.
    //ChangebarLocations.RIGHT_ALIGN  (Read Only) Change bars are in the right margin
    locationForChangeBar = ChangebarLocations.LEFT_ALIGN;
    //ChangeMarkings.OUTLINE  (Read Only)   Outlines changed text.
    //ChangeMarkings.NONE  (Read Only) Does not mark changed text.
    //ChangeMarkings.STRIKETHROUGH  (Read Only) Uses a strikethrough to mark changed
text.
```

```
    //ChangeMarkings.UNDERLINE_SINGLE  (Read Only) Underlines changed text.
    markingForAddedText = ChangeMarkings.OUTLINE;
    markingForDeletedText = ChangeMarkings.STRIKETHROUGH;
    markingForMovedText = ChangeMarkings.UNDERLINE_SINGLE;
    movedBackgroundColorChoice =
ChangeBackgroundColorChoices.CHANGE_BACKGROUND_USES_CHANGE_PREF_COLOR;
    movedTextColorChoice = ChangeTextColorChoices.CHANGE_USES_CHANGE_PREF_COLOR;
    showAddedText = true;
    shhowDeletedText = true;
    showMovedText = true;
    spellCheckDeletedtext = true;
    showChangeBar = true;
    textColorForAddedText = UIColors.blue;
    textColorForDeletedText = UIColors.yellow;
    textColorForMovedText = UIColors.green;
}
```