ADOBE® INDESIGN® CS3

SCRIPTING GUIDE: JAVASCRIPT

Adobe

1 Introduction

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 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 you already read *Adobe InDesign CS3 Scripting Tutorial* and know how to create, install, and run scripts.

2 Scripting Features

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 you already read *Adobe InDesign CS3 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 5. 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 CS3 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 the application in which they are being run (i.e., the current version). The mechanics of targeting are language specific.

- **Compilation —** This involves mapping the names in the script to the underlying script ids, which are what the application understands. The mechanics of compilation are language specific.
- **Interpretation —** This involves matching the ids to the appropriate request handler within the application. InDesign CS3 correctly interprets a script written for an earlier version of the scripting object model. To do this, run the script from a folder in the Scripts panel folder named `Version 4.0 Scripts` (for InDesign CS2 scripts) or `Version 3.0 Scripts` (for InDesign CS_J scripts), or explicitly set the application's script preferences to the old object model within the script (as shown below).

Targeting

Targeting for JavaScripts is implicit when the script is launched from the Scripts panel. If the script is launched externally (from the ESTK), use the `target` directive:

```
//target CS3
#target "InDesign-5.0"
//target the latest version of InDesign
#target "InDesign"
```

Compilation

JavaScripts are not pre-compiled. For compilation, the application 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.

```
//Set to 4.0 scripting object model
app.scriptPreferences.version = 4.0;
```

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 Start-up" on page 9.

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 CS3\\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

To return a value from a script executed by `doScript`, you can 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 DoScriptReturnValue):

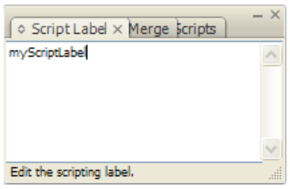
```
var myJavaScript = "app.scriptArgs.setValue(\"ScriptArgumentA\", \"This is the 
first script argument value.\");\r";
myJavaScript += "app.scriptArgs.setValue(\"ScriptArgumentB\", \"This is the 
second script argument value.\")";
var myScriptArgumentA = app.scriptArgs.getValue("ScriptArgumentA");
var myScriptArgumentB = app.scriptArgs.getValue("ScriptArgumentB");
alert("ScriptArgumentA: " + myScriptArgumentA + "\rScriptArgumentB: " + 
myScriptArgumentB);
if(File.fs == "Windows"){
   var myVBScript = "Set myInDesign = 
CreateObject(\"InDesign.Application.CS3\")\r";
   myVBScript += "myInDesign.ScriptArgs.SetValue \"ScriptArgumentA\", \"This 
is the first script argument value.\"\r";
   myVBScript += "myInDesign.ScriptArgs.SetValue \"ScriptArgumentB\", \"This 
is the second script argument value.\"";
```

```
    app.doScript(myVBScript, ScriptLanguage.visualBasic);
}
else{
    var myAppleScript = "tell application \"Adobe InDesign CS3\"\r";
    myAppleScript += "make script arg with properties{name:\"ScriptArgumentA\",
value:\"This is the first script argument value.\"}\r";
    myAppleScript += "make script arg with properties{name:\"ScriptArgumentB\",
value:\"This is the second script argument value.\"}\r";
    myAppleScript += "end tell\r";
    app.doScript(myAppleScript, ScriptLanguage.applescriptLanguage);
}
var myScriptArgumentA = app.scriptArgs.getValue("ScriptArgumentA");
var myScriptArgumentB = app.scriptArgs.getValue("ScriptArgumentB");
alert("ScriptArgumentA: " + myScriptArgumentA + "\rScriptArgumentB: " +
myScriptArgumentB);
```

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 > Automation > 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 myPageWidth = myDocument.documentPreferences.pageWidth;
var myPageHeight = myDocument.documentPreferences.pageHeight;
//Create 10 random page items.
for(var myCounter = 0; myCounter < 10; myCounter++){
    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 myPageItems = myPage.pageItems.item("myScriptLabel");
if(myPageItems.getElements().length != 0){
    alert("Found " + myPageItems.getElements().length + " 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 Start-up

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 CS3 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 9.

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

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.
- Print a document.
- Export a document as Adobe PDF.
- Export pages of a document as EPS.

We assume you already read *Adobe InDesign CS3 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 (i.e., 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 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 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.

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.paragraphSt
yles.item("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.paragraphS
tyles.item("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);
```

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. This following script shows how to set print preferences using scripting (for the complete script, see PrintPreferences):

```
//PrintPreferences.jsx
//An InDesign CS3 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 = "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;
    colorOutput = ColorOutputModes.separations;
    //Note the lowercase "i" in "Builtin"
    trapping = Trapping.applicationBuiltin;
    screening = "175 lpi/2400 dpi";
    flip = Flip.none;
    //If trapping is on, attempting to set the following
    //properties will generate an error.
    if(trapping == Trapping.off){
        printBlack = true;
```

```
   printCyan = true;
   printMagenta = true;
   printYellow = true;
}
//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.
   if(trapping == Trapping.off){
   printBlankPages = false;
   printGuidesGrids = false;
   printNonprinting = false;
}
//----------------------------------------------------------------------------
//Properties corresponding to the controls in the
//Setup panel of the Print dialog box.
//----------------------------------------------------------------------------
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;
//If trapping is on, attempting to set the
//following properties will produce an error.
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;
}
//----------------------------------------------------------------------------
//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.
//-----------------------------------------------------------------------
sendImageData = ImageDataTypes.allImageData;
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;
   interactiveElements = false;
   //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;
   contentToEmbed = PDFContentToEmbed.embedAll;
   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 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 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();
        }
    }
```

4 Text and Type

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 you already read *Adobe InDesign CS3 Scripting Tutorial* and know how to create, install, and run a script. We also assume 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 MakeTextFrame tutorial):

```
var myDocument = app.documents.add();
//Set the measurement units to points.
myDocument.viewPreferences.horizontalMeasurementUnits =
MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
//Create a text frame on page 1.
var myTextFrame = myDocument.pages.item(0).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."
```

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 very useful function you can add to your own scripts, and we use it in many other examples in this chapter. (For the complete script, see `MakeTextFrameWithinMargins`.)

```
//Creates a text frame in an example document
//and sizes the text frame to match the page margins.
var myDocument = app.documents.add();
//Create a text frame on page 1.
var myTextFrame = myDocument.pages.item(0).textFrames.add();
//Set the bounds of the text frame.
myTextFrame.geometricBounds = myGetBounds(myDocument,
myDocument.pages.item(0));
//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:myDocument,
myDocument.pages.item(0), contents:"This is some example text."});
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):

```
//Creates a text frame in an example document,
//enters text in the text frame, and then adds
//text at the end of the text frame.
var myDocument = app.documents.add();
//Set the measurement units to points.
myDocument.viewPreferences.horizontalMeasurementUnits =
MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
//Create a text frame on page 1.
var myTextFrame =
myDocument.pages.item(0).textFrames.add({geometricBounds:[72, 72, 288, 288],
contents:"This is some example text."});
//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.)
myTextFrame.parentStory.insertionPoints.item(-1).contents = "\rThis is a new
paragraph of example text.";
```

Stories and Text Frames

All text in an InDesign layout is part 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 script above, we added the text at the end of the parent story rather than 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 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 very 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).

```
//Creates a text frame in an example document.
var myDocument = app.documents.add();
//Set the measurement units to points.
myDocument.viewPreferences.horizontalMeasurementUnits =
MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
//Create a text frame on page 1.
var myTextFrame = myDocument.pages.item(0).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 54.

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):

```
//Creates a text frame in an example document,
//enters text in the text frame, and then replaces
//a word in the frame with a different phrase.
var myDocument = app.documents.add();
//Set the measurement units to points.
myDocument.viewPreferences.horizontalMeasurementUnits =
MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
//Create a text frame on page 1.
var myTextFrame =
myDocument.pages.item(0).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):

```
//Creates a text frame in an example document,
//enters text in the text frame, and then replaces
//the text in the second paragraph.
var myDocument = app.documents.add();
//Set the measurement units to points.
myDocument.viewPreferences.horizontalMeasurementUnits =
MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
//Create a text frame on page 1.
var myTextFrame =
myDocument.pages.item(0).textFrames.add({geometricBounds:[72, 72, 288, 288],
contents:"Paragraph 1.\rParagraph 2.\rParagraph 3.\r"});
//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 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 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 the examples in “Creating a Text Frame” on page 39” or the SpecialCharacters tutorial script.)

```
//Shows how to enter special characters in text.
var myDocument = app.documents.add();
//Create a text frame on page 1.
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: ‚Ñ¢\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):

```
//Places a text file on page 1 of a new document.
var myDocument = app.documents.add();
//Set the measurement units to points.
myDocument.viewPreferences.horizontalMeasurementUnits =
MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
//Get the top and left margins to use as a place point.
var myX = myDocument.pages.item(0).marginPreferences.left;
var myY = myDocument.pages.item(0).marginPreferences.top;
//Autoflow a text file on page 1 .
//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 = myDocument.pages.item(0).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 39," or see the PlaceTextFileInFrame tutorial script.)

```
//Places a text file in a text frame.
var myDocument = app.documents.add();
//Set the measurement units to points.
myDocument.viewPreferences.horizontalMeasurementUnits =
MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
//Create a text frame.
var myTextFrame =
myDocument.pages.item(0).textFrames.add({geometricBounds:myGetBounds(myDocument,
myDocument.pages.item(0))});
//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 39," or see the InsertTextFile tutorial script.)

```
//Places a text file in text inside a text frame (without replacing the text).
var myDocument = app.documents.add();
//Create a text frame.
var myTextFrame =
myDocument.pages.item(0).textFrames.add({geometricBounds:myGetBounds(myDocument,
myDocument.pages.item(0)), contents:"Inserted text file follows:\r"});
//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.

```
//Sets the text import filter preferences.
with(app.textImportPreferences){
    //Options for characterSet:
    //TextImportCharacterSet.ansi
    //TextImportCharacterSet.chineseBig5
    //TextImportCharacterSet.gb18030
    //TextImportCharacterSet.gb2312
    //TextImportCharacterSet.ksc5601
    //TextImportCharacterSet.macintoshCE
    //TextImportCharacterSet.macintoshCyrillic
    //TextImportCharacterSet.macintoshGreek
    //TextImportCharacterSet.macintoshTurkish
    //TextImportCharacterSet.recommendShiftJIS83pv
    //TextImportCharacterSet.shiftJIS90ms
    //TextImportCharacterSet.shiftJIS90pv
    //TextImportCharacterSet.unicode
    //TextImportCharacterSet.windowsBaltic
    //TextImportCharacterSet.windowsCE
    //TextImportCharacterSet.windowsCyrillic
    //TextImportCharacterSet.windowsEE
    //TextImportCharacterSet.windowsGreek
    //TextImportCharacterSet.windowsTurkish
    characterSet = TextImportCharacterSet.unicode;
    convertSpacesIntoTabs = true;
    spacesIntoTabsCount = 3;
    //The dictionary property can take any of the following
    //language names (as strings):
    //Bulgarian
    //Catalan
    //Croatian
    //Czech
    //Danish
    //Dutch
    //English:Canadian
    //English:UK
    //English:USA
    //English:USA Legal
    //English:USA Medical
    //Estonian
    //Finnish
```

```
        //French
        //French:Canadian
        //German:Reformed
        //German:Swiss
        //German:Traditional
        //Greek
        //Hungarian
        //Italian
        //Latvian
        //Lithuanian
        //Neutral
        //Norwegian:Bokmal
        //Norwegian:Nynorsk
        //Polish
        //Portuguese
        //Portuguese:Brazilian
        //Romanian
        //Russian
        //Slovak
        //Slovenian
        //Spanish:Castilian
        //Swedish
        //Turkish
        dictionary = "English:USA";
        //platform options:
        //ImportPlatform.macintosh
        //ImportPlatform.pc
        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):

```
    //Sets the tagged text import filter preferences.
    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):

```
//Sets the Word/RTF import filter preferences.
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):

```
//Sets the Excel import filter preferences.
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;
   //You'll have to enter a valid worksheet name.
   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 you must use text or story objects to export in 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 39," or see the ExportTextFile tutorial script.)

```
//Creates a story in an example document and then exports the text to a text file.
var myDocument = app.documents.add();
//Create a text frame on page 1.
var myTextFrame = myDocument.pages.item(0).textFrames.add();
//Set the bounds of the text frame.
myTextFrame.geometricBounds = myGetBounds(myDocument,
myDocument.pages.item(0));
//Fill the text frame with placeholder text.
myTextFrame.contents = TextFrameContents.placeholderText;
//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 39," or see the ExportTextRange tutorial script.)

```
//Creates a story in an example document and then exports
//some of the text to a text file.
var myDocument = app.documents.add();
//Create a text frame on page 1.
var myTextFrame = myDocument.pages.item(0).textFrames.add();
//Set the bounds of the text frame.
myTextFrame.geometricBounds = myGetBounds(myDocument,
myDocument.pages.item(0));
//Fill the text frame with placeholder text.
myTextFrame.contents = TextFrameContents.placeholderText;
var myStory = myTextFrame.parentStory;
var myStartCharacter = myStory.paragraphs.item(0).characters.item(0);
var myEndCharacter = myStory.paragraphs.item(1).characters.item(-1);
myText = myTextFrame.parentStory.texts.itemByRange(myStartCharacter,
myEndCharacter);
//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):

```
//Sets the text export filter preferences.
with(app.textExportPreferences){
    //Options for characterSet:
    //TextExportCharacterSet.unicode
    //TextExportCharacterSet.defaultPlatform
    characterSet = TextExportCharacterSet.unicode;
    //platform options:
    //ImportPlatform.macintosh
    //ImportPlatform.pc
    platform = ImportPlatform.macintosh;
}
```

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

```
//Sets the tagged text export filter preferences.
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 39," or see the ExportAllText tutorial script.) For any format other than text only, the latter method can become quite complex.

```
//Exports all of the text in the active document as a single
//text file. To do this, the script will create a new document,
//combine the stories in the new document using export/import,
//and then export the text from the new document.
if(app.documents.length != 0){
   if(app.documents.item(0).stories.length != 0){
      myExportAllText(app.documents.item(0).name);
   }
}
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);
    }
```

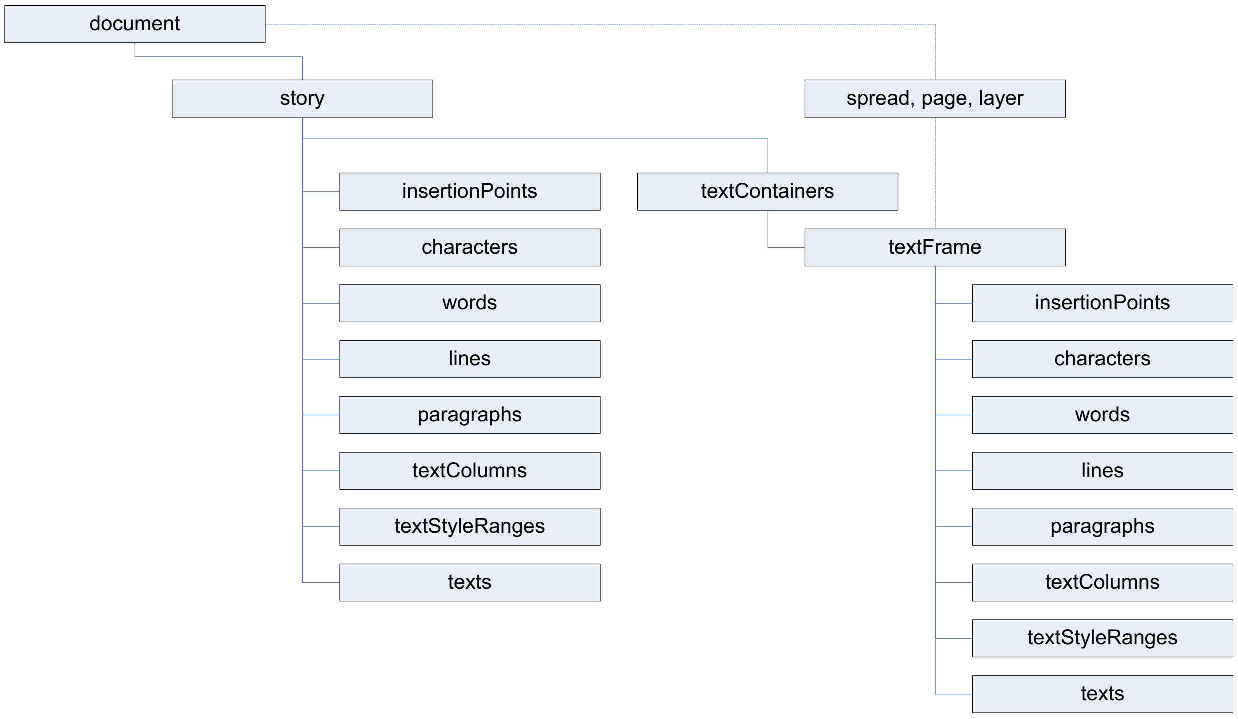
Do not assume you are limited to exporting text using existing export filters. Since JavaScript can write text files to disk, you can have your script traverse the text in a document and export it in any order you like, using whatever text mark-up scheme you prefer. Here is a very simple example that 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 39," or see the ExportHTML tutorial script.)

```
//Exports the text in a document as HTML by traversing the stories in
//the document and writing the text to disk as HTML markup.
//
main();
function main(){
    myCreateExampleDocument();
    myExportHTML();
}
function myCreateExampleDocument(){
    var myParagraphStyle, myCharacterStyle, myColor, myName;
    //Create an example document.
    var myDocument = app.documents.add();
    //Create example paragraph styles
    myHeading1Style = myMakeParagraphStyle(myDocument, "heading1");
    myHeading1Style.pointSize = 24;
    myHeading2Style = myMakeParagraphStyle(myDocument, "heading2");
    myHeading2Style.pointSize = 16;
    myHeading3Style = myMakeParagraphStyle(myDocument, "heading3");
    myHeading3Style.pointSize = 12;
    myBodyTextStyle = myMakeParagraphStyle(myDocument, "body_text");
    myBodyTextStyle.pointSize = 10;
    //Create a text frame on page 1.
    var myTextFrame = myDocument.pages.item(0).textFrames.add();
    //Set the bounds of the text frame.
    myTextFrame.geometricBounds = myGetBounds(myDocument, myDocument.pages.item(0));
    //Fill the text frame with placeholder text.
    var myString = "Heading 1\r";
    myString += "This is a normal paragraph.\r";
    myString += "Heading 2\r";
    myString += "This is another normal paragraph.\r";
    myString += "Heading 3\r";
    myString += "This is yet another normal paragraph.\r";
    myString += "Column 1\tColumn 2\tColumn 3\r1a\t1b\t1c\r2a\t2b\t2c\r3a\t3b\t3c\r";
    myTextFrame.contents = myString;
    myTextFrame.parentStory.paragraphs.item(0).applyStyle(myHeading1Style);
    myTextFrame.parentStory.paragraphs.item(1).applyStyle(myBodyTextStyle);
    myTextFrame.parentStory.paragraphs.item(2).applyStyle(myHeading2Style);
    myTextFrame.parentStory.paragraphs.item(3).applyStyle(myBodyTextStyle);
    myTextFrame.parentStory.paragraphs.item(4).applyStyle(myHeading3Style);
    myTextFrame.parentStory.paragraphs.item(5).applyStyle(myBodyTextStyle);
    var myStartCharacter =
myTextFrame.parentStory.paragraphs.item(6).characters.item(0);
    var myEndCharacter = myTextFrame.parentStory.characters.item(-2);
    var myText = myTextFrame.parentStory.texts.itemByRange(myStartCharacter,
myEndCharacter)
    myText.convertToTable();
    var myTable = myTextFrame.parentStory.tables.item(0);
    var myRow = myTable.rows.item(0);
    myRow.rowType = RowTypes.headerRow;
}
function myMakeParagraphStyle(myDocument, myStyleName){
    var myStyle;
    try{
        myStyle = myDocument.paragraphStyles.item(myStyleName);
        var myName = myStyle.name;
    }
    catch (myError){
        myStyle = myDocument.paragraphStyles.add({name:myStyleName});
    }
    return myStyle;
}
function myExportHTML(){
    var myStory, myParagraph, myString, myTag, myStartTag, 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>");
```

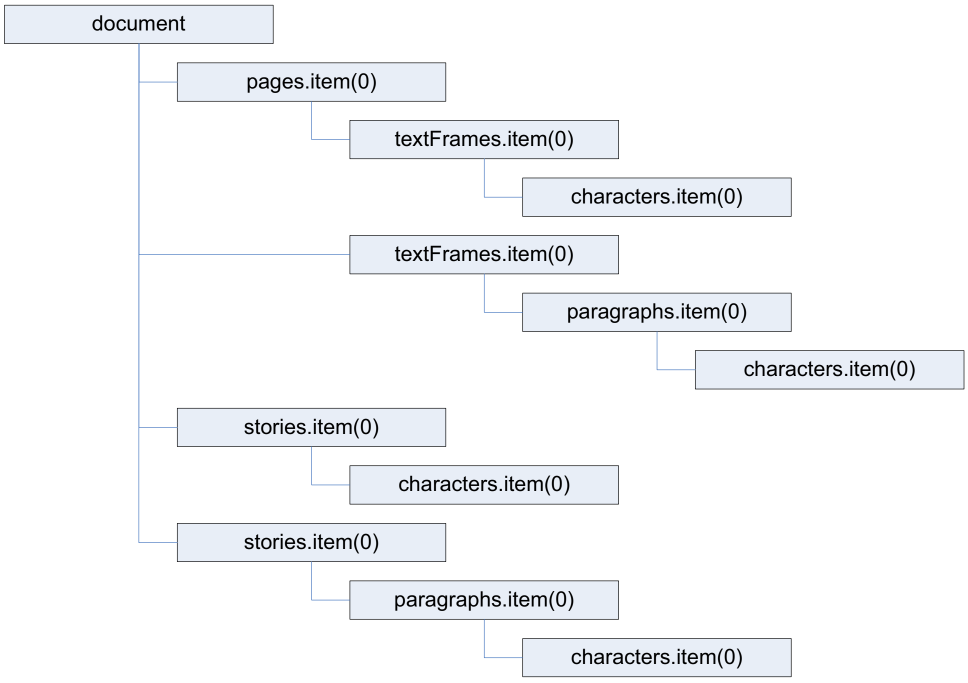
```
                     for(var myRowCounter = 0; myRowCounter <
                     myTable.rows.length; myRowCounter ++){
                        myTextFile.writeln("<tr>");
                        for(var myColumnCounter = 0; myColumnCounter <
                        myTable.columns.length; myColumnCounter++){
                           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();
         }
      }
   }
}
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 find out 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 you can use in your own scripts (for the complete script, see TextSelection).

```
//Shows how to determine whether the current selection is a text selection.
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.");
    }
}
function myProcessText(myTextObject){
    //Do something with the text object.
}
```

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 (which is identical to the `move` method in every way but its name). The following script fragment shows how it works (for the complete script, see MoveText):

```
//Given a series of text frames A, B, C, and D containing the text
//"WordA," "WordB", "WordC" and "WordD"...
//Move WordC between the words in TextFrameC.
myTextFrameD.parentStory.paragraphs.item(-1).words.item(0).move(LocationOptio
ns.before, myTextFrameC.parentStory.paragraphs.item(0).words.item(1))
//Move WordB after the word in TextFrameB.
myTextFrameD.parentStory.paragraphs.item(-2).words.item(0).move(LocationOptio
ns.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(LocationOptio
ns.before, myTextFrameA.parentStory.paragraphs.item(0).words.item(0))
//Note that moving text removes it from its original location. To copy the
//text from one place to another, use the duplicate method, as shown below:
//myTextFrameD.parentStory.paragraphs.item(-3).words.item(0).duplicate(Locati
onOptions.before, myTextFrameA.parentStory.paragraphs.item(0).words.item(0))
```

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 39," or see the MoveTextBetweenDocuments tutorial script.)

```
//MoveTextBetweenDocuments.jsx
//An InDesign CS JavaScript
//
//Moves text from one document to another.
var mySourceDocument = app.documents.add();
//Create a text frame.
var myTextFrame =
mySourceDocument.pages.item(0).textFrames.add({geometricBounds:myGetBounds(my
SourceDocument, mySourceDocument.pages.item(0)), contents:"This is the source
text.\rThis text is not the source text."});
myTextFrame.parentStory.paragraphs.item(0).pointSize = 24;
//Create a new document to move the text to.
var myTargetDocument = app.documents.add();
//Create a text frame in the target document.
var myTargetTextFrame =
myTargetDocument.pages.item(0).textFrames.add({geometricBounds:myGetBounds(my
TargetDocument, myTargetDocument.pages.item(0)), contents:"This is the target
text. Insert the source text after this paragraph.\r"});
//Move the text from the first document to the second. This deletes
//the text from the first document.
myTextFrame.parentStory.paragraphs.item(0).move(LocationOptions.atBeginning,
myTargetTextFrame.insertionPoints.item(0));
//To duplicate (rather than move) the text, use the following:
//myTextFrame.parentStory.paragraphs.item(0).duplicate(LocationOptions.atBegi
nning, 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 39," 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});
//Create another example document.
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.add();
//Set the measurement units to points.
myDocument.viewPreferences.horizontalMeasurementUnits =
MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
myDocument.viewPreferences.rulerOrigin = RulerOrigin.pageOrigin;
//Create a text frame on page 1.
var myTextFrameA =
myDocument.pages.item(0).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 page 1.
var myTextFrameB =
myDocument.pages.item(0).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 page 1.
var myTextFrameC =
myDocument.pages.item(0).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 39," or see the TextIterationWrong tutorial script.)

```
//Shows how *not* to iterate through text.
//
myCreateExampleDocument();
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;
   }
}
function myCreateExampleDocument(){
   //Create an example document.
   var myDocument = app.documents.add();
   //Create a text frame on page 1.
   var myTextFrame = myDocument.pages.item(0).textFrames.add();
   //Set the bounds of the text frame.
   myTextFrame.geometricBounds = myGetBounds(myDocument,
   myDocument.pages.item(0));
   var myString = "Paragraph 1.\rDelete this paragraph.\r";
   myString += "Paragraph 2.\rParagraph 3.\rParagraph 4.\r";
   myString += "Paragraph 5.\rDelete this paragraph.\rParagraph 6.\r";
   myTextFrame.contents = myString;
}
```

In the above 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 39," or see the TextIterationRight tutorial script.)

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

```
//Given a document containing text frames "my TextFrameA",
//"my TextFrameB", and "myTextFrameC"...
//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):

```
//Link TextFrameA to TextFrameB using the nextTextFrame property.
myTextFrameA.nextTextFrame = myTextFrameB;
//Fill the two frames with placeholder text.
myTextFrameA.contents = TextFrameContents.placeholderText;
//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):

```
//Given a text frame "myTextFrame":
var myNewFrame = myTextFrame.duplicate();
if(myTextFrame.contents != ""){
    myTextFrame.texts.item(0).remove();
}
myTextFrame.remove();
```

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):

```
//Given a story "myStory":
function mySplitStory(myStory){
    var myTextFrame;
    //Duplicate each text frame in the story.
    for(var myCounter = myStory.textFrames.length-1; myCounter >= 0; myCounter --){
        myTextFrame = myStory.textFrames.item(myCounter);
        myTextFrame.duplicate();
    }
}
function myRemoveFrames(myStory){
    //Remove each text frame in the story. Iterate backwards to
    //avoid invalid references.
    for(var myCounter = myStory.textFrames.length-1; myCounter >= 0; myCounter --){
        myStory.textFrames.item(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):

```
//Given a document with a text frame on its first page...
var myDocument = app.documents.item(0);
var myPage = myDocument.pages.item(0);
var myTextFrame = myPage.textFrames.item(0);
var myInlineFrame =
myTextFrame.paragraphs.item(0).insertionPoints.item(0).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.
myInlineFrame.geometricBounds = [myBounds[0], myBounds[1], myBounds[0]+24,
myBounds[1]+72];
myInlineFrame.contents = "This is an inline frame.";
```

```
var myAnchoredFrame =
myTextFrame.paragraphs.item(1).insertionPoints.item(0).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.
myAnchoredFrame.geometricBounds = [myBounds[0], myBounds[1], myBounds[0]+24,
myBounds[1]+72];
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;
}
```

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.)

```
//Sets the text defaults of a new document, which set the default formatting
//for all new text frames. Existing text frames are unaffected.
var myDocument = app.documents.add();
//Set the measurement units to points.
myDocument.viewPreferences.horizontalMeasurementUnits =
MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = MeasurementUnits.points;
//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;
        //More properties in the tutorial script file.
    }
```

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.)

```
//Shows the difference between the fonts collection of the application
//and the fonts collection of a document.
var myApplicationFonts = app.fonts;
var myDocument = app.documents.add();
var myDocumentFonts = myDocument.fonts;
var myPage = myDocument.pages.item(0);
var myTextFrame =
myPage.textFrames.add({geometricBounds:myGetBounds(myDocument, myPage)});
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";
}
```

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.add();
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;
//More properties in the tutorial script.
```

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):

```
//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):

```
//Create a text frame on page 1.
var myTextFrame = myDocument.pages.item(0).textFrames.add();
//Set the bounds of the text frame.
myTextFrame.geometricBounds = myGetBounds(myDocument,
myDocument.pages.item(0));
//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).applyStyle(myParagraphStyle, true);
var myStartCharacter = myTextFrame.parentStory.characters.item(13);
var myEndCharacter = myTextFrame.parentStory.characters.item(54);
myTextFrame.parentStory.texts.itemByRange(myStartCharacter,
myEndCharacter).applyStyle(myCharacterStyle, true);
```

Why use the `applyStyle` method instead of setting the `appliedParagraphStyle` or `appliedCharacterStyle` property of the text object? The `applyStyle` 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):

```
//Given a character style "myCharacterStyle" and a
//paragraph style "myParagraphStyle"...
var myNestedStyle =
myParagraphStyle.nestedStyles.add({appliedCharacterStyle:myCharacterStyle,
delimiter:".", inclusive:true, repetition:1});
```

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):

```
//Remove the paragraph style myParagraphStyleA and replace with
myParagraphStyleB.
myParagraphStyleA.remove(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):

```
//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(myFilePath),
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.)

```
//Clear the find/change 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 = app.documents.item(0).findText();
alert("Found " + myFoundItems.length + " instances of the search string.");
```

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):

```
//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 string "copy" and change it to "text".
app.findTextPreferences.findWhat = "copy";
app.changeTextPreferences.changeTo = "text";
app.documents.item(0).changeText();
//Clear the find/change 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):

```
//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;
app.documents.item(0).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):

```
//Clear the find/change 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;
app.documents.item(0).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):

```
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 FindChangeGlyphs):

```
//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.item("Times New Roman
Regular");
//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.item("ITC Zapf Dingbats
Medium");
app.changeGlyphPreferences.glyphID = 85;
myDocument.changeGlyph();
//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.)

```
//Given a table "myTable" that contains at least four rows and four columns...
//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.it
em(-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.)

```
//Given a table "myTable" that contains at least four rows and four columns...
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);
```

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

```
//Given a table "myTable with at least three rows...
//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):

```
//Given a table "myTable" containing at least four rows and a document
//"myDocument" containing the colors "DGC1_446a" and "DGC1_446b"...
//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" containing at least four rows and a document
//"myDocument" containing the colors "DGC1_446a" and "DGC1_446b"...
//Convert the first row to a header row.
myTable.rows.item(0).rowType = RowTypes.headerRow;
//Applly 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;
        }
    }
}
```

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 rectangle "myRectangle"...
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 60).

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 = [[]];
```

Footnotes

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

```
//Given a text frame "myTextFrame"...
//Add four footnotes at random locations in the story.
for(var 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.";
}
```

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;
}
```

5 User Interfaces

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.

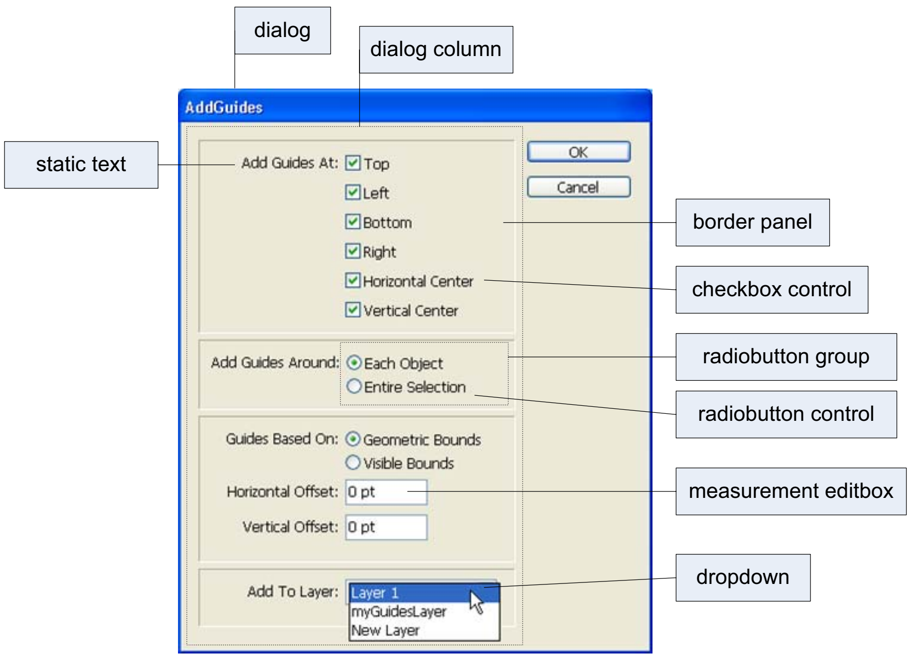
This chapter shows how to work with InDesign dialog scripting. The sample scripts in this chapter are presented in order of complexity, starting with very simple scripts and building toward more complex operations.

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 Creative Suite® 3 JavaScript Tools Guide*.

We assume you already read *Adobe InDesign CS3 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.



Add Guides At: ☑ Top ☑ Left ☑ Bottom ☑ Right ☑ Horizontal Center ☑ Vertical Center

Add Guides Around: ● Each Object ○ Entire Selection

Guides Based On: ● Geometric Bounds ○ Visible Bounds

Horizontal Offset: 0 pt

Vertical Offset: 0 pt

Add To Layer: Layer 1 / myGuidesLayer / New Layer

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”

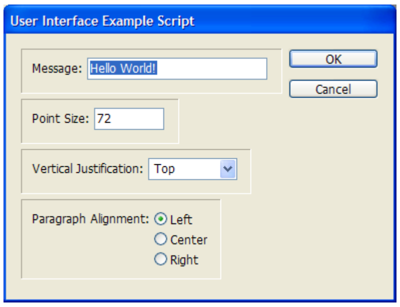
In this example, we add a simple user interface to the Hello World tutorial script presented in *Adobe InDesign CS3 Scripting Tutorial*. The options in the dialog box provide a way for you to specify the sample text and change the point size of the text:

```
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 = realEditboxes.add({editValue:72});
   }
}
//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();
   //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;
        }
    }
    else{
        //User clicked Cancel, so remove the dialog box from memory.
        myDialog.destroy();
    }
    function myGetBounds(myDocument, myPage){
        with(myDocument.documentPreferences){
            var myPageHeight = pageHeight;
            var myPageWidth = pageWidth;
        }
        with(myPage.marginPreferences){
            var myTop = top;
            var myLeft = left;
            var myRight = right;
            var myBottom = bottom;
        }
        myRight = myPageWidth - myRight;
        myBottom = myPageHeight- myBottom;
        return [myTop, myLeft, myBottom, myRight];
    }
```

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:


User Interface Example Script
Message: Hello World!
OK
Cancel
Point Size: 72
Vertical Justification: Top
Paragraph Alignment: Left
Center
Right

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 = realEditboxes.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();
        //Now create the document and apply the properties to the text.
        var myDocument = app.documents.add();
        with(myDocument){
            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;
                }
            }
        }
}
else{
    myDialog.destroy()
}
//Utility function for getting the bounds of the "live area" of a page.
function myGetBounds(myDocument, myPage){
    with(myDocument.documentPreferences){
        var myPageHeight = pageHeight;
        var myPageWidth = pageWidth;
    }
    with(myPage.marginPreferences){
        var myTop = top;
        var myLeft = left;
        var myRight = right;
        var myBottom = bottom;
    }
    myRight = myPageWidth - myRight;
    myBottom = myPageHeight- myBottom;
    return [myTop, myLeft, myBottom, myRight];
}
```

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):

```
#targetengine "session"
myCreateProgressPanel(100, 400);
myProgressPanel.show();
for(var myCounter = 0; myCounter < 101; myCounter ++){
   //Scale the value change to the maximum value of the progress bar.
   myProgressPanel.myProgressBar.value = myCounter/myIncrement;
   app.documents.item(0).pages.add();
   if(myCounter == 100){
      myProgressPanel.myProgressBar.value = 0;
      myProgressPanel.hide();
   }
}
```

Creating a Button-Bar Panel with ScriptUI

If you want to run your scripts by clicking buttons in a floating palette, you can create one using JavaScript and ScriptUI. It does not matter which scripting language the scripts themselves use.

The following tutorial script shows how to create a simple floating panel. The panel can contain a series of buttons, with each button being associated with a script stored on disk. Click the button, and the panel runs the script (the script, in turn can display dialog boxes or other user-interface elements. The button in the panel can contain text or graphics. (For the complete script, see ButtonBar.)

The tutorial script reads an XML file in the following form:

```
<buttons>
   <button>
      <buttonType></buttonType>
      <buttonName></buttonName>
      <buttonFileName></buttonFileName>
      <buttonIconFile></buttonIconFile>
   </button>
   ...
</buttons>
For example:
<buttons>
   <button>
      <buttonType>text</buttonType>
      <buttonName>AddGuides</buttonName>
      <buttonFileName>/c/buttons/AddGuides.jsx</buttonFileName>
      <buttonIconFile></buttonIconFile>
   </button>
   <button>
      <buttonType>icon<buttonType>
      <buttonName>CropMarks<buttonName>
      <buttonFileName>/c/buttons/CropMarks.jsx</buttonFileName>
      <buttonIconFile>/c/buttons/cropmarksicon.jpg<buttonIconFile>
   </button>
   ...
</buttons>
```

The following functions read the XML file and set up the button bar:

```
function myCreateButtonBar(){
   var myButtonName, myButtonFileName, myButtonType, myButtonIconFile,
myButton;
   var myButtons = myReadXMLPreferences();
   if(myButtons != ""){
      myButtonBar = new Window('window', 'Script Buttons', undefined,
{maximizeButton:false, minimizeButton:false});
      with(myButtonBar){
         spacing = 0;
         margins = [0,0,0,0];
         with(add('group')){
            spacing = 2;
            orientation = 'row';
            for(var myCounter = 0; myCounter < myButtons.length(); myCounter++){
               myButtonName = myButtons[myCounter].xpath("buttonName");
               myButtonType = myButtons[myCounter].xpath("buttonType");
               myButtonFileName = myButtons[myCounter].xpath("buttonFileName");
               myButtonIconFile = myButtons[myCounter].xpath("buttonIconFile");
               if(myButtonType == "text"){
                  myButton = add('button', undefined, myButtonName);
               }
               else{
                  myButton = add('iconbutton', undefined,File(myButtonIconFile));
               }
               myButton.scriptFile = myButtonFileName;
               myButton.onClick = function(){
                  myButtonFile = File(this.scriptFile)
                  app.doScript(myButtonFile);
```

```
                    }
                }
            }
        }
    }
    return myButtonBar;
}
function myReadXMLPreferences(){
    myXMLFile = File.openDialog("Choose the file containing your button bar defaults");
    var myResult = myXMLFile.open("r", undefined, undefined);
    var myButtons = "";
    if(myResult == true){
        var myXMLDefaults = myXMLFile.read();
        myXMLFile.close();
        var myXMLDefaults = new XML(myXMLDefaults);
        var myButtons = myXMLDefaults.xpath("/buttons/button");
    }
    return myButtons;
}
```

6 Events

InDesign scripting can respond to common application and document events, like 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, as 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 you already read *Adobe InDesign CS3 Scripting Tutorial* and know how to create, install, and run a script.

This chapter covers application and document events. For a discussion of events related to menus, see Chapter 7, "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 is made up 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).

The following table lists events to which `eventListeners` can respond. These events can be triggered by any available means, including menu selections, keyboard shortcuts, or script actions.

User-Interface Event	Event Name	Description	Object Type
Any menu action	`beforeDisplay`	Appears before the menu or submenu is displayed.	Event
	`beforeDisplay`	Appears before the script menu action is displayed or changed.	Event
	`beforeInvoke`	Appears after the menu action is chosen but before the content of the menu action is executed.	Event
	`afterInvoke`	Appears after the menu action is executed.	Event
	`onInvoke`	Executes the menu action or script menu action.	Event
Close	`beforeClose`	Appears after a close-document request is made but before the document is closed.	DocumentEvent
	`afterClose`	Appears after a document is closed.	DocumentEvent
Export	`beforeExport`	Appears after an export request is made but before the document or page item is exported.	ImportExportEvent
	`afterExport`	Appears after a document or page item is exported.	ImportExportEvent
Import	`beforeImport`	Appears before a file is imported but before the incoming file is imported into a document (before place).	ImportExportEvent
	`afterImport`	Appears after a file is imported but before the file is placed on a page.	ImportExportEvent
New	`beforeNew`	Appears after a new-document request is made but before the document is created.	DocumentEvent
	`afterNew`	Appears after a new document is created.	DocumentEvent
Open	`beforeOpen`	Appears after an open-document request is made but before the document is opened.	DocumentEvent
	`afterOpen`	Appears after a document is opened.	DocumentEvent
Print	`beforePrint`	Appears after a print-document request is made but before the document is printed.	DocumentEvent
	`afterPrint`	Appears after a document is printed.	DocumentEvent

User-Interface Event	Event Name	Description	Object Type
Revert	`beforeRevert`	Appears after a document-revert request is made but before the document is reverted to an earlier saved state.	DocumentEvent
	`afterRevert`	Appears after a document is reverted to an earlier saved state.	DocumentEvent
Save	`beforeSave`	Appears after a save-document request is made but before the document is saved.	DocumentEvent
	`afterSave`	Appears after a document is saved.	DocumentEvent
Save A Copy	`beforeSaveACopy`	Appears after a document save-a-copy-as request is made but before the document is saved.	DocumentEvent
	`afterSaveACopy`	Appears after a document is saved.	DocumentEvent
Save As	`beforeSaveAs`	Appears after a document save-as request is made but before the document is saved.	DocumentEvent
	`afterSaveAs`	Appears after a document is saved.	DocumentEvent

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 three 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.
- **Capturing** — The event starts at the top of the scripting object model—the application—then propagates through the model to the target of the event. Any `eventListeners` capable of responding to the event registered to objects above the `target` will process the event.
- **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 .
`Captures`	If true, the `event` may be handled by `eventListeners` registered to scripting objects above the target object of the event during the capturing phase of event propagation. This means an `eventListener` on the application, for example, can respond to a document event before an `eventListener` is triggered.
`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 cancelling 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 the `event` occurred.

Working with eventListeners

When you create an `eventListener`, you specify the event type (as a string) the event handler (as a JavaScript function or file reference), and whether the `eventListener` can be triggered in the capturing phase of the event. The following script fragment shows how to add an `eventListener` for a specific event (for the complete script, see AddEventListener).

```
main();
function main(){
    var myEventListener = app.addEventListener("afterNew",
    myDisplayEventType, false);
}
function myDisplayEventType(myEvent){
    alert("This event is the " + myEvent.eventType + " event.");
}
```

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

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

When an `eventListener` responds 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 CS3 InDesign Scripting Tutorial*). When you add an `eventListener` script to a document, it is not saved with the document or exported to INX.

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

When you run the above script and place a file, InDesign displays alerts showing, in sequence, the name of the document, then the name of the application.

The following sample script creates an `eventListener` for each supported 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, false);
    }
}
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 += "\rCaptures: " +myEvent.captures;
    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.capturingPhase:
                myPhaseName = "Capturing";
                break;
            case EventPhases.done:
                myPhaseName = "Done";
                break;
            case EventPhases.notDispatching:
                myPhaseName = "Not Dispatching";
                break;
        }
        return myPhaseName;
    }
}
```

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

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

An example “afterNew” eventListener

The `afterNew` event provides a convenient place to add information to the document, like 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 sort 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, false);
}
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){
                    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” eventListener

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 eventListener 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, false);
}
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;
    }
}
```

7 Menus

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 you already read *Adobe InDesign CS3 Scripting Tutorial* and know how to create, install, and run a script.

Understanding the Menu Model

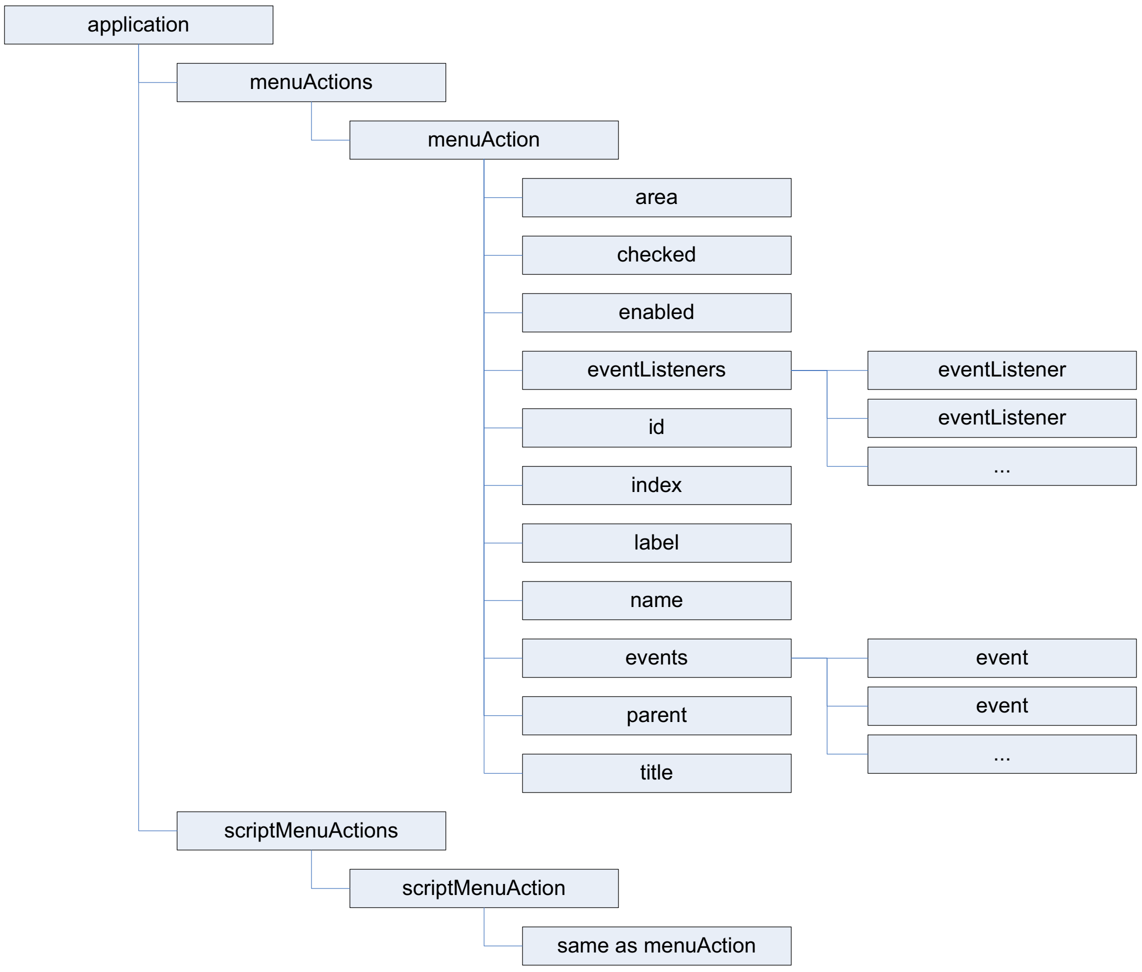
The InDesign menu-scripting model is made up of a series of objects that correspond to the menus you see in the application's user interface, including menus associated with panels as well as those displayed on the main menu bar. A `menu` object contains the following objects:

- `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 `getKeyStrings` 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 a note 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);
}
```

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 6, "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();
            }
        }
    }
  }
}
```

8 XML

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 you already read *Adobe InDesign CS3 Scripting Tutorial* and know how to create and run a script. We also assume 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 9, "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 112.

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).
textFrames.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.item(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;
```

9 XML Rules

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 6, "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 you already read *Adobe InDesign CS3 Scripting Tutorial* and know how to create and run a script. We also assume you have some knowledge of XML and have read Chapter 8, "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 124."

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

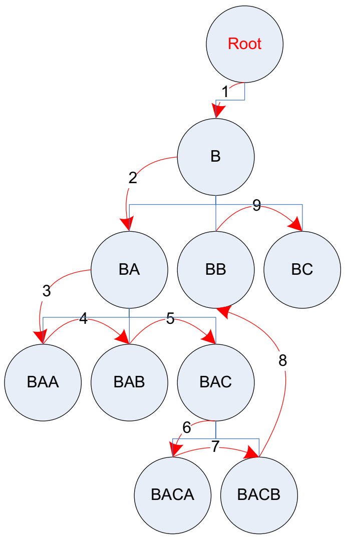
The XML-rules processor iterates through the XML structure of a document by processing each XML element in the order in which it appears in the XML hierarchy of the document. The XML-rules processor uses a forward-only traversal of the XML structure, and it visits each XML element in the structure twice (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.

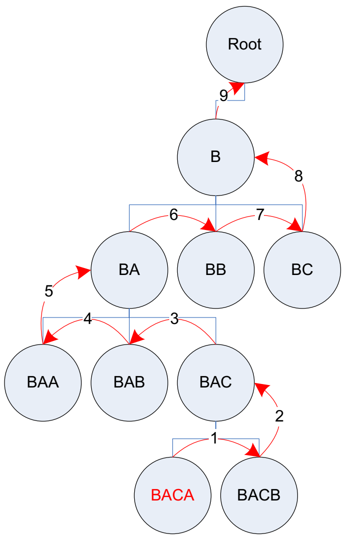
After an XML rule is applied to an XML element, the XML-rules processor continues searching for rules to apply to the descendents of that XML element. An XML rule can alter this behavior by using the `__skipChildren` or `__processChildren` function, or by changing the operation of other rules.

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:

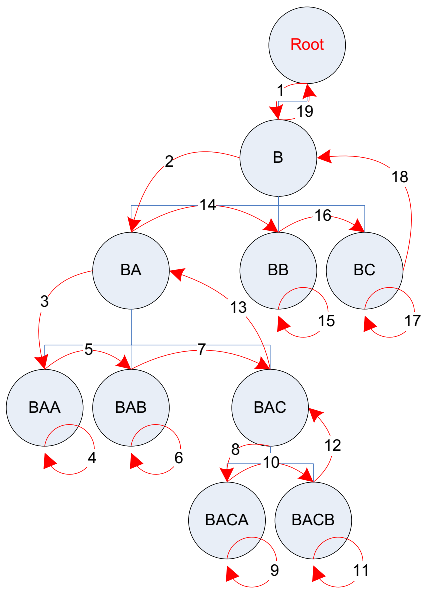


Iteration with __processChildren (assuming a rule that matches every XML element in the structure) is shown in the following figure:



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

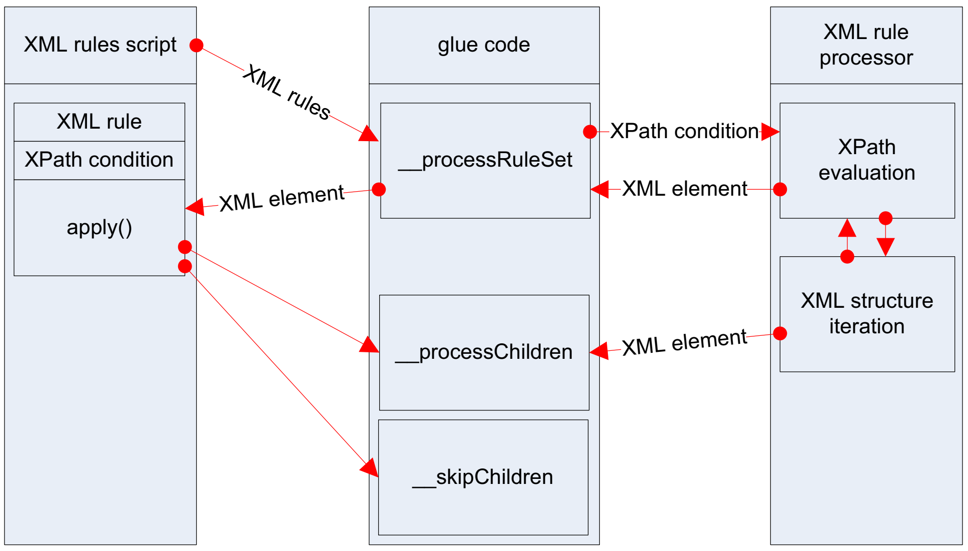
Because XML rules are part of the InDesign scripting model, scripts that use rules do not differ in nature from ordinary scripts, and they benefit from the same error-handling mechanism. When InDesign generates an error, an XML-rules script behaves no differently than any other script. InDesign errors 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 124). 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 after the end of the XML element
                //(this means that the return does not become part of the
                //XML element data, but becomes text data associated with the
                //parent XML element).
                insertTextAsContent("\r", XMLElementPosition.afterElement);
                //To add the return at the end of the element, use:
                //insertTextAsContent("\r", XMLElementPosition.afterElement);
            }
            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 8, "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 8, "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){
         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){
        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){
        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){
        //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){
        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){
        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){
                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){
        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){
        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.fit(FitOptions.frameToContent);
            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's likely that the XML elements we want to arrange in columns use heterogenous 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;
        }
    }
}
```