ADOBE® INDESIGN® CS4

ADOBE INDESIGN CS4 SCRIPTING GUIDE: VBSCRIPT

Adobe

Adobe® InDesign® CS4 Scripting Guide: VBScript

Contents

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 page items (rectangles, ellipses, graphic lines, polygons, text frames, and groups).
- Work with text and type in an InDesign document, including finding and changing text.
- Create dialog boxes and other user-interface items.
- Customize and add menus and create menu actions.
- Respond to user-interface events.
- Work with XML, from creating XML elements and importing XML to adding XML elements to a layout.
- Apply XML rules, a new scripting feature that makes working with XML in InDesign faster and easier.

We assume that you have already read the *Adobe InDesign CS4 Scripting Tutorial* and know how to create, install, and run scripts. If you need to know how to connect with your scripting environment or view the InDesign scripting object model from your script editor, that information can be found in the *Adobe InDesign CS4 Scripting Tutorial*.

How to Use the Scripts in This Document

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

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

About the structure of the scripts

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

For More Information

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

We assume you already read *Adobe InDesign CS4 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 `idUserInteractionLevels.idNeverInteract`, InDesign does not display any alerts or dialogs. Set it to `idUserInteractionLevels.idInteractWithAlerts` to enable alerts but disable dialogs. Set it to `idUserInteractionLevels.idInteractWithAll` 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 `idUserInteractionLevels.idNeverInteract` before opening the document, then restore user interaction (set the property to `idUserInteractionLevels.idInteractWithAll`) before completing script execution.
`Version`	The version of the scripting environment in use. For more information, see "Script versioning" on page 10. 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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
myActiveScript = myInDesign.ActiveScript
MsgBox ("The current script is: " & myActiveScript)
Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
myParentFolder = myFileSystemObject.GetFile(myActiveScript).ParentFolder
MsgBox ("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.

Script versioning

InDesign CS4 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 CS4 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 5.0 Scripts` (for InDesign CS3 scripts) or `Version 2.0 Scripts` (for InDesign CS2 scripts), or explicitly set

the application's script preferences to the old object model within the script (as shown below). Put the previous version scripts in the folder, and run them from the Scripts panel.

Targeting

Targeting for Visual Basic applications and VBScripts must be done using the `CreateObject` method:

```
Rem Target InDesign CS4:
Set myApp = CreateObject("InDesign.Application.CS4")
Rem Target the last version of InDesign that was launched:
Set myApp = CreateObject("InDesign.Application")
```

Compilation

Compilation of Visual Basic applications may be versioned by referencing the CS2 type library. To generate a CS2 version of the type library, use the PublishTerminology method, which is exposed on the Application object. The type library is published into a folder (named with the version of the DOM) that is in the `Scripting Support` folder in your application's preferences folder. For example, `C:\Documents and Settings\`*`user-name`*`\Application Data\Adobe\InDesign\Version 4.0\Scripting Support\3.0` (where *`user-name`* is your user name).

```
Set myApp = CreateObject("InDesign.Application.CS4")
Rem Publish the InDesign CS3 type library (version 5.0 DOM)
myApp.PublishTerminology(5.0)
```

VBScripts are not pre-compiled. The application generates and references the appropriate type library automatically, based on the version of the DOM 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.

The following examples show how to set the version to the CS3 (5.0) version of the scripting object model.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem Set to 5.0 DOM
myInDesign.ScriptPreferences.Version = 5.0
```

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 15.

Sending parameters to DoScript

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

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
myJavaScript = "alert(""First argument: "" + arguments[0] + ""\rSecond argument: "" +
arguments[1]);"
myInDesign.DoScript myJavaScript, idScriptLanguage.idJavascript, Array("Hello from
DoScript", "Your message here.")
myVBScript = "msgbox arguments(1), vbOKOnly, ""First argument: "" & arguments(0)"
myInDesign.DoScript myVBScript, idScriptLanguage.idVisualBasic, Array("Hello from
DoScript", "Your message here.")
```

Returning values from DoScript

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

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrame = myPage.TextFrames.Item(1)
Set myDestinationPage = myDocument.Pages.Add(idLocationOptions.idAfter, myPage)
myPageIndex = myDestinationPage.Name
myID = myTextFrame.Id
myJavaScript = "var myDestinationPage = arguments[1];" & vbCr
myJavaScript = myJavaScript & "myID = arguments[0];" & vbCr
myJavaScript = myJavaScript & "var myX = arguments[2];" & vbCr
myJavaScript = myJavaScript & "var myY = arguments[3];" & vbCr
myJavaScript = myJavaScript & "var myPageItem =
app.documents.item(0).pages.item(0).pageItems.itemByID(myID);" & vbCr
myJavaScript = myJavaScript &
"myPageItem.duplicate(app.documents.item(0).pages.item(myDestinationPage));" & vbCr
Rem Create an array for the parameters we want to pass to the JavaScript.
myArguments = Array(myID, myPageIndex, 0, 0)
Set myDuplicate = myInDesign.DoScript(myJavaScript, idScriptLanguage.idJavascript,
myArguments)
Rem myDuplicate now contains a reference to the duplicated text frame.
Rem Change the text in the duplicated text frame.
myDuplicate.contents = "Duplicated text frame."
```

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

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
myJavaScript = "app.scriptArgs.setValue(""ScriptArgumentA"", ""This is the first
script argument value."");" & vbCr
myJavaScript = myJavaScript & "app.scriptArgs.setValue(""ScriptArgumentB"", ""This is
the second script argument value."");" & vbCr
myInDesign.DoScript myJavaScript, idScriptLanguage.idJavascript
myScriptArgumentA = myInDesign.ScriptArgs.GetValue("ScriptArgumentA")
myScriptArgumentB = myInDesign.ScriptArgs.GetValue("ScriptArgumentB")
MsgBox "ScriptArgumentA: " & myScriptArgumentA & vbCr & "ScriptArgumentB: " &
myScriptArgumentB
myVBScript = "Set myInDesign = CreateObject(""InDesign.Application.CS4"")" & vbCr
myVBScript = myVBScript & "myInDesign.ScriptArgs.SetValue ""ScriptArgumentA"", ""This
is the first script argument value."" & vbCr
myVBScript = myVBScript & "myInDesign.ScriptArgs.SetValue ""ScriptArgumentB"", ""This
is the second script argument value."""
myInDesign.DoScript myVBScript, idScriptLanguage.idVisualBasic
myScriptArgumentA = myInDesign.ScriptArgs.GetValue("ScriptArgumentA")
myScriptArgumentB = myInDesign.ScriptArgs.GetValue("ScriptArgumentB")
MsgBox "ScriptArgumentA: " & myScriptArgumentA & vbCr & "ScriptArgumentB: " &
myScriptArgumentB
```

Controlling Undo with DoScript

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

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

```
Rem Given a script "myVBScript" and an array of parameters "myParameters"...
myInDesign.DoScript myVBScript, idScriptLanguage.idVisualBasic, myParameters,
idUndoModes.idFastEntireScript, "Script Action"
Rem idUndoModes can be:
Rem idUndoModes.idAutoUnto: Add no events to the Undo queue.
Rem idUndoModes.idEntireScript: Put a single event in the Undo queue.
Rem idUndoModes.idFastEntireScript: Put a single event in the Undo queue.
Rem idUndoModes.idScriptRequest: Undo each script action as a separate event.
Rem The last parameter is the text that appears in the Undo menu item.
```

Working with script labels

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

The label of page items can be viewed, entered, or edited using the Script Label panel (choose Window > 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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
Set myPage = myDocument.Pages.Item(1)
myPageWidth = myDocument.DocumentPreferences.PageWidth
myPageHeight = myDocument.DocumentPreferences.PageHeight
Rem Create 10 random page items.
For myCounter = 1 To 10
    myX1 = myGetRandom(0, myPageWidth, False)
    myY1 = myGetRandom(0, myPageHeight, False)
    myX2 = myGetRandom(0, myPageWidth, False)
    myY2 = myGetRandom(0, myPageHeight, False)
    Set myRectangle = myPage.Rectangles.Add
    myRectangle.GeometricBounds = Array(myY1, myX1, myY2, myX2)
    If myGetRandom(0, 1, True) > 0 Then
        myRectangle.Label = "myScriptLabel"
    End If
Next
Set myPageItems = myPage.PageItems.Item("myScriptLabel")
If myPageItems.Count <> 0 Then
    MsgBox ("Found " & CStr(myPageItems.Count) & " page items with the label.")
End If
Rem This function gets a random number in the range myStart to myEnd.
Function myGetRandom(myStart, myEnd, myInteger)
    Rem Here's how to generate a random number from a given range:
    Rem Int((upperbound - lowerbound + 1) * Rnd + lowerbound)
    If myInteger = True Then
        myGetRandom = Int(((myEnd - myStart + 1) * Rnd) + myStart)
    Else
        myGetRandom = ((myEnd - myStart + 1) * Rnd) + myStart
    End If
End Function
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
Set myPage = myDocument.Pages.Item(1)
myDocument.ViewPreferences.HorizontalMeasurementUnits = idMeasurementUnits.idPoints
myDocument.ViewPreferences.VerticalMeasurementUnits = idMeasurementUnits.idPoints
Set myRectangle = myPage.Rectangles.Add
myRectangle.GeometricBounds = Array(72, 72, 144, 144)
Rem Insert a custom label using insertLabel. The first parameter is the
Rem 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."
Rem Extract the text from the label and display it in an message box.
myString = myRectangle.ExtractLabel("CustomLabel")
MsgBox ("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 CS4 Scripting Tutorial*).

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 CS4 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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.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):

```
Rem Creates a new document using the specified document preset.
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem Replace "myDocumentPreset" in the following line with the name
Rem of the document preset you want to use.
Set myDocument = myInDesign.Documents.Add(True,
myInDesign.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):

```
Rem Creates a new document using the specified document preset.
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem Replace "myDocumentPreset" in the following line with the name
Rem of the document preset you want to use.
Set myDocument = myInDesign.Documents.Add(False)
Rem To show the window:
Set 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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Open("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):

```
Rem Opens an existing document in the background, then shows the document.
Rem Youíll have to fill in your own file path.
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Open("c:\myTestDocument.indd", False)
Rem At this point, you can do things with the document without showing
Rem the document window. In some cases, scripts will run faster when
Rem the document window is not visible.
Rem When you want to show the hidden document, create a new window.
Set 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):

```
Rem If the active document has been changed since it was last saved, save it.
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.ActiveDocument.Modified = True Then
    myInDesign.ActiveDocument.Save
End If
```

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):

```
Rem If the active document has not been saved (ever), save it.
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myDocument.Saved = False Then
    Rem If you do not provide a file name,
    Rem InDesign displays the Save dialog box.
    myInDesign.ActiveDocument.Save "c:\myTestDocument.indd"
End If
```

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

```
Rem Save the active document as a template.
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.ActiveDocument.Saved = True Then
    Rem Convert the file name to a string.
    myFileName = myInDesign.ActiveDocument.FullName
    Rem If the file name contains the extension ".indd",
    Rem change it to ".indt".
    If InStr(1, myFileName, ".indd") <> 0 Then
        myFileName = Replace(myFileName, ".indd", ".indt")
    End If
Else
    Rem If the document has not been saved, then give it a
    Rem default file name/file path.
    myFileName = "c:\myTestDocument.indt"
End If
myInDesign.ActiveDocument.Save myFileName, True
```

Closing a document

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

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
myInDesign.ActiveDocument.Close
Rem Note that you could also use:
Rem myInDesign.Documents.Item(1).Close
```

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

```
Rem Use idSaveOptions.idYes to save the document,idSaveOptions.idNo
Rem to close the document without saving, or idSaveOptions.idAsk to
Rem display a prompt. If you use idSaveOptions.idYes, you'll need to
Rem provide a reference to a file to save to in the second parameter Rem(SavingIn).
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem If the file has not been saved, display a prompt.
If myInDesign.ActiveDocument.Saved <> True Then
    myInDesign.ActiveDocument.Close idSaveOptions.idAsk
    Rem Or, to save to a specific file name:
    Rem myFile = "c:\myTestDocument.indd"
    Rem myInDesign.ActiveDocument.Close idSaveOptions.idYes, myFile
Else
    Rem If the file has already been saved, save it.
    myInDesign.ActiveDocument.Close idSaveOptions.idYes
End If
```

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

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
For myCounter = myInDesign.Documents.Length To 1 Step -1
    myInDesign.Documents.Item(myCounter).Close idSaveOptions.idNo
Next
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
With myDocument.DocumentPreferences
    .PageHeight = "800pt"
    .PageWidth = "600pt"
    .PageOrientation = idPageOrientation.idLandscape
    .PagesPerDocument = 16
End With
```

NOTE: The `Application` 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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
Rem The bleed and slug properties belong to the DocumentPreferences object.
With myDocument.DocumentPreferences
    Rem Bleed
    .DocumentBleedBottomOffset = "3p"
    .DocumentBleedTopOffset = "3p"
    .DocumentBleedInsideOrLeftOffset = "3p"
    .DocumentBleedOutsideOrRightOffset = "3p"
    Rem Slug
    .SlugBottomOffset = "18p"
    .SlugTopOffset = "3p"
    .SlugInsideOrLeftOffset = "3p"
    .SlugRightOrOutsideOffset = "3p"
End With
```

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):

```
Rem Create a new document.
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
Rem The bleed properties belong to the documentPreferences object.
With myDocument.DocumentPreferences
    Rem Bleed
    .DocumentBleedTopOffset = "3p"
    .DocumentBleedUniformSize = True
End With
```

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):

```
Rem Create a new document.
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
Rem The slug properties belong to the documentPreferences object.
With myDocument.DocumentPreferences
    Rem Slug:
    .SlugTopOffset = "3p"
    .DocumentSlugUniformSize = True
End With
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
With myInDesign.ActiveDocument.PasteboardPreferences
    Rem Any of InDesign's guides can use the UIColors constants...
    .BleedGuideColor = idUIColors.idCuteTeal
    .SlugGuideColor = idUIColors.idCharcoal
    Rem ...or you can specify an array of RGB values
    Rem(with values from 0 to 255)
    Rem .BleedGuideColor = Array(0, 198, 192)
    Rem .SlugGuideColor = Array(192, 192, 192)
End With
```

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.)

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
With myDocument.Pages.Item(1).MarginPreferences
    .ColumnCount = 3
    Rem columnGutter can be a number or a measurement string.
    .ColumnGutter = "1p"
    .Top = "4p"
    .Bottom = "6p"
    Rem When document.documentPreferences.facingPages = true,
    Rem "left" means inside "right" means outside.
    .Left = "6p"
    .Right = "4p"
End With
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
With myDocument.Pages.Item(1).MarginPreferences
    .ColumnCount = 3
    Rem columnGutter can be a number or a measurement string.
    .ColumnGutter = "1p"
    .Top = "4p"
    .Bottom = "6p"
    Rem When document.documentPreferences.facingPages = true,
    Rem "left" means inside "right" means outside.
    .Left = "6p"
    .Right = "4p"
End With
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
myDocument.ViewPreferences.HorizontalMeasurementUnits = idMeasurementUnits.idPoints
myDocument.ViewPreferences.VerticalMeasurementUnits = idMeasurementUnits.idPoints
With myDocument.MarginPreferences
    .Top = 0
    .Left = 0
    .Bottom = 0
    .Right = 0
    .ColumnCount = 1
    .ColumnGutter = 0
End With
Rem The following assumes that your default document contains a single page.
With myDocument.Pages.Item(1).MarginPreferences
    .Top = 0
    .Left = 0
    .Bottom = 0
    .Right = 0
    .ColumnCount = 1
    .ColumnGutter = 0
End With
Rem The following assumes that your default master spread contains two pages.
With myDocument.MasterSpreads.Item(1).Pages.Item(1).MarginPreferences
    .Top = 0
    .Left = 0
    .Bottom = 0
    .Right = 0
    .ColumnCount = 1
    .ColumnGutter = 0
End With
With myDocument.MasterSpreads.Item(1).Pages.Item(2).MarginPreferences
    .Top = 0
    .Left = 0
    .Bottom = 0
    .Right = 0
    .ColumnCount = 1
    .ColumnGutter = 0
End With
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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
With myInDesign.MarginPreferences
    Rem Save the current application default margin preferences.
    myY1 = .Top
    myX1 = .Left
    myY2 = .Bottom
    myX2 = .Right
    Rem Set the application default margin preferences.
    .Top = 0
    .Left = 0
    .Bottom = 0
    .Right = 0
End With
Rem Create a new example document to demonstrate the change.
Set myDocument = myInDesign.Documents.Add
myDocument.DocumentPreferences.PageHeight = "1p"
myDocument.DocumentPreferences.PageWidth = "6p"
Rem Reset the application default margin preferences to their former state.
With myInDesign.MarginPreferences
    .Top = myY1
    .Left = myX1
    .Bottom = myY2
    .Right = myX2
End With
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
With myDocument.PasteboardPreferences
    Rem You can use either a number or a measurement
    Rem string to set the space above/below.
    .MinimumSpaceAboveAndBelow = "12p"
    Rem You can set the preview background color (which you'll only see
    Rem in Preview mode) to any of the predefined UIColor constants...
    .PreviewBackgroundColor = idUIColors.idGrassGreen
    Rem ...or you can specify an array of RGB values (with values from 0 to 255)
    Rem .PreviewBackgroundColor = Array(192, 192, 192)
End With
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
myPageWidth = myDocument.DocumentPreferences.PageWidth
myPageHeight = myDocument.DocumentPreferences.PageHeight
With myDocument.Pages.Item(1)
    Set myMarginPreferences = .MarginPreferences
    Rem Place guides at the margins of the page.
    With .Guides.Add
        .Orientation = idHorizontalOrVertical.idVertical
        .Location = myMarginPreferences.Left
    End With
    With .Guides.Add
        .Orientation = idHorizontalOrVertical.idVertical
        .Location = (myPageWidth - myMarginPreferences.Right)
    End With
    With .Guides.Add
        .Orientation = idHorizontalOrVertical.idHorizontal
        .Location = myMarginPreferences.Top
    End With
    With .Guides.Add
        .Orientation = idHorizontalOrVertical.idHorizontal
        .Location = (myPageHeight - myMarginPreferences.Bottom)
    End With
    Rem Place a guide at the vertical center of the page.
    With .Guides.Add
        .Orientation = idHorizontalOrVertical.idVertical
        .Location = (myPageWidth / 2)
    End With
    Rem Place a guide at the horizontal center of the page.
    With .Guides.Add
        .Orientation = idHorizontalOrVertical.idHorizontal
        .Location = (myPageHeight / 2)
    End With
End With
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
myPageWidth = myDocument.DocumentPreferences.PageWidth
myPageHeight = myDocument.DocumentPreferences.PageHeight
With myDocument.Pages.Item(1)
    Set myMarginPreferences = .MarginPreferences
    Rem Place guides at the margins of the page.
    With .Guides.Add
        .Orientation = idHorizontalOrVertical.idVertical
        .Location = myMarginPreferences.Left
    End With
    With .Guides.Add
        .Orientation = idHorizontalOrVertical.idVertical
        .Location = (myPageWidth - myMarginPreferences.Right)
    End With
    With .Guides.Add
        .Orientation = idHorizontalOrVertical.idHorizontal
        .Location = myMarginPreferences.Top
    End With
```

```
    With .Guides.Add
        .Orientation = idHorizontalOrVertical.idHorizontal
        .Location = (myPageHeight - myMarginPreferences.Bottom)
    End With
    Rem Place a guide at the vertical center of the page.
    With .Guides.Add
        .Orientation = idHorizontalOrVertical.idVertical
        .Location = (myPageWidth / 2)
    End With
    Rem Place a guide at the horizontal center of the page.
    With .Guides.Add
        .Orientation = idHorizontalOrVertical.idHorizontal
        .Location = (myPageHeight / 2)
    End With
End With
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
Rem Parameters (all optional): row count, column count, row gutter,
Rem column gutter,guide color, fit margins, remove existing, layer.
Rem Note that the createGuides method does not take an RGB array
Rem for the guide color parameter.
myDocument.Spreads.Item(1).CreateGuides 4, 4, "1p", "1p", idUIColors.idGray, 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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
Rem Set the document measurement units to points.
myDocument.ViewPreferences.HorizontalMeasurementUnits = idMeasurementUnits.idPoints
myDocument.ViewPreferences.VerticalMeasurementUnits = idMeasurementUnits.idPoints
Rem Set up grid preferences.
With myDocument.GridPreferences
    .BaselineStart = 56
    .BaselineDivision = 14
    .BaselineGridShown = True
    .HorizontalGridlineDivision = 14
    .HorizontalGridSubdivision = 5
    .VerticalGridlineDivision = 14
    .VerticalGridSubdivision = 5
    .DocumentGridShown = True
End With
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.ActiveDocument
With myDocument.ViewPreferences
    Rem Measurement unit choices are:
    Rem * idMeasurementUnits.idAgates
Rem * idMeasurementUnits.idPicas
    Rem * idMeasurementUnits.idPoints
    Rem * idMeasurementUnits.idInches
    Rem * idMeasurementUnits.idInchesDecimal
    Rem * idMeasurementUnits.idMillimeters
    Rem * idMeasurementUnits.idCentimeters
    Rem * idMeasurementUnits.idCiceros
    Rem * idMeasurementUnits.idCustom
    Rem If you set the the vertical or horizontal measurement units
    Rem to idMeasurementUnits.idCustom, you can also set a custom
    Rem ruler increment (in points) using:
    Rem .HorizontalCustomPoints = 15
    Rem .VerticalCustomPoints = 15
    Rem Set horizontal and vertical measurement units to points.
    .HorizontalMeasurementUnits = idMeasurementUnits.idPoints
    .VerticalMeasurementUnits = idMeasurementUnits.idPoints
End With
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.ActiveDocument
With myDocument.ViewPreferences
    myOldXUnits = .HorizontalMeasurementUnits
    myOldYUnits = .VerticalMeasurementUnits
    .HorizontalMeasurementUnits = idMeasurementUnits.idPoints
    .VerticalMeasurementUnits = idMeasurementUnits.idPoints
End With
Rem At this point, you can perform any series of script actions that
Rem depend on the measurement units you've set. At the end of the
Rem script, reset the units to their original state.
With myDocument.ViewPreferences
       .HorizontalMeasurementUnits = myOldXUnits
       .VerticalMeasurementUnits = myOldYUnits
End With
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.Documents.Count > 0 Then
    Set myDocument = myInDesign.ActiveDocument
    Rem If the document preset "myDocumentPreset" does not
    Rem already exist, create it.
    Rem Disable normal error handling.
    Err.Clear
    On Error Resume Next
    Set myDocumentPreset = myInDesign.DocumentPresets.Item("myDocumentPreset")
    Rem If the document preset did not exist, the above line
    Rem generates an error. Handle the error.
    If (Err.Number <> 0) Then
        Set myDocumentPreset = myInDesign.DocumentPresets.Add
        myDocumentPreset.Name = "myDocumentPreset"
        Err.Clear
    End If
    Rem Restore normal error handling.
    On Error GoTo 0
    Rem Fill in the properties of the document preset with the corresponding
    Rem properties of the active document.
    With myDocumentPreset
        Rem Note that the following gets the page margins
        rem from the margin preferences of the document to get the margin
        Rem preferences from the active page, replace "myDocument" with
        Rem "myInDesign.activeWindow.activePage" in the following six lines
        Rem (assuming the active window is a layout window).
        .Left = myDocument.MarginPreferences.Left
        .Right = myDocument.MarginPreferences.Right
        .Top = myDocument.MarginPreferences.Top
```

```
        .Bottom = myDocument.MarginPreferences.Bottom
        .ColumnCount = myDocument.MarginPreferences.ColumnCount
        .ColumnGutter = myDocument.MarginPreferences.ColumnGutter
        .DocumentBleedBottomOffset =
        myDocument.DocumentPreferences.DocumentBleedBottomOffset
        .DocumentBleedTopOffset =
        myDocument.DocumentPreferences.DocumentBleedTopOffset
        .DocumentBleedInsideOrLeftOffset =
        myDocument.DocumentPreferences.DocumentBleedInsideOrLeftOffset
        .DocumentBleedOutsideOrRightOffset =
        myDocument.DocumentPreferences.DocumentBleedOutsideOrRightOffset
        .FacingPages = myDocument.DocumentPreferences.FacingPages
        .PageHeight = myDocument.DocumentPreferences.PageHeight
        .PageWidth = myDocument.DocumentPreferences.PageWidth
        .PageOrientation = myDocument.DocumentPreferences.PageOrientation
        .PagesPerDocument = myDocument.DocumentPreferences.PagesPerDocument
        .SlugBottomOffset = myDocument.DocumentPreferences.SlugBottomOffset
        .SlugTopOffset = myDocument.DocumentPreferences.SlugTopOffset
        .SlugInsideOrLeftOffset =
        myDocument.DocumentPreferences.SlugInsideOrLeftOffset
        .SlugRightOrOutsideOffset =
        myDocument.DocumentPreferences.SlugRightOrOutsideOffset
    End With
End If
```

Creating a document preset

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

```
On Error Resume Next
Set myDocumentPreset = myInDesign.DocumentPresets.Item("myDocumentPreset")
If Err.Number <> 0 Then
    Set myDocumentPreset = myInDesign.DocumentPresets.Add
    myDocumentPreset.Name = "myDocumentPreset"
    Err.Clear
End If
On Error GoTo 0
Rem Fill in the properties of the document preset.
With myDocumentPreset
    .PageHeight = "9i"
    .PageWidth = "7i"
    .Left = "4p"
    .Right = "6p"
    .Top = "4p"
    .Bottom = "9p"
    .ColumnCount = 1
    .DocumentBleedBottomOffset = "3p"
    .DocumentBleedTopOffset = "3p"
    .DocumentBleedInsideOrLeftOffset = "3p"
    .DocumentBleedOutsideOrRightOffset = "3p"
    .FacingPages = True
    .PageOrientation = idPageOrientation.idPortrait
    .PagesPerDocument = 1
    .SlugBottomOffset = "18p"
    .SlugTopOffset = "3p"
    .SlugInsideOrLeftOffset = "3p"
    .SlugRightOrOutsideOffset = "3p"
End With
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem Set up the first master spread in a new document.
Set myDocument = myInDesign.Documents.Add
Rem Set up the document.
With myDocument.DocumentPreferences
    .PageHeight = "11i"
    .PageWidth = "8.5i"
    .FacingPages = True
    .PageOrientation = idPageOrientation.idPortrait
End With
Rem Set the document's ruler origin to page origin. This is very important
Rem --if you don't do this, getting objects to the correct position on the
Rem page is much more difficult.
myDocument.ViewPreferences.RulerOrigin = idRulerOrigin.idPageOrigin
With myDocument.MasterSpreads.Item(1)
    Rem Set up the left page (verso).
    With .Pages.Item(1)
        With .MarginPreferences
            .ColumnCount = 3
            .ColumnGutter = "1p"
            .Bottom = "6p"
            Rem "left" means inside "right" means outside.
            .Left = "6p"
            .Right = "4p"
            .Top = "4p"
        End With
        Rem Add a simple footer with a section number and page number.
        With .TextFrames.Add
            .GeometricBounds = Array("61p", "4p", "62p", "45p")
            .InsertionPoints.Item(1).Contents =
            idSpecialCharacters.idSectionMarker
            .InsertionPoints.Item(1).Contents = idSpecialCharacters.idEmSpace
            .InsertionPoints.Item(1).Contents =
            idSpecialCharacters.idAutoPageNumber
            .Paragraphs.Item(1).Justification = idJustification.idLeftAlign
        End With
    End With
    Rem Set up the right page (recto).
    With .Pages.Item(2)
        With .MarginPreferences
            .ColumnCount = 3
            .ColumnGutter = "1p"
            .Bottom = "6p"
            Rem "left" means inside "right" means outside.
            .Left = "6p"
            .Right = "4p"
            .Top = "4p"
        End With
```

```
        Rem Add a simple footer with a section number and page number.
        With .TextFrames.Add
            .GeometricBounds = Array("61p", "6p", "62p", "47p")
            .InsertionPoints.Item(1).Contents =
            idSpecialCharacters.idAutoPageNumber
            .InsertionPoints.Item(1).Contents = idSpecialCharacters.idEmSpace
            .InsertionPoints.Item(1).Contents =
            idSpecialCharacters.idSectionMarker
            .Paragraphs.Item(1).Justification = idJustification.idRightAlign
        End With
    End With
End With
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem Assumes that the active document has a master page named "B-Master"
Rem and at least two pages.
myInDesign.ActiveDocument.Pages.Item(2).AppliedMaster =
myInDesign.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):

```
Rem Assumes that the default master spread name is "A-Master".
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
Rem Create a new master spread.
Set myBMaster = myDocument.MasterSpreads.Add
myBMaster.NamePrefix = "B"
myBMaster.BaseName = "Master"
Rem Apply master spread "A" to the first page of the new master spread.
myInDesign.ActiveDocument.MasterSpreads.Item("B-Master").Pages.Item(1).AppliedMaster
= myInDesign.ActiveDocument.MasterSpreads.Item("A-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.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
With myDocument.MetadataPreferences
    .Author = "Adobe"
    .CopyrightInfoURL = "http://www.adobe.com"
    .CopyrightNotice = "This document is copyrighted."
    .CopyrightStatus = idCopyrightStatus.idYes
    .Description = "Example of xmp metadata scripting in InDesign CS"
    .DocumentTitle = "XMP Example"
    .JobName = "XMP_Example_2004"
    .Keywords = Array("animal", "mineral", "vegetable")
    Rem The metadata preferences object also includes the read-only
    Rem creator, format, creationDate, modificationDate, and serverURL properties that
are
    Rem automatically entered and maintained by InDesign.
    Rem Create a custom XMP container, "email"
    .CreateContainerItem "http://ns.adobe.com/xap/1.0/", "email"
    .SetProperty "http://ns.adobe.com/xap/1.0/", "email/*[1]", "someone@adobe.com"
End With
```

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 myInDesign = CreateObject("InDesign.Application.CS4")
Rem Set the application measurement unit defaults to points.
myInDesign.ViewPreferences.HorizontalMeasurementUnits = idMeasurementUnits.idPoints
myInDesign.ViewPreferences.VerticalMeasurementUnits = idMeasurementUnits.idPoints
Rem Set the application default margin preferences.
With myInDesign.MarginPreferences
    Rem Save the current application default margin preferences.
    myY1 = .Top
    myX1 = .Left
    myY2 = .Bottom
    myX2 = .Right
    Rem Set the application default margin preferences.
    Rem Document baseline grid will be based on 14 points, and
    Rem all margins are set in increments of 14 points.
    .Top = 14 * 4
    .Left = 14 * 4
    .Bottom = 74
    .Right = 14 * 5
End With
Rem Make a new document.
Set myDocument = myInDesign.Documents.Add
myDocument.DocumentPreferences.PageWidth = "7i"
myDocument.DocumentPreferences.PageHeight = "9i"
myDocument.DocumentPreferences.PageOrientation = idPageOrientation.idPortrait
Rem At this point, we can reset the application default margins
Rem to their original state.
With myInDesign.MarginPreferences
    .Top = myY1
    .Left = myX1
    .Bottom = myY2
    .Right = myX2
End With
```

```
Rem Set up the bleed and slug areas.
With myDocument.DocumentPreferences
    Rem Bleed
    .DocumentBleedBottomOffset = "3p"
    .DocumentBleedTopOffset = "3p"
    .DocumentBleedInsideOrLeftOffset = "3p"
    .DocumentBleedOutsideOrRightOffset = "3p"
    Rem Slug
    .SlugBottomOffset = "18p"
    .SlugTopOffset = "3p"
    .SlugInsideOrLeftOffset = "3p"
    .SlugRightOrOutsideOffset = "3p"
End With
Rem Create a color.
Err.Clear
On Error Resume Next
    Rem If the color does not already exist, InDesign will generate an error.
    Set myColor = myDocument.Colors.Item("PageNumberRed")
    If Err.Number <> 0 Then
        Set myColor = myDocument.Colors.Add
        myColor.Name = "PageNumberRed"
        myColor.colorModel = idColorModel.idProcess
        myColor.ColorValue = Array(20, 100, 80, 10)
        Err.Clear
    End If
Rem restore normal error handling
On Error GoTo 0
Rem Next, set up some default styles.
Rem Create up a character style for the page numbers.
Err.Clear
On Error Resume Next
    Rem If the character style does not already exist, InDesign generates an error.
    Set myCharacterStyle = myDocument.CharacterStyles.Item("page_number")
    If Err.Number <> 0 Then
        Set myCharacterStyle = myDocument.CharacterStyles.Add
        myCharacterStyle.Name = "page_number"
        Err.Clear
    End If
Rem restore normal error handling
On Error GoTo 0
myDocument.CharacterStyles.Item("page_number").FillColor =
myDocument.Colors.Item("PageNumberRed")
Rem Create up a pair of paragraph styles for the page footer text.
Rem These styles have only basic formatting.
Err.Clear
On Error Resume Next
    Rem If the paragraph style does not already exist, InDesign generates an error.
    Set myParagraphStyle = myDocument.ParagraphStyles.Item("footer_left")
    If Err.Number <> 0 Then
        Set myParagraphStyle = myDocument.ParagraphStyles.Add
        myParagraphStyle.Name = "footer_left"
        myParagraphStyle.PointSize = 11
        myParagraphStyle.Leading = 14
        Err.Clear
    End If
Rem restore normal error handling
On Error GoTo 0
Err.Clear
```

```
On Error Resume Next
    Rem If the paragraph style does not already exist, InDesign generates an error.
    Set myParagraphStyle = myDocument.ParagraphStyles.Item("footer_right")
    If Err.Number <> 0 Then
        Set myParagraphStyle = myDocument.ParagraphStyles.Add
        myParagraphStyle.Name = "footer_right"
        myParagraphStyle.BasedOn = myDocument.ParagraphStyles.Item("footer_left")
        myParagraphStyle.Justification = idJustification.idRightAlign
        myParagraphStyle.PointSize = 11
        myParagraphStyle.Leading = 14
        Err.Clear
    End If
Rem restore normal error handling
On Error GoTo 0
Rem Create a layer for guides.
Err.Clear
On Error Resume Next
    Set myLayer = myDocument.Layers.Item("GuideLayer")
    If Err.Number <> 0 Then
        Set myLayer = myDocument.Layers.Add
        myLayer.Name = "GuideLayer"
        Err.Clear
    End If
Rem restore normal error handling
On Error GoTo 0
Rem Create a layer for the footer items.
Err.Clear
On Error Resume Next
    Set myLayer = myDocument.Layers.Item("Footer")
    If Err.Number <> 0 Then
        Set myLayer = myDocument.Layers.Add
        myLayer.Name = "Footer"
        Err.Clear
    End If
Rem restore normal error handling
On Error GoTo 0
Rem Create a layer for the slug items.
Err.Clear
On Error Resume Next
    Set myLayer = myDocument.Layers.Item("Slug")
    If Err.Number <> 0 Then
        Set myLayer = myDocument.Layers.Add
        myLayer.Name = "Slug"
        Err.Clear
    End If
Rem restore normal error handling
On Error GoTo 0
Rem Create a layer for the body text.
Err.Clear
On Error Resume Next
    Set myLayer = myDocument.Layers.Item("BodyText")
    If Err.Number <> 0 Then
        Set myLayer = myDocument.Layers.Add
        myLayer.Name = "BodyText"
        Err.Clear
    End If
Rem restore normal error handling
On Error GoTo 0
```

```
With myDocument.ViewPreferences
    .RulerOrigin = idRulerOrigin.idPageOrigin
    .HorizontalMeasurementUnits = idMeasurementUnits.idPoints
    .VerticalMeasurementUnits = idMeasurementUnits.idPoints
End With
Rem Document baseline grid and document grid
With myDocument.GridPreferences
    .BaselineStart = 56
    .BaselineDivision = 14
    .BaselineGridShown = False
    .HorizontalGridlineDivision = 14
    .HorizontalGridSubdivision = 5
    .VerticalGridlineDivision = 14
    .VerticalGridSubdivision = 5
    .DocumentGridShown = False
End With
Rem Document XMP information.
With myDocument.MetadataPreferences
    .Author = "Olav Martin Kvern"
    .CopyrightInfoURL = "http:rem www.adobe.com"
    .CopyrightNotice = "This document is not copyrighted."
    .CopyrightStatus = idCopyrightStatus.idNo
    .Description = "Example 7 x 9 book layout"
    .DocumentTitle = "Example"
    .JobName = "7 x 9 book layout template"
    .Keywords = Array("7 x 9", "book", "template")
    .CreateContainerItem "http://ns.adobe.com/xap/1.0/", "email"
    .SetProperty "http://ns.adobe.com/xap/1.0/", "email/*[1]", "okvern@adobe.com"
End With
Rem Set up the master spread.
With myDocument.MasterSpreads.Item(1)
   With .Pages.Item(1)
       Rem Left and right are reversed for left-hand pages (becoming "inside" and
"outside"--
       Rem this is also true in the InDesign user interface).
       myTopMargin = .MarginPreferences.Top
       myBottomMargin = myDocument.DocumentPreferences.PageHeight -
.MarginPreferences.Bottom
       myRightMargin = myDocument.DocumentPreferences.PageWidth -
.MarginPreferences.Left
       myLeftMargin = .MarginPreferences.Right
       With .Guides.Add
           .ItemLayer = myDocument.Layers.Item("GuideLayer")
           .Orientation = idHorizontalOrVertical.idVertical
           .Location = myLeftMargin
       End With
       With .Guides.Add
           .ItemLayer = myDocument.Layers.Item("GuideLayer")
           .Orientation = idHorizontalOrVertical.idVertical
           .Location = myRightMargin
       End With
       With .Guides.Add
           .ItemLayer = myDocument.Layers.Item("GuideLayer")
           .Orientation = idHorizontalOrVertical.idHorizontal
           .Location = myTopMargin
           .FitToPage = False
       End With
```

```
        With .Guides.Add
            .ItemLayer = myDocument.Layers.Item("GuideLayer")
            .Orientation = idHorizontalOrVertical.idHorizontal
            .Location = myBottomMargin
            .FitToPage = False
        End With
        With .Guides.Add
            .ItemLayer = myDocument.Layers.Item("GuideLayer")
            .Orientation = idHorizontalOrVertical.idHorizontal
            .Location = myBottomMargin + 14
            .FitToPage = False
        End With
        With .Guides.Add
            .ItemLayer = myDocument.Layers.Item("GuideLayer")
            .Orientation = idHorizontalOrVertical.idHorizontal
            .Location = myBottomMargin + 28
            .FitToPage = False
        End With
        Set myLeftFooter = .TextFrames.Add
        myLeftFooter.ItemLayer = myDocument.Layers.Item("Footer")
        myLeftFooter.GeometricBounds = Array(myBottomMargin + 14,
.MarginPreferences.Right, myBottomMargin + 28, myRightMargin)
        myLeftFooter.ParentStory.InsertionPoints.Item(1).Contents =
idSpecialCharacters.idSectionMarker
        myLeftFooter.ParentStory.InsertionPoints.Item(1).Contents =
idSpecialCharacters.idEmSpace
        myLeftFooter.ParentStory.InsertionPoints.Item(1).Contents =
idSpecialCharacters.idAutoPageNumber
        myLeftFooter.ParentStory.Characters.Item(1).AppliedCharacterStyle =
myDocument.CharacterStyles.Item("page_number")
        myLeftFooter.ParentStory.Paragraphs.Item(1).ApplyStyle
myDocument.ParagraphStyles.Item("footer_left"), False
        Rem Slug information.
        myDate = Date
        With myDocument.MetadataPreferences
            myString = "Author:" & vbTab & .Author & vbTab & "Description:" & vbTab &
.Description & vbCrLf & _
            "Creation Date:" & vbTab & myDate & vbTab & "Email Contact" & vbTab &
.GetProperty("http://ns.adobe.com/xap/1.0/", "email/*[1]")
        End With
        Set myLeftSlug = .TextFrames.Add
        myLeftSlug.ItemLayer = myDocument.Layers.Item("Slug")
        myLeftSlug.GeometricBounds = Array(myDocument.DocumentPreferences.PageHeight +
36, .MarginPreferences.Right, myDocument.DocumentPreferences.PageHeight + 144,
myRightMargin)
        myLeftSlug.Contents = myString
        myLeftSlug.ParentStory.Texts.Item(1).ConvertToTable
        Rem Body text master text frame.
        Set myLeftFrame = .TextFrames.Add
        myLeftFrame.ItemLayer = myDocument.Layers.Item("BodyText")
        myLeftFrame.GeometricBounds = Array(.MarginPreferences.Top,
.MarginPreferences.Right, myBottomMargin, myRightMargin)
    End With
    With .Pages.Item(2)
        myTopMargin = .MarginPreferences.Top
        myBottomMargin = myDocument.DocumentPreferences.PageHeight -
.MarginPreferences.Bottom
    myRightMargin = myDocument.DocumentPreferences.PageWidth -
.MarginPreferences.Right
        myLeftMargin = .MarginPreferences.Left
```

```
        With .Guides.Add
            .ItemLayer = myDocument.Layers.Item("GuideLayer")
            .Orientation = idHorizontalOrVertical.idVertical
            .Location = myLeftMargin
        End With
        With .Guides.Add
            .ItemLayer = myDocument.Layers.Item("GuideLayer")
            .Orientation = idHorizontalOrVertical.idVertical
            .Location = myRightMargin
        End With
        Set myRightFooter = .TextFrames.Add
        myRightFooter.ItemLayer = myDocument.Layers.Item("Footer")
        myRightFooter.GeometricBounds = Array(myBottomMargin + 14,
.MarginPreferences.Left, myBottomMargin + 28, myRightMargin)
        myRightFooter.ParentStory.InsertionPoints.Item(1).Contents =
idSpecialCharacters.idAutoPageNumber
        myRightFooter.ParentStory.InsertionPoints.Item(1).Contents =
idSpecialCharacters.idEmSpace
        myRightFooter.ParentStory.InsertionPoints.Item(1).Contents =
idSpecialCharacters.idSectionMarker
        myRightFooter.ParentStory.Characters.Item(-1).AppliedCharacterStyle =
myDocument.CharacterStyles.Item("page_number")
        myRightFooter.ParentStory.Paragraphs.Item(1).ApplyStyle
myDocument.ParagraphStyles.Item("footer_right"), False
        Rem Slug information.
        Set myRightSlug = .TextFrames.Add
        myRightSlug.ItemLayer = myDocument.Layers.Item("Slug")
        myRightSlug.GeometricBounds = Array(myDocument.DocumentPreferences.PageHeight +
36, myLeftMargin, myDocument.DocumentPreferences.PageHeight + 144, myRightMargin)
        myRightSlug.Contents = myString
        myRightSlug.ParentStory.Texts.Item(1).ConvertToTable
        Rem Body text master text frame.
        Set myRightFrame = .TextFrames.Add
        myRightFrame.ItemLayer = myDocument.Layers.Item("BodyText")
        myRightFrame.GeometricBounds = Array(.MarginPreferences.Top,
.MarginPreferences.Left, myBottomMargin, myRightMargin)
        myRightFrame.PreviousTextFrame = myLeftFrame
    End With
End With
Rem Add section marker text--this text will appear in the footer.
myDocument.Sections.Item(1).Marker = "Section 1"
Rem When you link the master page text frames, one of the frames
Rem sometimes becomes selected. Deselect it.
myInDesign.Select idNothingEnum.idNothing
```

Printing a document

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

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem The PrintOut method has two optional parameters:
Rem PrintDialog--if true, display the Print dialog box
Rem Using--the printer preset to use. The following line
Rem prints the document using the default settings and
Rem without displaying the Print dialog box.
myInDesign.ActiveDocument.PrintOut False
```

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):

```
Rem Prints a page range from the active document.
Rem The page range can be either idPageRange.idAllPages or a page range string.
Rem A page number entered in the page range must correspond to a page
Rem name in the document (i.e., not the page index). If the page name is
Rem not found, InDesign will display an error message.
myInDesign.ActiveDocument.PrintPreferences.PageRange = "1-3, 6, 9"
myInDesign.ActiveDocument.PrintOut 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):

```
Rem PrintPreferences.vbs
Rem An InDesign CS4 VBScript
Rem Sets the print preferences of the active document.
Set myInDesign = CreateObject("InDesign.Application.CS4")
With myInDesign.ActiveDocument.PrintPreferences
    Rem Properties corresponding to the controls in the General panel
    Rem of the Print dialog box.
    Rem ActivePrinterPreset is ignored in this example--we'll set our own
    Rem print preferences.
    Rem printer can be either a string (the name of the printer) or
    Rem idPrinter.idPostscriptFile.
    .Printer = "AGFA-SelectSet 5000SF v2013.108"
    Rem If the printer property is the name of a printer, then the ppd property
    Rem is locked (and will return an error if you try to set it).
    Rem ppd = "AGFA-SelectSet5000SF"
    Rem If the printer property is set to Printer.postscript file, the copies
    Rem property is unavailable. Attempting to set it will generate an error.
    .Copies = 1
    Rem If the printer property is set to Printer.postscript file, or if the
    Rem selected printer does not support collation, then the collating
    Rem property is unavailable. Attempting to set it will generate an error.
    Rem collating = false
    .ReverseOrder = False
    Rem pageRange can be either PageRange.allPages or a page range string.
    .PageRange = idPageRange.idAllPages
    .PrintSpreads = False
    .PrintMasterPages = False
    Rem If the printer property is set to Printer.postScript file, then
    Rem the printFile property contains the file path to the output file.
    Rem printFile = "/c/test.ps"
    .Sequence = idSequences.idAll
    Rem If trapping is set to either idTrapping.idApplicationBuiltIn
    Rem or idTrapping.idAdobeInRIP,
    Rem then setting the following properties will produce an error.
```

```
If (.ColorOutput = idColorOutputModes.idInRIPSeparations) Or _
(.ColorOutput = idColorOutputModes.idSeparations) Then
    If .Trapping = idTrapping.idOff Then
        .PrintBlankPages = False
        .PrintGuidesGrids = False
        .PrintNonprinting = False
    End If
End If
Rem-------------------------------------------------------------------------
Rem Properties corresponding to the controls in the Setup panel
Rem of the Print dialog box.
Rem-------------------------------------------------------------------------
.PaperSize = idPaperSizes.idCustom
Rem Page width and height are ignored if paperSize is not PaperSizes.custom.
Rem .PaperHeight = 1200
Rem .PaperWidth = 1200
.PrintPageOrientation = idPrintPageOrientation.idPortrait
.PagePosition = idPagePositions.idCentered
.PaperGap = 0
.PaperOffset = 0
.PaperTransverse = False
.ScaleHeight = 100
.ScaleWidth = 100
.ScaleMode = idScaleModes.idScaleWidthHeight
.ScaleProportional = True
Rem If trapping is set to either idTrapping.idApplicationBuiltIn or
Rem idTrapping.idAdobeInRIP,then setting the following properties will
Rem produce an error.
If (.ColorOutput = idColorOutputModes.idInRIPSeparations) Or _
(.ColorOutput = idColorOutputModes.idSeparations) Then
    If .Trapping = idTrapping.idOff Then
        .TextAsBlack = False
        .Thumbnails = False
        Rem The following properties is not needed because thumbnails
        Rem is set to false.
        Rem thumbnailsPerPage = 4
        .Tile = False
        Rem The following properties are not needed
        Rem because tile is set to false.
        Rem .TilingOverlap = 12
        Rem .TilingType = TilingTypes.auto
    End If
End If
Rem-------------------------------------------------------------------------
Rem Properties corresponding to the controls in the Marks and Bleed
Rem panel of the Print dialog box.
Rem-------------------------------------------------------------------------
Rem Set the following property to true to print all printer's marks.
Rem allPrinterMarks = true
.UseDocumentBleedToPrint = False
Rem If useDocumentBleedToPrint = false then setting any of
Rem the bleed properties will result in an error.
Rem Get the bleed amounts from the document's bleed and add a bit.
.BleedBottom = myInDesign.ActiveDocument.DocumentPreferences.
DocumentBleedBottomOffset + 3
.BleedTop = myInDesign.ActiveDocument.
DocumentPreferences.DocumentBleedTopOffset + 3
.BleedInside = myInDesign.ActiveDocument.DocumentPreferences.
DocumentBleedInsideOrLeftOffset + 3
.BleedOutside = myInDesign.ActiveDocument.DocumentPreferences.
```

```
DocumentBleedOutsideOrRightOffset + 3
Rem If any bleed area is greater than zero, then print the bleed marks.
If ((.BleedBottom = 0) And (.BleedTop = 0) And (.BleedInside = 0) _
And (.BleedOutside = 0)) Then
    .BleedMarks = True
Else
    .BleedMarks = False
End If
.ColorBars = True
.CropMarks = True
.IncludeSlugToPrint = False
.MarkLineWeight = idMarkLineWeight.idP125pt
.MarkOffset = 6
Rem .MarkType = MarkTypes.default
.PageInformationMarks = True
.RegistrationMarks = True
Rem-------------------------------------------------------------------------
Rem Properties corresponding to the controls in the
Rem Output panel of the Print dialog box.
Rem-------------------------------------------------------------------------
.Negative = True
.ColorOutput = idColorOutputModes.idSeparations
Rem Note the lowercase "i" in "Builtin"
.Trapping = idTrapping.idApplicationBuiltin
.Screening = "175 lpi/2400 dpi"
.Flip = idFlip.idNone
Rem The following options are only applicable if trapping is set to
Rem idTrapping.idAdobeInRIP.
If .Trapping = idTrapping.idAdobeInRIP Then
    .PrintBlack = True
    .PrintCyan = True
    .PrintMagenta = True
    .PrintYellow = True
End If
Rem Only change the ink angle and frequency when you want to override the
Rem screening set by the screening specified by the screening property.
Rem .BlackAngle = 45
Rem .BlackFrequency = 175
Rem .CyanAngle = 15
Rem .CyanFrequency = 175
Rem .MagentaAngle = 75
Rem .MagentaFreqency = 175
Rem .YellowAngle = 0
Rem .YellowFrequency = 175
Rem The following properties are not needed (because colorOutput
Rem is set to separations).
Rem .CompositeAngle = 45
Rem .CompositeFrequency = 175
Rem .SimulateOverprint = false
Rem-------------------------------------------------------------------------
Rem Properties corresponding to the controls in the Graphics
Rem panel of the Print dialog box.
Rem-------------------------------------------------------------------------
.SendImageData = idImageDataTypes.idAllImageData
.FontDownloading = idFontDownloading.idComplete
Err.Clear
```

```
    On Error Resume Next
        .DownloadPPDFonts = True
        .DataFormat = idDataFormat.idBinary
        .PostScriptLevel = idPostScriptLevels.idLevel3
        If Err.Number <> 0 Then
            Err.Clear
        End If
    On Error GoTo 0
    Rem--------------------------------------------------------------------------
    Rem Properties corresponding to the controls in the Color
    Rem Management panel of the Print dialog box.
    Rem--------------------------------------------------------------------------
    Rem If the UseColorManagement property of myInDesign.ColorSettings is false,
    Rem attempting to set the following properties will return an error.
    Err.Clear
    On Error Resume Next
        .SourceSpace = SourceSpaces.useDocument
        .Intent = RenderingIntent.useColorSettings
        .CRD = ColorRenderingDictionary.useDocument
        .Profile = Profile.postscriptCMS
        If Err.Number <> 0 Then
            Err.Clear
        End If
    On Error GoTo 0
    Rem--------------------------------------------------------------------------
    Rem Properties corresponding to the controls in the
    Rem Advanced panel of the Print dialog box.
    Rem--------------------------------------------------------------------------
    .OPIImageReplacement = False
    .OmitBitmaps = False
    .OmitEPS = False
    .OmitPDF = False
    Rem The following line assumes that you have a flattener
    Rem preset named "high quality flattener".
    Err.Clear
    On Error Resume Next
        .FlattenerPresetName = "high quality flattener"
        If Err.Number <> 0 Then
            Err.Clear
        End If
    On Error GoTo 0
    .IgnoreSpreadOverrides = False
End With
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
myInDesign.ActiveDocument.Export idExportFormat.idPDFType, "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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
myInDesign.ActiveDocument.Export idExportFormat.idPDFType, "c:\myTestDocument.pdf",
False, myInDesign.PDFExportPresets.Item("[Press]")
```

Setting PDF export options

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

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
With myInDesign.PDFExportPreferences
    Rem Basic PDF output options.
    .PageRange = idPageRange.idAllPages
    .AcrobatCompatibility = idAcrobatCompatibility.idAcrobat6
    .ExportGuidesAndGrids = False
    .ExportLayers = False
    .ExportNonprintingObjects = False
    .ExportReaderSpreads = False
    .GenerateThumbnails = False
    On Error Resume Next
    .IgnoreSpreadOverrides = False
    .IncludeICCProfiles = True
    On Error GoTo 0
    .IncludeBookmarks = True
    .IncludeHyperlinks = True
    .IncludeSlugWithPDF = False
    .IncludeStructure = False
    .InteractiveElements = False
    Rem Setting subsetFontsBelow to zero disallows font subsetting
    Rem set subsetFontsBelow to some other value to use font subsetting.
    .SubsetFontsBelow = 0
    Rem Bitmap compression/sampling/quality options
    Rem (note the additional "s" in "compression").
    .ColorBitmapCompression = idBitmapCompression.idZip
    .ColorBitmapQuality = idCompressionQuality.idEightBit
    .ColorBitmapSampling = idSampling.idNone
```

```
    Rem ThresholdToCompressColor is not needed in this example.
    Rem ColorBitmapSamplingDPI is not needed when
    Rem ColorBitmapSampling is set to none.
    .GrayscaleBitmapCompression = idBitmapCompression.idZip
    .GrayscaleBitmapQuality = idCompressionQuality.idEightBit
    .GrayscaleBitmapSampling = idSampling.idNone
    Rem ThresholdToCompressGray is not needed in this
    Rem example.
    Rem GrayscaleBitmapSamplingDPI is not needed when
    Rem GrayscaleBitmapSampling is set to none.
    .MonochromeBitmapCompression = idBitmapCompression.idZip
    .MonochromeBitmapSampling = idSampling.idNone
    Rem ThresholdToCompressMonochrome is not needed in this example.
    Rem MonochromeBitmapSamplingDPI is not needed when
    Rem MonochromeBitmapSampling is set to none.
    Rem Other compression options.
    .CompressionType = idPDFCompressionType.idCompressNone
    .CompressTextAndLineArt = True
    .ContentToEmbed = idPDFContentToEmbed.idEmbedAll
    .CropImagesToFrames = True
    .OptimizePDF = True
    Rem Printers marks and prepress options.
    Rem Get the bleed amounts from the document's bleed.
    .BleedBottom = myInDesign.ActiveDocument.DocumentPreferences.
    DocumentBleedBottomOffset
    .BleedTop = myInDesign.ActiveDocument.DocumentPreferences.
    DocumentBleedTopOffset
    .BleedInside = myInDesign.ActiveDocument.DocumentPreferences.
    DocumentBleedInsideOrLeftOffset
    .BleedOutside = myInDesign.ActiveDocument.DocumentPreferences.
    DocumentBleedOutsideOrRightOffset
    Rem If any bleed area is greater than zero, then export the bleed marks.
    If ((.BleedBottom = 0) And (.BleedTop = 0) And (.BleedInside = 0) And
    (.BleedOutside = 0)) Then
        .BleedMarks = True
    Else
        .BleedMarks = False
    End If
    .ColorBars = True
    Rem ColorTileSize and GrayTileSize are only used when
    Rem the export format is set to JPEG2000.
    Rem .ColorTileSize = 256
    Rem .GrayTileSize = 256
    .CropMarks = True
    .OmitBitmaps = False
    .OmitEPS = False
    .OmitPDF = False
    .PageInformationMarks = True
    .PageMarksOffset = 12
    .PDFColorSpace = idPDFColorSpace.idUnchangedColorSpace
    .PDFMarkType = idMarkTypes.idDefault
    .PrinterMarkWeight = idPDFMarkWeight.idP125pt
    .RegistrationMarks = True
    On Error Resume Next
    .SimulateOverprint = False
    On Error GoTo 0
    .UseDocumentBleedWithPDF = True
    Rem Set viewPDF to true to open the PDF in Acrobat or Adobe Reader.
    .ViewPDF = False
End With
```

```
Rem Now export the document. You'll have to fill in your own file path.
myInDesign.ActiveDocument.Export idExportFormat.idPDFType, "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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem Enter the names of the pages you want to export in the following line.
Rem Note that the page name is not necessarily the index of the page in the
Rem document (e.g., the first page of a document whose page numbering starts
Rem with page 21 will be "21", not 1).
myInDesign.PDFExportPreferences.PageRange = "1-3, 6, 9"
Rem Fill in your own file path.
myFile = "c:\myTestFile.pdf"
myInDesign.ActiveDocument.Export idExportFormat.idPDFType, myFile, False
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
If myInDesign.Documents.Count <> 0 Then
    Set myDocument = myInDesign.ActiveDocument
    Rem VBScript doesn't have a native "get folder" statement, so we'll use
    Rem InDesign's JavaScript to display a folder browser.
    myJavaScript = "myFolder = Folder.selectDialog(""Choose a Folder"");
    myFolderName = myFolder.fsName;"
    Rem Run the string "myJavaScript" as a JavaScript
    myFolderName = myInDesign.DoScript(myJavaScript,
    idScriptLanguage.idJavascript)
    If myFileSystemObject.FolderExists(myFolderName) Then
        myExportPages myInDesign, myDocument, myFolderName
    End If
End If
Function myExportPages(myInDesign, myDocument, myFolderName)
    myDocumentName = myDocument.Name
    Set myDialog = myInDesign.Dialogs.Add
    With myDialog
        .Name = "ExportPages"
        With .DialogColumns.Add
            With .DialogRows.Add
                With .StaticTexts.Add
                    .StaticLabel = "Base Name:"
                End With
                Set myBaseNameField = .TextEditboxes.Add
                myBaseNameField.EditContents = myDocumentName
                myBaseNameField.MinWidth = 160
            End With
        End With
    End With
    myResult = myDialog.Show
```

```
    If myResult = True Then
        myBaseName = myBaseNameField.EditContents
        Rem Remove the dialog box from memory.
        myDialog.Destroy
        For myCounter = 1 To myDocument.Pages.Count
            myPageName = myDocument.Pages.Item(myCounter).Name
            myInDesign.PDFExportPreferences.PageRange = myPageName
            Rem Generate a file path from the folder name,
            Rem the base document name, and the page name.
            Rem Replace the colons in the page name (e.g., "Sec1:1") wtih
            Rem underscores.
            myPageName = Replace(myPageName, ":", "_")
            myFilePath = myFolderName & "\" & myBaseName & "_" &
            myPageName & ".pdf"
            myDocument.Export idExportFormat.idPDFType, myFilePath, False
        Next
    Else
        myDialog.Destroy
    End If
End Function
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
myFile = "c:\myTestFile.eps"
myInDesign.ActiveDocument.Export idExportFormat.idEPSType, 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.)

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem Enter the name of the page you want to export in the following line.
Rem Note that the page name is not necessarily the index of the page in the
Rem document (e.g., the first page of a document whose page numbering starts
Rem with page 21 will be "21", not 1).
myInDesign.EPSExportPreferences.PageRange = "1-3, 6, 9"
Rem Fill in your own file path.
myFile = "c:\myTestFile.eps"
myInDesign.ActiveDocument.Export idExportFormat.idEPSType, 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.)

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
If myInDesign.Documents.Count <> 0 Then
    Set myDocument = myInDesign.ActiveDocument
    Rem VBScript doesn't have a native "get folder" statement, so we'll use
    Rem InDesign's JavaScript to display a folder browser.
    myJavaScript = "myFolder = Folder.selectDialog(""Choose a Folder"");
    myFolderName = myFolder.fsName;"
    Rem Run the string "myJavaScript" as a JavaScript
    myFolderName = myInDesign.DoScript(myJavaScript,
    idScriptLanguage.idJavascript)
    If myFileSystemObject.FolderExists(myFolderName) Then
        myExportEPSPages myInDesign, myDocument, myFolderName
    End If
End If
Function myExportEPSPages(myInDesign, myDocument, myFolderName)
    myDocumentName = myDocument.Name
    Set myDialog = myInDesign.Dialogs.Add
    With myDialog
        .Name = "ExportPages"
        With .DialogColumns.Add
            With .DialogRows.Add
                With .StaticTexts.Add
                    .StaticLabel = "Base Name:"
                End With
                Set myBaseNameField = .TextEditboxes.Add
                myBaseNameField.EditContents = myDocumentName
                myBaseNameField.MinWidth = 160
            End With
        End With
    End With
    myResult = myDialog.Show
    If myResult = True Then
        myBaseName = myBaseNameField.EditContents
        Rem Remove the dialog box from memory.
        myDialog.Destroy
        For myCounter = 1 To myDocument.Pages.Count
            myPageName = myDocument.Pages.Item(myCounter).Name
            myInDesign.EPSExportPreferences.PageRange = myPageName
            Rem Generate a file path from the folder name, the base document
            Rem name, and the page name.
            Rem Replace the colons in the page name (e.g., "Sec1:1") with
            Rem underscores.
            myPageName = Replace(myPageName, ":", "_")
            myFilePath = myFolderName & "\" & myBaseName & "_" & _
            myPageName & ".eps"
            myDocument.Export idExportFormat.idEPSType, myFilePath, False
        Next
    Else
        myDialog.Destroy
    End If
End Function
```

4 Working with Page Items

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

This document discusses the following:

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

Creating Page Items

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

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

```
Rem Given a page "myPage", create a new rectangle at the default size and location
myPage.Rectangles.Add
```

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

```
Rem Given a page "myPage", create a new rectangle and specify its size and location...
Set myRectangle = myPage.Rectangles.Add
myRectangle.GeometricBounds = Array(72, 72, 144, 144)
```

Page Item Types

It is important to note that you cannot create a "generic" page item--you have to create a page item of a specific type (a rectangle, oval, graphic line, polygon, text frame, or button). You will also notice that InDesign changes the type of a page item as the geometry of the page item changes. A rectangle, for example, is always made up of a single, closed path containing four path points and having 90 degree

interior angles. Change the location of a single point, however, or add another path, and the type of the page item changes to a polygon. Open the path and remove two of the four points, and InDesign will change the type to a graphic line. The only things that define the type of a rectangle, ellipse, graphic line, or polygon are:

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

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

```
myPageItemType = TypeName(myPageItem)
```

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

Getting the Type of a Page Item

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

```
Rem Given a generic page item "myPageItem"...
myType = TypeName(myPageItem)
MsgBox myType
```

Referring to Page Items

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

```
Rem Given a reference to InDesign "myInDesign"...
SSet myPageItems = myInDesign.Documents.Item(1).Pages.Item(1).PageItems
```

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

```
Rem Given a reference to InDesign "myInDesign"...
Set myAllPageItems = myInDesign.Documents.Item(1).Pages.Item(1).AllPageItems
```

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

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

```
Rem Given a reference to InDesign "myInDesign"...
Set myPageItems = myInDesign.Documents.Item(1).Pages.Item(1).PageItems.Item("myLabel”)
```

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

Page Item Geometry

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

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

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

Working with Paths and Path Points

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

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

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

```
Array(Array(x1, y1), Array(x2, y2), ...)
```

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

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

```
Array(Array(Array(xL1, YL1), Array(x1, y1), Array(xR1, yR1)), Array(Array(xL2, YL2),
Array(x2, y2), Array(xR2, yR2)), ...)
```

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

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

```
Array(Array(Array(xL1, YL1), Array(x1, y1), Array(xR1, yR1)), Array(x2, y2), ...)
```

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

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

```
Rem Given a graphic line "myGraphicLine"...
Set myPathPoint = myGraphicLine.Paths.Item(1).PathPoints.Add
Rem Move the path point to a specific location.
myPathPoint.Anchor = Array(144, 144)
```

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

```
Rem Given a polygon "myPolygon", remove the
Rem last path point in the first path.
myPolygon.Paths.Item(1).PathPoints.Item(-1).Delete
```

Grouping Page Items

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

```
Rem Given a page "myPage" containing at least two ovals and two rectangles...
ReDim myArray(0)
Rem Add the items to the array.
myPush myArray, myPage.Rectangles.Item(1), True
myPush myArray, myPage.Rectangles.Item(2), True
myPush myArray, myPage.Ovals.Item(1), True
myPush myArray, myPage.Ovals.Item(1), True
Rem Group the items.
myPage.Groups.Add myArray
```

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

```
Rem Given a group "myGroup"...
Set myPageItems = myGroup.Ungroup
```

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

Duplicating and Moving Page Items

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

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

```
Rem Given a reference to a rectangle "myRectangle"...
Rem Move the rectangle to the location (12, 12).
Rem Absolute move:
myRectangle.Move Array(12, 12)
Rem Move the rectangle *by* 12 points horizontally, 12 points vertically.
Rem Relative move (note empty first parameter):
myRectangle.Move , Array(12, 12)
Rem Move the rectangle to another page (rectangle appears at (0,0).
Set myPage = myInDesign.Documents.Item(1).Pages.Add
myRectangle.Move myPage
Rem To move a page item to another document, use the Duplicate method.
```

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

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

```
Rem Given a reference to a rectangle "myRectangle"...
Rem Duplicate the rectangle and move the
Rem duplicate to the location (12, 12).
Rem Absolute move:
Set myDuplicate = myRectangle.Duplicate(Array(12, 12))
Rem Duplicate the rectangle and move the duplicate *by* 12
Rem points horizontally, 12 points vertically.
Rem Relative move (note empty first parameter):
Set myDuplicate = myRectangle.Duplicate (, Array(12, 12))
Rem Duplicate the rectangle to another page (rectangle appears at (0,0).
Set myPage = myInDesign.Documents.Item(1).Pages.Add
Set myDuplicate = myRectangle.Duplicate(myPage)
Rem Duplicate the rectangle to another document.
Set myDocument = myInDesign.Documents.Add
Set myDuplicate = myRectangle.Duplicate(myDocument.Pages.Item(1))
```

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

Creating Compound Paths

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

```
Rem Given a rectangle "myRectangle" and an Oval "myOval"...
myRectangle.MakeCompoundPath myOval
```

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

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

```
Rem Given a polygon "myPolygon" (all compound paths are type Polygon)...
Set mPageItems = myPolygon.ReleaseCompoundPath
```

Using Pathfinder Operations

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

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

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

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

```
Rem Given a rectangle "myRectangle" and an Oval "myOval"...
myRectangle.AddPath myOval
```

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

```
Rem Given a rectangle "myRectangle" and an Oval "myOval"...
myRectangle.ExcludeOverlapPath myOval
```

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

```
Rem Given a rectangle "myRectangle" and an Oval "myOval"...
myRectangle.IntersectPath myOval
```

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

```
Rem Given a rectangle "myRectangle" and an Oval "myOval"...
myRectangle.MinusBack myOval
```

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

```
Rem Given a rectangle "myRectangle" and an Oval "myOval"...
myOval.SubtractPath myRectangle
```

Converting Page Item Shapes

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

```
Rem Given a rectangle "myRectangle"...
myRectangle.ConvertShape idConvertShapeOptions.idConvertToRoundedRectangle
```

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

```
Rem Given a rectangle "myRectangle"...
myRectangle.ConvertShape idConvertShapeOptions.idConvertToOpenPath
```

Arranging Page Items

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

```
Rem Given a rectangle "myRectangle" and an oval "myOval",
Rem where "myOval" is in front of "myRectangle", bring
Rem the rectangle to the front...
myRectangle.BringToFront
```

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

```
Rem Given a rectangle "myRectangle" and a layer "myLayer",
Rem send the rectangle to the layer...
myRectangle.ItemLayer = myInDesign.Documents.Item(1).Layers.Item("myLayer")
```

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

```
Rem Given a layer "myLayer", move the layer behind
Rem the default layer (the lowest layer in the document
Rem is Layers.Item(-1).
myLayer.Move idLocationOptions.idAfter, myInDesign.Documents.Item(1).Layers.Item(-1)
```

Transforming Page Items

Operations that change the geometry of items on an InDesign page are called *transformations*. Transformations include scaling, rotation, shearing (skewing), and movement (or translation). In scripting, you apply transformations using the `transform` method. This one method replaces the `resize`, `rotate`, and `shear` methods used in versions of InDesign prior to InDesign CS3 (5.0).

This document shows you how to transform objects and discusses some of the technical details behind the transformation architecture.

Using the transform method

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

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

To transform an object, you follow two steps:

1. Create a transformation matrix.
2. Apply the transformation matrix to the object using the `transform` method. When you do this, you also specify the coordinate system in which the transformation is to take place. For more on coordinate systems, see "Coordinate spaces" on page 55. In addition, you specify the center of transformation, or transformation origin. For more on specifying the transformation origin, see "Transformation origin" on page 57.

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

```
Rem Rotate a rectangle "myRectangle" around its center point.
set myRotateMatrix = myInDesign.transformationMatrices.add(, , , 27)
myRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates,
idAnchorPoint.idCenterAnchor, myRotateMatrix
Rem Scale a rectangle "myRectangle" around its center point.
set myScaleMatrix = myInDesign.TransformationMatrices.Add(.5, .5)
myRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates,
idAnchorPoint.idCenterAnchor, myScaleMatrix
Rem Shear a rectangle "myRectangle" around its center point.
set myShearMatrix = myInDesign.TransformationMatrices.Add(, , 30)
myRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates,
idAnchorPoint.idCenterAnchor, myShearMatrix
Rem Rotate a rectangle "myRectangle" around a specified ruler point ([72, 72]).
Set myRotateMatrix = myInDesign.transformationMatrices.add(, , , 27)
myRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates, Array(Array(72, 72),
idAnchorPoint.idCenterAnchor), myRotateMatrix, , True
Rem Scale a rectangle "myRectangle" around a specified ruler point ([72, 72]).
Set myScaleMatrix = myInDesign.transformationMatrices.add(.5, .5)
myRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates, Array(Array(72, 72),
idAnchorPoint.idCenterAnchor), myScaleMatrix, , True
```

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

Working with transformation matrices

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

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

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

```
Rem The following statements are equivalent
Rem (0.25881904510252 is the sine of 15 degrees; 0.96592582628907, the cosine).
Set myTransformationMatrix = myTransformationMatrix.RotateMatrix(15)
Set myTransformationMatrix = myTransformationMatrix.RotateMatrix(, 0.96592582628907);
Set myTransformationMatrix = myTransformationMatrix.RotateMatrix(, ,
0.25881904510252);
```

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

```
Set myRectangle = myInDesign.Documents.Item(1).Pages.Item(1).Rectangles.Item(1)
Set myTransformationMatrix = myInDesign.TransformationMatrices.Add(, , 0)
Rem ShearMatrix can take the following parameters: byAngle, bySlope
Rem Replace the current matrix with the sheared matrix.
Set myTransformationMatrix = myTransformationMatrix.ShearMatrix(45)
myRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates,
idAnchorPoint.idCenterAnchor, myTransformationMatrix
Rem The following statements are equivalent.
Rem slope = rise/run--so a 45 degree slope is 1.
Set myTransformationMatrix = myTransformationMatrix.shearMatrix(45)
Set myTransformationMatrix = myTransformationMatrix.shearMatrix(, 1)
```

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

```
Set myRectangle = myInDesign.Documents.Item(1).Pages.Item(1).Rectangles.Item(1)
Set myTransformationMatrix = myInDesign.TransformationMatrices.Add(, , , 30, 12, 12)
myRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates,
idAnchorPoint.idCenterAnchor, myTransformationMatrix
Set myNewRectangle = myRectangle.Duplicate
Rem Move the duplicated rectangle to the location of the original
Rem rectangle by inverting, then applying the transformation matrix.
Set myTransformationMatrix = myTransformationMatrix.InvertMatrix
myRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates,
idAnchorPoint.idCenterAnchor, myTransformationMatrix
```

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

```
Rem Transformation matrix with counterclockwise rotation angle = 30.
Set myTransformationMatrixA = myInDesign.TransformationMatrices.Add(, , , 30)
Rem Transformation matrix with horizontal translation = 12,
Rem vertical translation = 12.
Set myTransformationMatrixB = myInDesign.TransformationMatrices.Add(, , , , 12, 12)
Set myRectangle = myInDesign.Documents.Item(1).Pages.Item(1).Rectangles.Item(1)
Set myNewRectangle = myRectangle.Duplicate
Rem Rotate the duplicated rectangle.
myNewRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates,
idAnchorPoint.idCenterAnchor, myTransformationMatrixA
Set myNewRectangle = myRectangle.Duplicate
Rem Move the duplicate (unrotated) rectangle.
myNewRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates,
idAnchorPoint.idCenterAnchor, myTransformationMatrixB)
Rem Merge the two transformation matrices.
Set myTransformationMatrix =
myTransformationMatrixA.CatenateMatrix(myTransformationMatrixB)
Set myNewRectangle = myRectangle.Duplicate
Rem The duplicated rectangle will be both moved and rotated.
myNewRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates,
idAnchorPoint.idCenterAnchor, myTransformationMatrix
```

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

```
Set myRectangle = myInDesign.Documents.Item(1).Pages.Item(1).Rectangles.Item(1)
Rem Note that transformValuesOf() always returns an array containing a single
transformationMatrix.
Set myTransformArray =
myRectangle.TransformValuesOf(idCoordinateSpaces.idPasteboardCoordinates)
Set myTransformationMatrix = myTransformArray.Item(1)
myRotationAngle = myTransformationMatrix.CounterclockwiseRotationAngle
myShearAngle = myTransformationMatrix.ClockwiseShearAngle
myXScale = myTransformationMatrix.HorizontalScaleFactor
myYScale = myTransformationMatrix.VerticalScaleFactor
myXTranslate = myTransformationMatrix.HorizontalTranslation
myYTranslate = myTransformationMatrix.VerticalTranslation
myString = "Rotation Angle: " & myRotationAngle & vbCr
myString = myString & "Shear Angle: " & myShearAngle & vbCr
myString = myString & "Horizontal Scale Factor: " & myXScale & vbCr
myString = myString & "Vertical Scale Factor: " & myYScale & vbCr
myString = myString & "Horizontal Translation: " & myXTranslate & vbCr
myString = myString & "Vertical Translation: " & myYTranslate & vbCr & vbCr
myString = myString & "Note that the Horizontal Translation and" & vbCr
myString = myString & "Vertical Translation values are the location" & vbCr
myString = myString & "of the center anchor in pasteboard coordinates."
MsgBox myString
```

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

Coordinate spaces

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

- **idCoordinateSpaces.idPasteboardCoordinates** is the coordinate space of the entire InDesign document. This space uses points as units and extends across all spreads in a document. It does not correspond to InDesign's rulers or zero point. Transformations applied to objects have no effect on this coordinate space (e.g., the angle of the horizontal and vertical axes do not change).

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

- **idCoordinateSpaces.idInnerCoordinates** is the coordinate space of the object itself.

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

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

```
Set myRectangle =
myInDesign.Documents.Item(1).Pages.Item(1).Groups.Item(1).Rectangles.Item(1)'
myString = "The page contains a group which has been" & vbCr
myString = myString & "rotated 45 degrees (counterclockwise)." & vbCr
myString = myString & "The rectangle inside the group was" & vbCr
myString = myString & "rotated 45 degrees counterclockwise" & vbCr
myString = myString & "before it was added to the group." & vbCr & vbcr
myString = myString & "Watch as we apply a series of scaling" & vbCr
myString = myString & "operations in different coordinate spaces." & vbCr & vbCr
myString = myString & "(You might need to move the alert aside" & vbCr
myString = myString & "to see the effect of the transformations.)" & vbCr
MsgBox myString
Rem Create a transformation matrix with horizontal scale factor = 2.
Set myTransformationMatrix = myInDesign.TransformationMatrices.Add(2)
Rem Transform the rectangle using inner coordinates.
myRectangle.Transform idCoordinateSpaces.idInnerCoordinates,
idAnchorPoint.idCenterAnchor, myTransformationMatrix
Rem Select the rectangle and display an alert.
myInDesign.Select myRectangle
MsgBox "Transformed by inner coordinates."
Rem Undo the transformation.
myInDesign.Documents.Item(1).Undo
Rem Transform using parent coordinates.
myRectangle.Transform idCoordinateSpaces.idParentCoordinates,
idAnchorPoint.idCenterAnchor, myTransformationMatrix
myInDesign.Select myRectangle
MsgBox "Transformed by parent coordinates."
myInDesign.Documents.Item(1).Undo
Rem Transform using pasteboard coordinates.
myRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates,
idAnchorPoint.idCenterAnchor, myTransformationMatrix
myInDesign.Select myRectangle
MsgBox "Transformed by pasteboard coordinates."
myInDesign.Documents.Item(1).Undo
```

Transformation origin

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

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

    ```
    AnchorPoint.CenterAnchor
    ```

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

    ```
    Array(idAnchorPoint.idBottomLeftAnchor,
    idBoundingBoxLimits.idOuterStrokeBounds)
    ```

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

    ```
    Array(idAnchorPoint.idBottomLeftAnchor,
    idBoundingBoxLimits.idOuterStrokeBounds,
    idCoordinateSpaces.idPasteboardCoordinates)
    ```

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

    ```
    Array(Array(.5, .5), idBoundingBoxLimits.idOuterStrokeBounds)
    ```

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

    ```
    Array(Array(.5, .5), idBoundingBoxLimits.idOuterStrokeBounds,
    idCoordinateSpaces.idPasteboardCoordinates)
    ```

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

    ```
    Array(Array(72, 144), 1)
    ```

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

    ```
    Array(Array(72, 144), idAnchorPoint.idCenterAnchor)
    ```

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

    ```
    Array(72, 72)
    ```

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

```
Array(Array(72, 72), idCoordinateSpaces.idParentCoordinates
```

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

```
Array(Array(72, 72))
```

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

```
Set myPage = myInDesign.Documents.Item(1).Pages.Item(1)
Set myRectangle = myPage.Rectangles.Item(1)
myString = "Watch as we rotate the rectangle using different anchor points," & vbCr
myString = myString & "bounds types, and coordinate spaces." & vbCr & vbCr
myString = myString & "(You might need to move the alert aside" & vbCr
myString = myString & "to see the effect of the transformations.)"
MsgBox myString
Set myNewRectangle = myRectangle.Duplicate
Rem Create a transformation matrix with counterclockwise rotation angle = 30.
Set myTransformationMatrix = myInDesign.TransformationMatrices.Add(, , , 30)
Rem Rotate around the duplicated rectangle's center point.
myNewRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates,
idAnchorPoint.idCenterAnchor, myTransformationMatrix
Rem Select the rectangle and display an alert.
myInDesign.Select myNewRectangle
MsgBox "Transformed around center anchor."
Rem Undo the transformation.
myInDesign.Documents.Item(1).Undo
Rem Rotate the rectangle around the ruler location [-100, -100].
Rem Note that the anchor point specified here specifes the page
Rem containing the point--*not* that transformation point itself.
Rem The transformation gets the ruler coordinate [-100, -100] based
Rem on that page. Setting the considerRulerUnits parameter to true
Rem makes certain that the transformation uses the current
Rem ruler units.
myNewRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates,
Array(Array(-100, -100), idAnchorPoint.idTopLeftAnchor), myTransformationMatrix, ,
True
Rem Move the page guides to reflect the transformation point.
myPage.Guides.Item(1).Location = -100
myPage.Guides.Item(2).Location = -100
Rem Select the rectangle and display an alert.
myInDesign.Select myNewRectangle
MsgBox "Rotated around -100x, -100y."
Rem Undo the transformation and the guide moves.
myInDesign. Documents.Item(1).Undo
myInDesign. Documents.Item(1).Undo
myInDesign. Documents.Item(1).Undo
myNewRectangle.StrokeWeight = 12
myBounds = myNewRectangle.visibleBounds
Rem Move the page guides to reflect the transformation point.
myPage.Guides.Item(1).Location = myBounds(1)
myPage.Guides.Item(2).Location = myBounds(2)
Rem Rotate the rectangle around the outer edge of the lower left corner.
myNewRectangle.Transform idCoordinateSpaces.idPasteboardCoordinates,
Array(idAnchorPoint.idBottomLeftAnchor, idBoundingBoxLimits.idOuterStrokeBounds),
myTransformationMatrix, idMatrixContent.idRotationValue, True
```

```
Rem Select the rectangle and display an alert.
myInDesign.Select myNewRectangle
MsgBox "Rotated around the outside edge of the bottom left corner."
Rem Undo the transformation and the guide moves.
myInDesign. Documents.Item(1).Undo
myInDesign. Documents.Item(1).Undo
myInDesign. Documents.Item(1).Undo
myInDesign. Documents.Item(1).Undo
```

Resolving locations

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

```
Set myPage = myInDesign.Documents.Item(1).Pages.Item(1)
Set myRectangle = myPage.Groups.Item(1).Rectangles.Item(1)
Rem Template for resolve():
Rem PageItem.resolve (Location:any, in: CoordinateSpaces,
Rem ConsideringRulerUnits:boolean)
Rem Get a ruler coordinate in pasteboard coordinates.
Rem The following should work, but, due to a bug in InDesign CS4,
Rem it does not work in VBScript. It does work in VB6.
'myPageLocation = myRectangle.Resolve(Array(Array(72, 72),
idAnchorPoint.idTopRightAnchor),idCoordinateSpaces.idPasteboardCoordinates, True)
Rem resolve() returns an array (in this case, the array containins a single item).
'myPageLocation = myPageLocation(0)
'myPageLocationX = myPageLocation(0)
'myPageLocationY = myPageLocation(1)
'MsgBox "X: " & CStr(myPageLocation(0)) & vbCr & "Y: " & CStr(myPageLocation(1))
Rem Because of the above bug, here's a workaround using JavaScript.
myString = "var myRectangle =
app.documents.item(0).pages.item(0).groups.item(0).rectangles.item(0);" & vbCr
myString = myString & "var myPageLocation = myRectangle.resolve([[72, 72],
AnchorPoint.topRightAnchor], CoordinateSpaces.pasteboardCoordinates, true);" & vbCr
myString = myString & "alert(""X: "" + myPageLocation[0][0] + ""\rY: "" +
myPageLocation[0][1])"
myInDesign.DoScript myString, idScriptLanguage.idJavaScript
```

Transforming points

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

```
Rem General purpose routine for drawing regular polygons from their center point.
Function myDrawPolygon(myInDesign, myParent, myCenterPoint, myNumberOfPoints,
myRadius, myStarPolygon, myStarInset)
    ReDim myPathPoints(0)
    myPoint = Array(0, 0)
    If myStarPolygon = True Then
        myNumberOfPoints = myNumberOfPoints * 2
    End If
    myInnerRadius = myRadius * myStarInset
    myAngle = 360 / myNumberOfPoints
    Set myRotateMatrix = myInDesign.TransformationMatrices.Add(, , , myAngle)
    Set myOuterTranslateMatrix = myInDesign.TransformationMatrices.Add
    (, , , , myRadius)
    Set myInnerTranslateMatrix = myInDesign.TransformationMatrices.Add
    (, , , , myInnerRadius)
    For myPointCounter = 0 To myNumberOfPoints
        Rem Translate the point to the inner/outer radius.
        If ((myStarInset = 1) Or (myIsEven(myPointCounter) = True)) Then
            myTransformedPoint = myOuterTranslateMatrix.ChangeCoordinates(myPoint)
        Else
            myTransformedPoint = myInnerTranslateMatrix.ChangeCoordinates(myPoint)
        End If
        myTransformedPoint = myRotateMatrix.ChangeCoordinates(myTransformedPoint)
        myPathPoints = myPush(myPathPoints, myTransformedPoint)
        Set myRotateMatrix = myRotateMatrix.RotateMatrix(myAngle)
    Next
    Rem Create a new polygon.
    Set myPolygon = myParent.Polygons.Add
    Rem Set the entire path of the polygon to the array we've created.
    myPolygon.Paths.Item(1).EntirePath = myPathPoints
    Rem If the center point is somewhere other than [0,0],
    Rem translate the polygon to the center point.
    If (myCenterPoint(0) <> 0) Or (myCenterPoint(1) <> 0) Then
        Set myTranslateMatrix = myInDesign.TransformationMatrices.Add
        (, , , , myCenterPoint(0), myCenterPoint(1))
        myPolygon.Transform idCoordinateSpaces.idPasteboardCoordinates,
        idAnchorPoint.idCenterAnchor, myTranslateMatrix
    End If
    Set myDrawPolygon = myPolygon
End Function
Rem Generic function for adding a value to an array.
Function myPush(myArray, myValue)
    If Not (IsEmpty(myArray(0))) Then
        ReDim Preserve myArray(UBound(myArray) + 1)
    End If
    Set myArray(UBound(myArray)) = myValue
    myPush = myArray
End Function
Rem This function returns true if myNumber is even, false if it is not.
Function myIsEven(myNumber)
    myResult = myNumber Mod 2
    If myResult = 0 Then
        myResult = True
    Else
        myResult = False
    End If
    myIsEven = myResult
End Function
```

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

Transforming again

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

- `TransformAgain`
- `TransformAgainIndividually`
- `TransformSequenceAgain`
- `TransformSequenceAgainIndividually`

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

```
set myRectangle = myPage.Rectangles.Item(1)
myBounds = myRectangle.GeometricBounds
myX1 = myBounds(1)
myY1 = myBounds(0)
Set myRectangleA = myPage.Rectangles.Add
myRectangleA.GeometricBounds = Array(myY1-12, myX1-12, myY1+12, myX1+12)
Set myTransformationMatrix = myInDesign.TransformationMatrices.add(, , , 45)
myRectangleA.Transform idCoordinateSpaces.idPasteboardCoordinates,
idAnchorPoint.idCenterAnchor, myTransformationMatrix
Set myRectangleB = myRectangleA.Duplicate
myRectangleB.Transform idCoordinateSpaces.idPasteboardCoordinates, Array(Array(0,0),
idAnchorPoint.idTopLeftAnchor), myTransformationMatrix, , True
Set myRectangleC = myRectangleB.Duplicate
myRectangleC.TransformAgain
Set myRectangleD = myRectangleC.Duplicate
myRectangleD.transformAgain
Set myRectangleE = myRectangleD.Duplicate
myRectangleE.TransformAgain
set myRectangleF = myRectangleE.Duplicate
myRectangleF.TransformAgain
set myRectangleG = myRectangleF.Duplicate
myRectangleG.TransformAgain
set myRectangleH = myRectangleG.Duplicate
myRectangleH.TransformAgain
myRectangleA.Transform idCoordinateSpaces.idPasteboardCoordinates,
idAnchorPoint.idCenterAnchor, myTransformationMatrix
myRectangleD.TransformAgain
myRectangleF.TransformAgain
myRectangleH.TransformAgain
```

Resize and Reframe

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

```
Rem Given a reference to a rectangle "myRectangle"...
Set myDuplicate = myRectangle.Duplicate
myDuplicate.Resize idCoordinateSpaces.idInnerCoordinates,
idAnchorPoint.idCenterAnchor, idResizeMethods.idMultiplyingCurrentDimensionsBy,
Array(2, 2)
```

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

```
Rem Given a reference to a rectangle "myRectangle"...
myBounds = myRectangle.GeometricBounds
myX1 = myBounds(1)-72
myY1 = myBounds(0)-72
myX2 = myBounds(3)+72
myY2 = myBounds(2)+72
Set myDuplicate = myRectangle.Duplicate
myDuplicate.Reframe idCoordinateSpaces.idInnerCoordinates, Array(Array(myY1, myX1),
Array(myY2, myX2))
```

5 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 CS4 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 the MakeTextFrame tutorial script):

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrame = myPage.TextFrames.Add
Rem Set the bounds of the text frame.
myTextFrame.GeometricBounds = Array(72, 72, 288, 288)
Rem 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.)

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Rem Create a text frame on the current page.
Set myTextFrame = myPage.TextFrames.Add
Rem Set the bounds of the text frame.
myTextFrame.GeometricBounds = myGetBounds(myDocument, myDocument.Pages.Item(1))
Rem Enter text in the text frame.
myTextFrame.Contents = "This is some example text."
```

The following script fragment shows the myGetBounds function.

```
Function myGetBounds(myDocument, myPage)
    myPageWidth = myDocument.DocumentPreferences.PageWidth
    myPageHeight = myDocument.DocumentPreferences.PageHeight
    If myPage.Side = idPageSideOptions.idLeftHand Then
        myX2 = myPage.MarginPreferences.Left
        myX1 = myPage.MarginPreferences.Right
    Else
        myX1 = myPage.MarginPreferences.Left
        myX2 = myPage.MarginPreferences.Right
    End If
    myY1 = myPage.marginPreferences.Top
    myX2 = myPageWidth - myX2
    myY2 = myPageHeight - myPage.MarginPreferences.Bottom
    myGetBounds = Array(myY1, myX1, myY2, myX2)
End Function
```

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):

```
set myDocument = myInDesign.Documents.Item(1)
Set myTextFrame = myDocument.TextFrames.Item(1)
Rem Add text at the end of the text in the text frame.
Rem To do this, we'll use the last insertion point in the story.
Rem (vbCr is a return character, "&" concatenates two strings.)
myNewText = "This is a new paragraph of example text."
myTextFrame.ParentStory.InsertionPoints.Item(-1).Contents = vbCr & myNewText
```

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).

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrame = myPage.TextFrames.Item(1)
Rem Now add text beyond the end of the text frame.
myTextFrame.InsertionPoints.Item(-1).Contents = vbCr & "This is some overset text"
myString = myTextFrame.Contents
MsgBox ("The last paragraph in this alert should be ""This is some overset text"". Is
it?" & vbCr & myString)
myString = myTextFrame.ParentStory.Contents
MsgBox ("The last paragraph in this alert should be ""This is some overset text"". Is
it?" & vbCr & myString)
```

For more on understanding the relationships between text objects in an InDesign document, see "Understanding text objects" on page 74.

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):

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrame = myPage.TextFrames.Item(1)
Rem Replace the third word with the phrase
Rem "a little bit of".
myTextFrame.ParentStory.Words.Item(3).contents = "a little bit of"
```

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

```
Rem Replace the text in the second paragraph without replacing
Rem the return character at the end of the paragraph. To do this,
Rem we'll use the ItemByRange method.
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrame = myPage.TextFrames.Item(1)
Set myStartCharacter = myTextFrame.ParentStory.Paragraphs.Item(2).Characters.Item(1)
Set myEndCharacter = myTextFrame.ParentStory.Paragraphs.Item(2).Characters.Item(-2)
myTextFrame.Texts.ItemByRange(myStartCharacter, myEndCharacter).Item(1).Contents =
"This text replaces the text in paragraph 2."
```

In the script above, we used the `ItemByRange` method to get a reference to the text of the paragraph (excluding the return character at the end of the paragraph), as a single text object. We excluded the return character because deleting the return might change the paragraph style applied to the paragraph. To use the `ItemByRange` method, we used the texts collection of the story, but we supplied two characters—the starting and ending characters of the paragraph—as parameters. If we used `myTextFrame.ParentStory.Characters.ItemByRange`, InDesign would return a collection of `Character` objects. We wanted one `Text` object, so we could replace the contents in one action.

Inserting special characters

Because most VBScript editors support Unicode, you can simply enter Unicode characters in text strings you send to InDesign. The following script shows several ways to enter special characters. (We omitted the `myGetBounds` function from this listing; you can find it in "Creating a text frame" on page 63" or in the SpecialCharacters tutorial script.)

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrame = myPage.TextFrames.Item(1)
Rem Entering special characters directly.
myTextFrame.contents = "Registered trademark: Æ" & vbCr & "Copyright: ©" & vbCr &
"Trademark: ?" & vbCr & ""
Rem Entering special characters by their Unicode glyph ID
Rem value ("&H" indicates a hexadecimal number):
myTextFrame.ParentStory.InsertionPoints.Item(-1).contents = "Not equal to: " &
ChrW(&H2260) & vbCr & "Square root: " & ChrW(&H221A) & vbCr & "Paragraph: " & ChrW(&HB6)
& vbCr
Rem Entering InDesign special characters by their enumerations:
myTextFrame.ParentStory.InsertionPoints.Item(-1).contents = "Page number marker:"
myTextFrame.ParentStory.InsertionPoints.Item(-1).contents =
idSpecialCharacters.idAutoPageNumber
myTextFrame.ParentStory.InsertionPoints.Item(-1).contents = vbCr
myTextFrame.ParentStory.InsertionPoints.Item(-1).contents = "Section symbol:"
myTextFrame.ParentStory.InsertionPoints.Item(-1).contents =
idSpecialCharacters.idSectionSymbol
myTextFrame.ParentStory.InsertionPoints.Item(-1).contents = vbCr
myTextFrame.ParentStory.InsertionPoints.Item(-1).contents = "En dash:"
myTextFrame.ParentStory.InsertionPoints.Item(-1).contents =
idSpecialCharacters.idEnDash
myTextFrame.ParentStory.InsertionPoints.Item(-1).contents = vbCr
```

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):

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
myX = myPage.MarginPreferences.Left
myY = myPage.MarginPreferences.Top
Rem Autoflow a text file on the current page.
Rem Parameters for Page.place():
Rem File as File object,
Rem [PlacePoint as Array [x, y]]
Rem [DestinationLayer as Layer object]
Rem [ShowingOptions as Boolean = False]
Rem [Autoflowing as Boolean = False]
Rem You'll have to fill in your own file path.
Set myTextFrame = myPage.Place("c:\test.txt", Array(myX, myY), , False, True)
Rem Note that if the PlacePoint parameter is inside a file, only the vertical (y)
Rem coordinate will be honored--the text frame will expand horizontally
Rem 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 63," or see the PlaceTextFileInFrame tutorial script.)

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrame = myPage.TextFrames.Item(1)
Rem Place a text file in the text frame.
Rem Parameters for TextFrame.place():
Rem File as string,
Rem [ShowingOptions as Boolean = False]
Rem You'll have to fill in your own file path.
myTextFrame.Place "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 63," or see the InsertTextFile tutorial script.)

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Rem Create a text frame.
Set myTextFrame = myPage.TextFrames.Add
myTextFrame.geometricBounds = myGetBounds(myDocument, myPage)
myTextFrame.Contents = "Inserted text file follows:" & vbCr
Rem Place a text file at the end of the text.
Rem Parameters for InsertionPoint.place():
Rem File as string (file path),
Rem [ShowingOptions as Boolean = False]
Rem You'll have to fill in your own file path.
myTextFrame.ParentStory.InsertionPoints.Item(-1).Place "c:\test.txt"
```

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

```
With myInDesign.TextImportPreferences
    Rem Options for characterSet:
    Rem idTextImportCharacterSet.idAnsi
    Rem idTextImportCharacterSet.idChineseBig5
    Rem idTextImportCharacterSet.idGB18030
    Rem idTextImportCharacterSet.idGB2312
    Rem idTextImportCharacterSet.idKSC5601
    Rem idTextImportCharacterSet.idMacintoshCE
    Rem idTextImportCharacterSet.idMacintoshCyrillic
    Rem idTextImportCharacterSet.idMacintoshGreek
    Rem idTextImportCharacterSet.idMacintoshTurkish
    Rem idTextImportCharacterSet.idRecommendShiftJIS83pv
    Rem idTextImportCharacterSet.idShiftJIS90ms
    Rem idTextImportCharacterSet.idshiftJIS90pv
    Rem idTextImportCharacterSet.idUnicode
    Rem idTextImportCharacterSet.idWindowsBaltic
    Rem idTextImportCharacterSet.idWindowsCE
    Rem idTextImportCharacterSet.idWindowsCyrillic
    Rem idTextImportCharacterSet.idWindowsEE
    Rem icTextImportCharacterSet.idWindowsGreek
    Rem idTextImportCharacterSet.idWindowsTurkish
    .CharacterSet = idTextImportCharacterSet.idUnicode
    .ConvertSpacesIntoTabs = True
    .SpacesIntoTabsCount = 3
    Rem The dictionary property can take any of the following
    Rem language names (as strings):
    Rem Bulgarian
    Rem Catalan
```

```
    Rem Croatian
    Rem Czech
    Rem Danish
    Rem Dutch
    Rem English:Canadian
    Rem English: UK
    Rem English: USA
    Rem English: USA Legal
    Rem English: USA Medical
    Rem Estonian
    Rem Finnish
    Rem French
    Rem French: Canadian
    Rem German: Reformed
    Rem German: Swiss
    Rem German: Traditional
    Rem Greek
    Rem Hungarian
    Rem Italian
    Rem Latvian
    Rem Lithuanian
    Rem Neutral
    Rem Norwegian: Bokmal
    Rem Norwegian: Nynorsk
    Rem Polish
    Rem Portuguese
    Rem Portuguese: Brazilian
    Rem Romanian
    Rem Russian
    Rem Slovak
    Rem Slovenian
    Rem Spanish: Castilian
    Rem Swedish
    Rem Turkish
    .Dictionary = "English:USA"
    Rem platform options:
    Rem idImportPlatform.idMacintosh
    Rem idImportPlatform.idPC
    .Platform = idImportPlatform.idPC
    .StripReturnsBetweenLines = True
    .StripReturnsBetweenParagraphs = True
    .UseTypographersQuotes = True
End With
```

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

```
With myInDesign.TaggedTextImportPreferences
    .RemoveTextFormatting = False
    Rem .styleConflict property can be:
    Rem idStyleConflict.idPublicationDefinition
    Rem idStyleConflict.idTagFileDefinition
    .StyleConflict = idStyleConflict.idPublicationDefinition
    .UseTypographersQuotes = True
End With
```

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

```
With myInDesign.WordRTFImportPreferences
    Rem convertPageBreaks property can be:
    Rem idConvertPageBreaks.idColumnBreak
    Rem idConvertPageBreaks.idNone
    Rem idConvertPageBreaks.idPageBreak
    .ConvertPageBreaks = idConvertPageBreaks.idNone
    Rem convertTablesTo property can be:
    Rem idConvertTablesOptions.idUnformattedTabbedText
    Rem idConvertTablesOptions.idUnformattedTable
    .ConvertTablesTo = idConvertTablesOptions.idUnformattedTable
    .ImportEndnotes = True
    .ImportFootnotes = True
    .ImportIndex = True
    .ImportTOC = True
    .ImportUnusedStyles = False
    .PreserveGraphics = False
    .PreserveLocalOverrides = False
    .PreserveTrackChanges = False
    .RemoveFormatting = False
    Rem resolveCharacterStyleClash and resolveParagraphStyleClash properties can be:
    Rem idResolveStyleClash.idResolveClashAutoRename
    Rem idResolveStyleClash.iduseExisting
    Rem idResolveStyleClash.iduseNew
    .ResolveCharacterStyleClash = idResolveStyleClash.iduseExisting
    .ResolveParagraphStyleClash = idResolveStyleClash.iduseExisting
    .UseTypographersQuotes = True
End With
```

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

```
With myInDesign.ExcelImportPreferences
    Rem alignmentStyle property can be:
    Rem AlignmentStyleOptions.centerAlign
    Rem AlignmentStyleOptions.leftAlign
    Rem AlignmentStyleOptions.rightAlign
    Rem AlignmentStyleOptions.spreadsheet
    .AlignmentStyle = idAlignmentStyleOptions.idSpreadsheet
    .DecimalPlaces = 4
    .PreserveGraphics = False
    Rem Enter the range you want to import as "start cell:end cell".
    .RangeName = "A1:B16"
    .SheetIndex = 1
    .SheetName = "pathpoints"
    .ShowHiddenCells = False
    Rem tableFormatting property can be:
    Rem idTableFormattingOptions.idExcelFormattedTable
    Rem idTableFormattingOptions.idExcelUnformattedTabbedText
    Rem idTableFormattingOptions.idExcelUnformattedTable
    .TableFormatting = idTableFormattingOptions.idExcelFormattedTable
    .UseTypographersQuotes = True
    .ViewName = ""
End With
```

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 63," or see the ExportTextFile tutorial script.)

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrame = myPage.TextFrames.Item(1)
Rem Text export method parameters:
Rem Format as idExportFormat
Rem To As File
Rem [ShowingOptions As Boolean = False]
Rem
Rem Format parameter can be:
Rem idExportFormat.idInCopy
Rem idExportFormat.idInCopyCS2Story
Rem idExportFormat.idRTF
Rem idExportFormat.idTaggedText
Rem idExportFormat.idTextType
Rem
Rem Export the story as text. You'll have to fill in a valid file path on your system.
myTextFrame.ParentStory.Export idExportFormat.idTextType, "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 63," or see the ExportTextRange tutorial script.)

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrame = myPage.TextFrames.Item(1)
Set myStory = myTextFrame.ParentStory
Set myStartCharacter = myStory.Paragraphs.Item(1).Characters.Item(1)
Set myEndCharacter = myStory.Paragraphs.Item(1).Characters.Item(-1)
Set myText = myTextFrame.ParentStory.Texts.ItemByRange(myStartCharacter,
myEndCharacter).Item(1)
Rem Text export method parameters:
Rem Format as idExportFormat
Rem To As File
Rem [ShowingOptions As Boolean = False]
Rem
Rem Format parameter can be:
Rem idExportFormat.idInCopy
Rem idExportFormat.idInCopyCS2Story
Rem idExportFormat.idRTF
Rem idExportFormat.idTaggedText
Rem idExportFormat.idTextType
Rem
Rem Export the text range. You'll have to fill in a valid file path on your system.
myText.Export idExportFormat.idTextType, "C:\test.txt"
```

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

```
With myInDesign.TextExportPreferences
    Rem Options for characterSet:
    Rem idTextExportCharacterSet.idUnicode
    Rem idTextExportCharacterSet.idDefaultPlatform
    .CharacterSet = idTextExportCharacterSet.idUnicode
    Rem platform options:
    Rem idImportPlatform.idMacintosh
    Rem idImportPlatform.idPC
    .Platform = idImportPlatform.idPC
End With
```

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

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
With myInDesign.TaggedTextExportPreferences
    Rem Options for characterSet:
    Rem idTagTextExportCharacterSet.idAnsi
    Rem idTagTextExportCharacterSet.idASCII
    Rem idTagTextExportCharacterSet.idGB18030
    Rem idTagTextExportCharacterSet.idKSC5601
    Rem idTagTextExportCharacterSet.idShiftJIS
    Rem idTagTextExportCharacterSet.idUnicode
    .CharacterSet = idTagTextExportCharacterSet.idUnicode
    Rem tagForm options:
    Rem idTagTextForm.idAbbreviated
    Rem idTagTextForm.idVerbose
    .TagForm = idTagTextForm.idVerbose
End With
```

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 63," or see the ExportAllText tutorial script.) For any format other than text only, the latter method can become quite complex.

```
If myInDesign.Documents.Count <> 0 Then
    If myInDesign.Documents.Item(1).Stories.Count <> 0 Then
        myExportAllText myInDesign, myInDesign.Documents.Item(1).Name
    End If
End If
```

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

```
Function myExportAllText(myInDesign, myDocumentName)
    mySeparatorString = "---------------------------------------" & vbCr
    Rem If you want to add a separator line between stories,
    Rem set myAddSeparator to true.
    myAddSeparator = True
    Set myNewDocument = myInDesign.Documents.Add
    Set myDocument = myInDesign.Documents.Item(myDocumentName)
    Set myTextFrame = myNewDocument.Pages.Item(1).TextFrames.Add
    myTextFrame.geometricBounds = myGetBounds(myNewDocument,
    myNewDocument.Pages.Item(1))
    Set myNewStory = myTextFrame.ParentStory
    For myCounter = 1 To myDocument.Stories.Count
    Set myStory = myDocument.Stories.Item(myCounter)
    myStory.texts.item(1).duplicate idLocationOptions.idAfter,
    myNewStory.InsertionPoints.Item(1)
    Rem If the text did not end with a return, enter a return
    Rem to keep the stories from running together.
    If myCounter <> myDocument.Stories.Count Then
        If myNewStory.Characters.Item(-1).Contents <> vbCr Then
            myNewStory.InsertionPoints.Item(-1).Contents = vbCr
        End If
        If myAddSeparator = True Then
            myNewStory.InsertionPoints.Item(-1).Contents = mySeparatorString
        End If
    End If
    Next
    myNewStory.Export idExportFormat.idTaggedText, "c:\test.txt"
    myNewDocument.Close idSaveOptions.idNo
End Function
```

Do not assume you are limited to exporting text using existing export filters. Since VBScript 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 63," or see the ExportHTML tutorial script.)

```
Set myDocument = myInDesign.Documents.Item(1)
Rem Use the myStyleToTagMapping dictionary to set
Rem up your paragraph style to tag mapping.
Set myStyleToTagMapping = CreateObject("Scripting.Dictionary")
Rem For each style to tag mapping, add a new item to the dictionary.
myStyleToTagMapping.Add "body_text", "p"
myStyleToTagMapping.Add "heading1", "h1"
myStyleToTagMapping.Add "heading2", "h2"
myStyleToTagMapping.Add "heading3", "h3"
Rem End of style to tag mapping.
If myDocument.Stories.Count <> 0 Then
    Rem Open a new text file.
    Set myDialog = CreateObject("UserAccounts.CommonDialog")
    myDialog.Filter = "HTML Files|*.html|All Files|*.*"
    myDialog.FilterIndex = 1
    myDialog.InitialDir = "C:\"
    myResult = myDialog.ShowOpen
    Rem If the user clicked the Cancel button, the result is null.
    If myResult = True Then
        myTextFileName = myDialog.FileName
        Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
        Set myTextFile = myFileSystemObject.CreateTextFile(myTextFileName)
```

```
For myCounter = 1 To myInDesign.Documents.Item(1).Stories.Count
    Set myStory = myDocument.Stories.Item(myCounter)
    For myParagraphCounter = 1 To myStory.Paragraphs.Count
        Set myParagraph = myStory.Paragraphs.Item(myParagraphCounter)
        If myParagraph.Tables.Count = 0 Then
            If myParagraph.TextStyleRanges.Count = 1 Then
                Rem If the paragraph is a simple paragraph--no tables,
                Rem no local formatting--then simply export the text
                Rem of the pararaph with the appropriate tag.
                myTag = myStyleToTagMapping.Item(myParagraph.
                AppliedParagraphStyle.Name)
                Rem If the tag comes back empty, map it to the basic paragraph tag.
                If myTag = "" Then
                    myTag = "p"
                End If
                myStartTag = "<" & myTag & ">"
                myEndTag = "</" & myTag & ">"
                Rem If the paragraph is not the last paragraph in the story,
                Rem omit the return character.
                If myParagraph.Characters.Item(-1).Contents = vbCr Then
                    myString = myParagraph.Texts.ItemByRange(myParagraph.
                    Characters.Item(1), myParagraph.Characters.Item(-2)).
                    Item(1).Contents
                Else
                    myString = myParagraph.Contents
                End If
                Rem Write the paragraphs' text to the text file.
                myTextFile.WriteLine myStartTag & myString & myEndTag
            Else
                Rem Handle text style range export by iterating
                Rem through the text style ranges in the paragraph..
                For myRangeCounter = 1 To myParagraph.TextStyleRanges.Length
                    myTextStyleRange = myParagraph.TextStyleRanges.Item
                    (myRangeCounter)
                    If myTextStyleRange.Characters.Item(-1) = vbCr Then
                        myString = myTextStyleRange.Texts.ItemByRange
                        (myTextStyleRange.Characters.Item(1),
                        myTextStyleRange.Characters.Item(-2)).Item(1).Contents
                    Else
                        myString = myTextStyleRange.Contents
                    End If
                    Select Case myTextStyleRange.FontStyle
                        Case "Bold":
                            myString = "<b>" & myString & "</b>"
                        Case "Italic":
                            myString = "<i>" & myString & "</i>"
                    End Select
                    myTextFile.write myString
                Next
                myTextFile.write vbCr
            End If
        Else
            Rem Handle table export (assumes that there is
            Rem only one table per paragraph,
            Rem and that the table is in the paragraph by itself).
            Set myTable = myParagraph.Tables.Item(1)
            myTextFile.write "<table border = 1>"
```

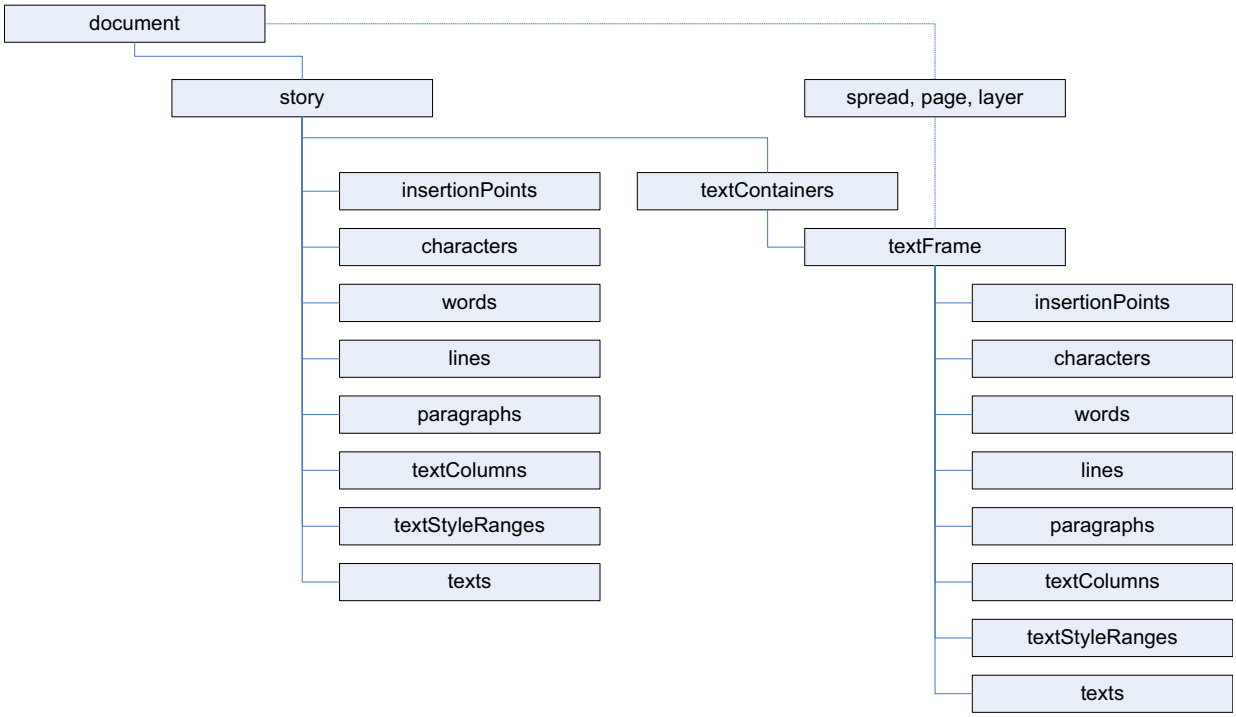
```
                    For myRowCounter = 1 To myTable.Rows.Count
                        myTextFile.write "<tr>"
                        For myColumnCounter = 1 To myTable.Columns.Count
                            If myRowCounter = 1 Then
                                myString = "<th>" & myTable.Rows.Item(myRowCounter).
                                Cells.Item(myColumnCounter).Texts.Item(1).Contents &
                                "</th>"
                            Else
                                myString = "<td>" & myTable.Rows.Item(myRowCounter).
                                Cells.Item(myColumnCounter).Texts.Item(1).Contents &
                                "</td>"
                            End If
                            myTextFile.write myString
                        Next
                        myTextFile.WriteLine "</tr>"
                    Next
                    myTextFile.WriteLine "</table>"
                End If
            Next
            Rem Close the text file.
            myTextFile.Close
        Next
    End If
End If
```

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

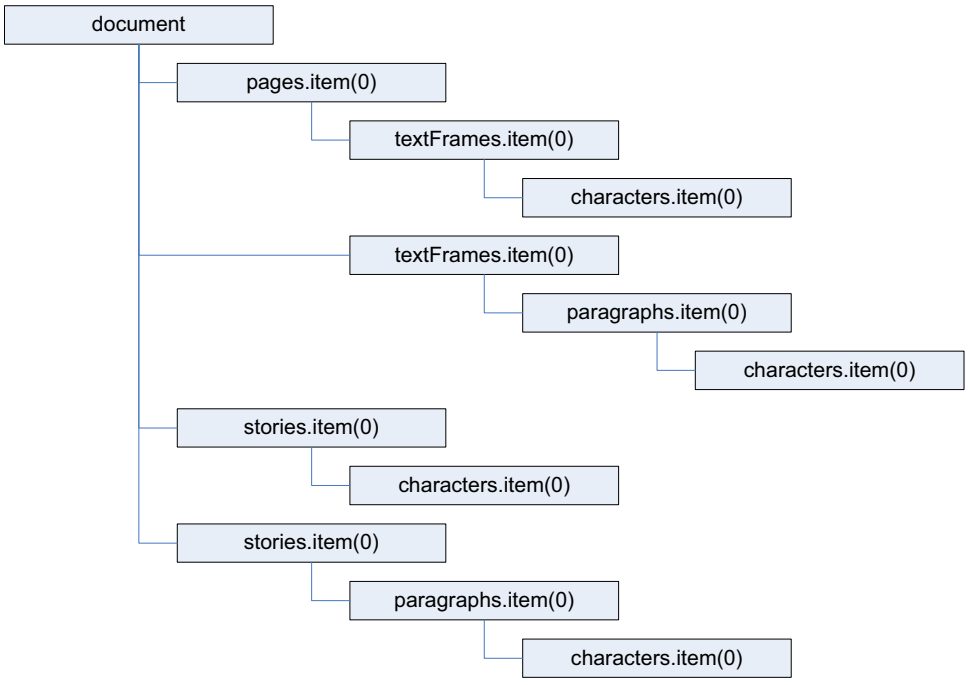
```
function myFindTag (myStyleName, myStyleToTagMapping){
    var myTag = "";
    var myDone = false;
    var myCounter = 0;
    do{
        if(myStyleToTagMapping[myCounter][0] == myStyleName){
            myTag = myStyleToTagMapping[myCounter][1];
            break;
        }
        myCounter ++;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

Understanding text objects

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



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



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

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

containing page item. If the text frame was converted to an anchored frame, the `parent` of the text frame is the character containing the anchored frame.

Working with text selections

Text-related scripts often act on a text selection. The following script demonstrates a way to 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).

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.Documents.Count <> 0 Then
    Rem If the selection contains more than one item, the selection
    Rem is not text selected with the Type tool.
    If myInDesign.Selection.Count = 1 Then
        Select Case TypeName(myInDesign.Selection.Item(1))
            Case "InsertionPoint", "Character", "Word", "TextStyleRange",
            "Line", "Paragraph", "TextColumn", "Text"
                MsgBox "The selection is a text object."
                Rem A real script would now act on the text object
                Rem or pass it on to a function.
            Case "TextFrame"
                Rem In addition to checking for the above text objects, we can
                Rem also continue if the selection is a text frame selected with
                Rem the Selection tool or the Direct Selection tool.
                Rem If the selection is a text frame, you get a reference to the
                Rem text in the text frame and then pass it along to a function.
                Rem Set myText = myInDesign.Selection.Item(1).Texts.Item(1)
                MsgBox "The selected object is a text frame."
            Case Else
                MsgBox "The selected object is not a text object.
                Select some text and try again."
        End Select
    Else
        MsgBox "Please select some text and try again."
    End If
Else
    MsgBox "No documents are open. Please open a document,
    select some text, and try again."
End If
```

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):

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Rem Set the bounds live area of the page.
myBounds = myGetBounds(myDocument, myDocument.Pages.Item(1))
myX1 = myBounds(1)
myY1 = myBounds(0)
myX2 = myBounds(3)
myY2 = myBounds(2)
myWidth = myX2 - myX1
myHeight = myY2 - myY1
Rem Create a series of text frames.
Set myTextFrameA = myPage.TextFrames.Add
myTextFrameA.geometricBounds = Array(myY1, myX1, myY1 + (myHeight / 2), myX1 + (myWidth
/ 2))
myTextFrameA.Contents = "Before." & vbCr
Set myTextFrameB = myPage.TextFrames.Add
myTextFrameB.geometricBounds = Array(myY1, myX1 + (myWidth / 2), myY1 + (myHeight / 2),
myX2)
myTextFrameB.Contents = "After." & vbCr
Set myTextFrameC = myPage.TextFrames.Add
myTextFrameC.geometricBounds = Array(myY1 + (myHeight / 2), myX1, myY2, myX1 + (myWidth
/ 2))
myTextFrameC.Contents = "Between words." & vbCr
Set myTextFrameD = myPage.TextFrames.Add
myTextFrameD.geometricBounds = Array(myY1 + (myHeight / 2), myX1 + (myWidth / 2), myY2,
myX2)
myTextFrameD.Contents = "Text to move:" & vbCr & "WordA" & vbCr & "WordB" & vbCr &
"WordC" & vbCr
Rem Move WordC between the words in TextFrameC.
myTextFrameD.ParentStory.Paragraphs.Item(-1).Words.Item(1).Move
idLocationOptions.idBefore, myTextFrameC.ParentStory.Paragraphs.Item(1).Words.Item(2)
Rem Move WordB after the word in TextFrameB.
myTextFrameD.ParentStory.Paragraphs.Item(-2).Words.Item(1).Move
idLocationOptions.idAfter, myTextFrameB.ParentStory.Paragraphs.Item(1).Words.Item(1)
Rem Move WordA to before the word in TextFrameA.
myTextFrameD.ParentStory.Paragraphs.Item(-3).Words.Item(1).Move
idLocationOptions.idBefore, myTextFrameA.ParentStory.Paragraphs.Item(1).Words.Item(1)
Rem Note that moving text removes it from its original location.
```

When you want to transfer formatted text from one document to another, you also can use the `move` method. Using the `move` or `duplicate` method is better than using copy and paste; to use copy and paste, you must make the document visible and select the text you want to copy. Using `move` or `duplicate` is much faster and more robust. The following script shows how to move text from one document to another using `move` and `duplicate`. (We omitted the `myGetBounds` function from this listing; you can find it in "Creating a text frame" on page 63," or see the MoveTextBetweenDocuments tutorial script.)

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
set mySourceTextFrame = myPage.TextFrames.Item(1)
Set mySourceParagraph = mySourceTextFrame.ParentStory.Paragraphs.Item(1)
Rem Create a target.
Set myTargetDocument = myInDesign.Documents.Add
Set myTargetPage = myTargetDocument.Pages.Item(1)
Rem Create a text frame.
Set myTextFrame = myTargetPage.TextFrames.Add
myTextFrame.GeometricBounds = myGetBounds(myTargetDocument, myTargetPage)
myTextFrame.Contents = "This is the target text. Insert the source text after this
paragraph." & vbCr
mySourceParagraph.duplicate idLocationOptions.idAfter,
myTextFrame.InsertionPoints.Item(-1)
```

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 63," or see the CopyPasteText tutorial script.)

```
Set myDocumentA = myInDesign.Documents.Item(1)
Set myDocumentB = myInDesign.Documents.Add
Set myPageB = myDocumentB.Pages.Item(1)
Set myTextFrameB = myPageB.TextFrames.Add
myTextFrameB.GeometricBounds = myGetBounds(myDocumentB, myPageB)
Rem Make document A the active document.
myInDesign.ActiveDocument = myDocumentA
Rem Select the text.
myInDesign.Select myTextFrameA.ParentStory.Texts.Item(1)
myInDesign.Copy
Rem Make document B the active document.
myInDesign.ActiveDocument = myDocumentB
Rem Select the insertion point at which you want to paste the text.
myInDesign.Select myTextFrameB.ParentStory.InsertionPoints.Item(1)
myInDesign.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):

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrameA = myPage.TextFrames.Add
myTextFrameA.geometricBounds = Array(72, 72, 144, 288)
myTextFrameA.Contents = "This is a formatted string."
myTextFrameA.ParentStory.Texts.Item(1).FontStyle = "Bold"
Set myTextFrameB = myPage.TextFrames.Add
myTextFrameB.geometricBounds = Array(228, 72, 300, 288)
myTextFrameB.Contents = "This is the destination text frame. Text pasted here will
retain its formatting."
myTextFrameB.ParentStory.Texts.Item(1).FontStyle = "Italic"
Rem Copy from one frame to another using a simple copy.
myInDesign.Select myTextFrameA.Texts.Item(1)
myInDesign.Copy
myInDesign.Select myTextFrameB.ParentStory.InsertionPoints.Item(-1)
myInDesign.Paste
Rem Create another text frame on the active page.
Set myTextFrameC = myPage.TextFrames.Add
myTextFrameC.geometricBounds = Array(312, 72, 444, 288)
myTextFrameC.Contents = "Text copied here will take on the formatting of the existing
text."
myTextFrameC.ParentStory.Texts.Item(1).FontStyle = "Italic"
Rem Copy the unformatted string from text frame A to the end of text frame C (note
Rem that this doesn't really copy the text it replicates the text string from one
Rem text frame in another text frame):
myTextFrameC.ParentStory.InsertionPoints.Item(-1).Contents =
myTextFrameA.ParentStory.Texts.Item(1).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 63," or see the TextIterationWrong tutorial script.)

```
Set myDocument = myInDesign.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Rem The following for loop cause an error.
For myParagraphCounter = 1 to myStory.Paragraphs.Count
    If myStory.Paragraphs.Item(myParagraphCounter).Words.Item(1).contents = "Delete"
Then
        myStory.Paragraphs.Item(myParagraphCounter).Delete
    Else
        myStory.Paragraphs.Item(myParagraphCounter).PointSize = 24
    End If
Next
```

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 3, 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 63," or see the TextIterationRight tutorial script.)

```
Set myDocument = myInDesign.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Rem The following for loop will format all of the paragraphs by iterating
Rem backwards through the paragraphs in the story.
For myCounter = myStory.Paragraphs.Count To 1 Step -1
    If myStory.Paragraphs.Item(myCounter).Words.Item(1).contents = "Delete" Then
        myStory.Paragraphs.Item(myCounter).Delete
    Else
        myStory.Paragraphs.Item(myCounter).PointSize = 24
    End If
Next
```

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):

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrameA = myPage.TextFrames.Add
myTextFrameA.geometricBounds = Array(72, 72, 144, 144)
Set myTextFrameB = myPage.TextFrames.Add
myTextFrameB.geometricBounds = Array(228, 72, 300, 144)
Rem Add a page.
Set myNewPage = myDocument.Pages.Add
Rem Create another text frame on the new page.
Set myTextFrameC = myNewPage.TextFrames.Add
myTextFrameC.geometricBounds = Array(72, 72, 144, 144)
Rem Link TextFrameA to TextFrameB using the nextTextFrame property.
myTextFrameA.NextTextFrame = myTextFrameB
Rem Link TextFrameC to TextFrameB using the previousTextFrame property.
myTextFrameC.PreviousTextFrame = myTextFrameB
Rem Fill the text frames with placeholder text.
myTextFrameA.Contents = idTextFrameContents.idPlaceholderText
```

Unlinking text frames

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

```
Rem Given two linked text frames on page 1...
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrameA = myPage.TextFrames.Item(-1)
Rem Unlink text frame A.
myTextFrameA.NextTextFrame = 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):

```
ReDim myObjectList(0)
Rem Script does nothing if no documents are open or if no objects are selected.
If myInDesign.Documents.Count <> 0 Then
    If myInDesign.Selection.Count <> 0 Then
        Rem Process the objects in the selection to create a list of
        Rem qualifying objects (text frames).
        For myCounter = 1 To myInDesign.Selection.Count
            Select Case TypeName(myInDesign.Selection.Item(myCounter))
                Case "TextFrame":
                    If Not (IsEmpty(myObjectList(0))) Then
                        ReDim Preserve myObjectList(UBound(myObjectList) + 1)
                    End If
                    Set myObjectList(UBound(myObjectList)) =
                    myInDesign.Selection.Item(myCounter)
```

```
            Case "InsertionPoint", "Character", "Word", "TextStyleRange", "Line",
            "Paragraph", "TextColumn", "Text":
                If Not (IsEmpty(myObjectList(0))) Then
                    ReDim Preserve myObjectList(UBound(myObjectList) + 1)
                End If
                Set myObjectList(UBound(myObjectList)) =
                myInDesign.Selection.Item(myCounter).ParentTextFrames.Item(1)
        End Select
    Next
    Rem If the object list is not empty, pass it on to the function
    Rem that does the real work.
    If Not (IsEmpty(myObjectList(0))) Then
        myBreakFrames myObjectList
    End If
  End If
End If
```

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

```
Function myBreakFrames(myObjectList)
    For myCounter = UBound(myObjectList) To 0 Step -1
        myBreakOutFrame myObjectList(myCounter)
    Next
End Function
Function myBreakFrame(myTextFrame)
    myProcessFrame = vbYes
    If (TypeName(myTextFrame.NextTextFrame) <> "Nothing") Or
    (TypeName(myTextFrame.PreviousTextFrame) <> "Nothing") Then
        If myTextFrame.ParentStory.Tables.Count <> 0 Then
            myProcessFrame = MsgBox("This story contains tables. If the text frame you
            are trying to remove from the story contains a table, the results might not be
            what you expect. Do you want to continue?", vbYesNo)
        End If
        If myProcessFrame = vbYes Then
            Set myNewFrame = myTextFrame.Duplicate
            If myTextFrame.Contents <> "" Then
                myTextFrame.Texts.Item(1).Delete
            End If
            myTextFrame.Delete
        End If
    End If
End Function
```

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):

```
Rem Get the first item in the selection.
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrame = myPage.TextFrames.Item(1)
Set myStory = myTextFrame.ParentStory
Rem If the text frame is the only text frame in the story, do nothing.
If myStory.TextContainers.Count > 1 Then
    Rem Splitting the story is a two-step process: first, duplicate
    Rem the text frames, second, delete the original text frames.
    mySplitStory myStory
    myRemoveFrames myStory
End If
```

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

```
Function mySplitStory(myStory)
    Rem Duplicate each text frame in the story.
    For myCounter = myStory.TextContainers.Count To 1 Step -1
        Set myTextFrame = myStory.TextContainers.Item(myCounter)
        myTextFrame.Duplicate
    Next
End Function
Function myRemoveFrames(myStory)
    Rem Remove each text frame in the story.
    RemIterate backwards to avoid invalid references.
    For myCounter = myStory.TextContainers.Count To 1 Step -1
        myStory.TextContainers.Item(myCounter).Delete
    Next
End Function
```

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):

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
set myTextFrame = myPage.TextFrames.Item(1)
Set myInsertionPoint = myTextFrame.Paragraphs.Item(1).InsertionPoints.Item(1)
Set myInlineFrame = myInsertionPoint.TextFrames.Add
Rem Recompose the text to make sure that getting the
Rem geometric bounds of the inline graphic will work.
myTextFrame.Texts.Item(1).Recompose
Rem Get the geometric bounds of the inline frame.
myBounds = myInlineFrame.GeometricBounds
Rem Set the width and height of the inline frame. In this example, we'll
Rem make the frame 24 points tall by 72 points wide.
myArray = Array(myBounds(0), myBounds(1), myBounds(0) + 24, myBounds(1) + 72)
myInlineFrame.GeometricBounds = myArray
myInlineFrame.Contents = "This is an inline frame."
Set myInsertionPoint = myTextFrame.Paragraphs.Item(2).InsertionPoints.Item(1)
Set myAnchoredFrame = myInsertionPoint.TextFrames.Add
```

```
Rem Recompose the text to make sure that getting the
Rem geometric bounds of the inline graphic will work.
myTextFrame.Texts.Item(1).Recompose
Rem Get the geometric bounds of the inline frame.
myBounds = myAnchoredFrame.GeometricBounds
Rem Set the width and height of the inline frame. In this example, we'll
Rem make the frame 24 points tall by 72 points wide.
myArray = Array(myBounds(0), myBounds(1), myBounds(0) + 24, myBounds(1) + 72)
myAnchoredFrame.GeometricBounds = myArray
myAnchoredFrame.Contents = "This is an anchored frame."
With myAnchoredFrame.AnchoredObjectSettings
    .AnchoredPosition = idAnchorPosition.idAnchored
    .AnchorPoint = idAnchorPoint.idTopLeftAnchor
    .HorizontalReferencePoint = idAnchoredRelativeTo.idAnchorLocation
    .HorizontalAlignment = idHorizontalAlignment.idLeftAlign
    .AnchorXoffset = 72
    .VerticalReferencePoint = idVerticallyRelativeTo.idLineBaseline
    .AnchorYoffset = 24
    .AnchorSpaceAbove = 24
End With
```

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.)

```
With myInDesign.TextDefaults
    .AlignToBaseline = True
    Rem Because the font might not be available, it's usually best
    Rem to trap errors using "On Error Resume Next" error handling.
    Rem Fill in the name of a font on your system.
    Err.Clear
    On Error Resume Next
    .AppliedFont = myInDesign.Fonts.Item("Minion Pro")
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0
    Rem Because the font style might not be available, it's usually best
    Rem to trap errors using "On Error Resume Next" error handling.
    Err.Clear
    On Error Resume Next
        .FontStyle = "Regular"
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0
```

```
Rem Because the language might not be available, it's usually best
Rem to trap errors using "On Error Resume Next" error handling.
Err.Clear
On Error Resume Next
.AppliedLanguage = "English: USA"
If Err.Number <> 0 Then
    Err.Clear
End If
On Error GoTo 0
.AutoLeading = 100
.BalanceRaggedLines = False
.BaselineShift = 0
.Capitalization = idCapitalization.idNormal
.Composer = "Adobe Paragraph Composer"
.DesiredGlyphScaling = 100
.DesiredLetterSpacing = 0
.DesiredWordSpacing = 100
.DropCapCharacters = 0
If .DropCapCharacters <> 0 Then
    .DropCapLines = 3
    On Error Resume Next
    .DropCapStyle = myInDesign.CharacterStyles.Item("myDropCap")
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0
End If
On Error Resume Next
.FillColor = myInDesign.Colors.Item("Black")
If Err.Number <> 0 Then
    Err.Clear
End If
On Error GoTo 0
.FillTint = 100
.FirstLineIndent = 14
.GridAlignFirstLineOnly = False
.HorizontalScale = 100
.HyphenateAfterFirst = 3
.HyphenateBeforeLast = 4
.HyphenateCapitalizedWords = False
.HyphenateLadderLimit = 1
.HyphenateWordsLongerThan = 5
.Hyphenation = True
.HyphenationZone = 36
.HyphenWeight = 9
.Justification = idJustification.idLeftAlign
.KeepAllLinesTogether = False
.KeepLinesTogether = True
.KeepFirstLines = 2
.KeepLastLines = 2
.KeepWithNext = 0
.KerningMethod = "Optical"
.Leading = 14
.LeftIndent = 0
.Ligatures = True
.MaximumGlyphScaling = 100
.MaximumLetterSpacing = 0
.MaximumWordSpacing = 160
.MinimumGlyphScaling = 100
.MinimumLetterSpacing = 0
```

```
.MinimumWordSpacing = 80
.NoBreak = False
.OTFContextualAlternate = True
.OTFDiscretionaryLigature = True
.OTFFigureStyle = idOTFFigureStyle.idProportionalOldstyle
.OTFFraction = True
.OTFHistorical = True
.OTFOrdinal = False
.OTFSlashedZero = True
.OTFSwash = False
.OTFTitling = False
.OverprintFill = False
.OverprintStroke = False
.PointSize = 11
.Position = idPosition.idNormal
.RightIndent = 0
.RuleAbove = False
If .RuleAbove = True Then
    On Error Resume Next
    .RuleAboveColor = myInDesign.Colors.Item("Black")
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0
    On Error Resume Next
    .RuleAboveGapColor = myInDesign.Swatches.Item("None")
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0
    .RuleAboveGapOverprint = False
    .RuleAboveGapTint = 100
    .RuleAboveLeftIndent = 0
    .RuleAboveLineWeight = 0.25
    .RuleAboveOffset = 14
    .RuleAboveOverprint = False
    .RuleAboveRightIndent = 0
    .RuleAboveTint = 100
    On Error Resume Next
    .RuleAboveType = myInDesign.StrokeStyles.Item("Solid")
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0
    .RuleAboveWidth = idRuleWidth.idColumnWidth
End If
.RuleBelow = False
If .RuleBelow = True Then
    On Error Resume Next
    .RuleBelowColor = myInDesign.Colors.Item("Black")
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0
    On Error Resume Next
    .RuleBelowGapColor = myInDesign.Swatches.Item("None")
    If Err.Number <> 0 Then
        Err.Clear
    End If
```

```
    On Error GoTo 0
    .RuleBelowGapOverPrint = False
    .RuleBelowGapTint = 100
    .RuleBelowLeftIndent = 0
    .RuleBelowLineWeight = 0.25
    .RuleBelowOffset = 0
    .RuleBelowOverPrint = False
    .RuleBelowRightIndent = 0
    .RuleBelowTint = 100
    On Error Resume Next
    .RuleBelowType = myInDesign.StrokeStyles.Item("Solid")
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0
    .RuleBelowWidth = idRuleWidth.idColumnWidth
End If
.SingleWordJustification = idSingleWordJustification.idLeftAlign
.Skew = 0
.SpaceAfter = 0
.SpaceBefore = 0
.StartParagraph = idStartParagraph.idAnywhere
.StrikeThru = False
If .StrikeThru = True Then
    On Error Resume Next
    .StrikeThroughColor = myInDesign.Colors.Item("Black")
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0
    On Error Resume Next
    .StrikeThroughGapColor = myInDesign.Swatches.Item("None")
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0
    .StrikeThroughGapOverprint = False
    .StrikeThroughGapTint = 100
    .StrikeThroughOffset = 3
    .StrikeThroughOverprint = False
    .StrikeThroughTint = 100
    On Error Resume Next
    .StrikeThroughType = myInDesign.StrokeStyles.Item("Solid")
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0
    .StrikeThroughWeight = 0.25
End If
On Error Resume Next
.StrokeColor = myInDesign.Swatches.Item("None")
If Err.Number <> 0 Then
    Err.Clear
End If
On Error GoTo 0
.StrokeTint = 100
.StrokeWeight = 0
.Tracking = 0
.Underline = False
```

```
    If .Underline = True Then
        On Error Resume Next
        .UnderlineColor = myInDesign.Colors.Item("Black")
        If Err.Number <> 0 Then
            Err.Clear
        End If
        On Error GoTo 0
        On Error Resume Next
        .UnderlineGapColor = myInDesign.Swatches.Item("None")
        If Err.Number <> 0 Then
            Err.Clear
        End If
        On Error GoTo 0
        .UnderlineGapOverprint = False
        .UnderlineGapTint = 100
        .UnderlineOffset = 3
        .UnderlineOverprint = False
        .UnderlineTint = 100
        On Error Resume Next
        .UnderlineType = myInDesign.StrokeStyles.Item("Solid")
        If Err.Number <> 0 Then
            Err.Clear
        End If
        On Error GoTo 0
        .UnderlineWeight = 0.25
    End If
    .VerticalScale = 100
End With
```

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.)

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myApplicationFonts = myInDesign.Fonts
myString = "Document Fonts:" & vbCr
For myCounter = 1 To myDocument.Fonts.Count
    myString = myString & myDocument.Fonts.Item(myCounter).Name & vbCr
Next
myString = myString & vbCr & "Application Fonts:" & vbCr
For myCounter = 1 To myInDesign.Fonts.Count
    myString = myString & myInDesign.Fonts.Item(myCounter) & vbCr
Next
Set myTextFrame = myPage.TextFrames.Item(1)
Set myStory = myTextFrame.ParentStory
myStory.Contents = myString
```

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

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

Applying a font

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

```
Rem Given a font name "myFontName" and a text object "myText"...
myText.AppliedFont = myInDesign.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 = myInDesign.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:

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myTextObject = myStory.Characters.Item(1)
myFontName = "Minion Pro" & vbTab & "Regular"
With myTextObject
    .AlignToBaseline = False
    .AppliedCharacterStyle = myDocument.CharacterStyles.Item("[None]")
    On Error Resume Next
    .AppliedFont = myInDesign.Fonts.Item(myFontName)
    .FontStyle = "Regular"
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error Goto 0
    .AppliedLanguage = myInDesign.LanguagesWithVendors.Item("English: USA")
    .AppliedNumberingList = myDocument.NumberingLists.Item("[Default]")
    .AppliedParagraphStyle = myDocument.ParagraphStyles.Item("[No Paragraph Style]")
    .AutoLeading = 120
    .BalanceRaggedLines = idBalanceLinesStyle.idNoBalancing
    .BaselineShift = 0
    .BulletsAlignment = idListAlignment.idLeftAlign
    .BulletsAndNumberingListType = idListType.idNoList
    .BulletsCharacterStyle = myDocument.CharacterStyles.Item("[None]")
    .BulletsTextAfter = "^t"
    .Capitalization = idCapitalization.idNormal
    .Composer = "Adobe Paragraph Composer"
    .DesiredGlyphScaling = 100
    .DesiredLetterSpacing = 0
    .DesiredWordSpacing = 100
    .DropCapCharacters = 0
    .DropCapLines = 0
    .DropCapStyle = myDocument.CharacterStyles.Item("[None]")
    .DropcapDetail = 0
    .FillColor = myDocument.Colors.Item("Black")
    .FillTint = -1
    .FirstLineIndent = 0
```

```
.GradientFillAngle = 0
.GradientFillLength = -1
.GradientFillStart = Array(0, 0)
.GradientStrokeAngle = 0
.GradientStrokeLength = -1
.GradientStrokeStart = Array(0, 0)
.GridAlignFirstLineOnly = False
.HorizontalScale = 100
.HyphenWeight = 5
.HyphenateAcrossColumns = True
.HyphenateAfterFirst = 2
.HyphenateBeforeLast = 2
.HyphenateCapitalizedWords = True
.HyphenateLadderLimit = 3
.HyphenateLastWord = True
.HyphenateWordsLongerThan = 5
.Hyphenation = True
.HyphenationZone = 3
.IgnoreEdgeAlignment = False
.Justification = idJustification.idLeftAlign
.KeepAllLinesTogether = False
.KeepFirstLines = 2
.KeepLastLines = 2
.KeepLinesTogether = False
.KeepRuleAboveInFrame = False
.KeepWithNext = 0
.KerningMethod = "Optical"
.LastLineIndent = 0
.Leading = 12
.LeftIndent = 0
.Ligatures = True
.MaximumGlyphScaling = 100
.MaximumLetterSpacing = 0
.MaximumWordSpacing = 133
.MinimumGlyphScaling = 100
.MinimumLetterSpacing = 0
.MinimumWordSpacing = 80
.NoBreak = False
.NumberingAlignment = idListAlignment.idLeftAlign
.NumberingApplyRestartPolicy = True
.NumberingCharacterStyle = myDocument.CharacterStyles.Item("[None]")
.NumberingContinue = True
.NumberingExpression = "^#.^t"
.NumberingFormat = "1, 2, 3, 4..."
.NumberingLevel = 1
.NumberingStartAt = 1
.OTFContextualAlternate = True
.OTFDiscretionaryLigature = False
.OTFFigureStyle = idOTFFigureStyle.idProportionalLining
.OTFFraction = False
.OTFHistorical = False
.OTFLocale = True
.OTFMark = True
.OTFOrdinal = False
.OTFSlashedZero = False
.OTFStylisticSets = 0
.OTFSwash = False
.OTFTitling = False
.OverprintFill = False
.OverprintStroke = False
```

```
.PointSize = 12
.Position = idPosition.idNormal
.PositionalForm = idPositionalForms.idNone
.RightIndent = 0
.RuleAbove = False
.RuleAboveColor = "Text Color"
.RuleAboveGapColor = myDocument.Swatches.Item("None")
.RuleAboveGapOverprint = False
.RuleAboveGapTint = -1
.RuleAboveLeftIndent = 0
.RuleAboveLineWeight = 1
.RuleAboveOffset = 0
.RuleAboveOverprint = False
.RuleAboveRightIndent = 0
.RuleAboveTint = -1
.RuleAboveType = myDocument.StrokeStyles.Item("Solid")
.RuleAboveWidth = idRuleWidth.idColumnWidth
.RuleBelow = False
.RuleBelowColor = "Text Color"
.RuleBelowGapColor = myDocument.Swatches.Item("None")
.RuleBelowGapOverprint = False
.RuleBelowGapTint = -1
.RuleBelowLeftIndent = 0
.RuleBelowLineWeight = 1
.RuleBelowOffset = 0
.RuleBelowOverprint = False
.RuleBelowRightIndent = 0
.RuleBelowTint = -1
.RuleBelowType = myDocument.StrokeStyles.Item("Solid")
.RuleBelowWidth = idRuleWidth.idColumnWidth
.SingleWordJustification = idSingleWordJustification.idLeftAlign
.Skew = 0
.SpaceAfter = 0
.SpaceBefore = 0
.StartParagraph = idStartParagraph.idAnywhere
.StrikeThroughColor = "Text Color"
.StrikeThroughGapColor = myDocument.Swatches.Item("None")
.StrikeThroughGapOverprint = False
.StrikeThroughGapTint = -1
.StrikeThroughOffset = -9999
.StrikeThroughOverprint = False
.StrikeThroughTint = -1
.StrikeThroughType = myDocument.StrokeStyles.Item("Solid")
.StrikeThroughWeight = -9999
.StrikeThru = False
.StrokeColor = myDocument.Swatches.Item("None")
.StrokeTint = -1
.StrokeWeight = 1
.Tracking = 0
.Underline = False
.UnderlineColor = "Text Color"
.UnderlineGapColor = myDocument.Swatches.Item("None")
.UnderlineGapOverprint = False
.UnderlineGapTint = -1
.UnderlineOffset = -9999
.UnderlineOverprint = False
.UnderlineTint = -1
.UnderlineType = myDocument.StrokeStyles.Item("Solid")
```

```
    .UnderlineWeight = -9999
    .VerticalScale = 100
End With
```

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):

```
Set myDocument = myInDesign.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Rem Apply a color to the fill of the text.
Set myText = myStory.Paragraphs.Item(1)
myText.FillColor = myDocument.Colors.Item("DGC1_446a")
Rem Use the itemByRange method to apply the color to the stroke of the text.
myText.StrokeColor = myDocument.Swatches.Item("DGC1_446b")
Set myText = myStory.Paragraphs.Item(2)
myText.FillColor = myDocument.Swatches.Item("DGC1_446b")
myText.StrokeColor = myDocument.Swatches.Item("DGC1_446a")
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):

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrame = myPage.TextFrames.Item(1)
Rem Create a color for use by one of the paragraph styles we'll create.
Set myColor = myAddColor(myDocument, "Red", idColorModel.idProcess,
Array(0, 100, 100, 0))
Rem Create a character style named "myCharacterStyle" if
Rem no style by that name already exists.
Set myCharacterStyle = myAddStyle(myDocument, "myCharacterStyle", 1)
Rem At this point, the variable myCharacterStyle contains a reference to a character
Rem style object, which you can now use to specify formatting.
myCharacterStyle.FillColor = myColor
Rem Create a paragraph style named "myParagraphStyle" if
Rem no style by that name already exists.
Set myParagraphStyle = myAddStyle(myDocument, "myParagraphStyle", 2)
Rem At this point, the variable myParagraphStyle contains a reference to a paragraph
Rem style object, which you can now use to specify formatting.
myTextFrame.ParentStory.Texts.Item(1).ApplyParagraphStyle myParagraphStyle, True
Set myStartCharacter = myTextFrame.ParentStory.Characters.Item(14)
Set myEndCharacter = myTextFrame.ParentStory.Characters.Item(55)
Set myText = myTextFrame.ParentStory.Texts.ItemByRange(myStartCharacter,
myEndCharacter)
myText.Item(1).ApplyCharacterStyle myCharacterStyle, True
```

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

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

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

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):

```
Set myDocument = myInDesign.Documents.Item(1)
Set myParagraphStyleA = myDocument.ParagraphStyles.Item("myParagraphStyleA")
Rem Delete the paragraph style myParagraphStyleA and replace with myParagraphStyleB.
myParagraphStyleA.Delete myDocument.ParagraphStyles.Item("myParagraphStyleB")
```

Importing paragraph and character styles

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

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myNewDocument = myInDesign.Documents.Add
Rem Import the styles from the saved document.
Rem ImportStyles parameters:
Rem Format as idImportFormat enumeration. Options for text styles are:
Rem idImportFormat.idParagraphStylesFormat
Rem idImportFormat.idCharacterStylesFormat
Rem idImportFormat.idTextStylesFormat
Rem From as string (file path)
Rem GlobalStrategy as idGlobalClashResolutionStrategy enumeration. Options are:
Rem idGlobalClashResolutionStrategy.idDoNotLoadTheStyle
Rem idGlobalClashResolutionStrategy.idLoadAllWithOverwrite
Rem idGlobalClashResolutionStrategy.idLoadAllWithRename
myNewDocument.ImportStyles idImportFormat.idTextStylesFormat, "c:\styles.indd",
idGlobalClashResolutionStrategy.idLoadAllWithOverwrite
```

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:

   ```
   Rem Find/Change text preferences
   Set myInDesign = CreateObject("InDesign.Application.CS4")
   myInDesign.FindTextPreferences = idNothingEnum.idNothing
   myInDesign.ChangeTextPreferences = idNothingEnum.idNothing
   ```

   ```
   Rem Find/Change grep preferences
   Set myInDesign = CreateObject("InDesign.Application.CS4")
   myInDesign.FindGrepPreferences = idNothingEnum.idNothing
   myInDesign.ChangeGrepPreferences = idNothingEnum.idNothing
   ```

   ```
   Rem Find/Change glyph preferences
   Set myInDesign = CreateObject("InDesign.Application.CS4")
   myInDesign.FindGlyphPreferences = idNothingEnum.idNothing
   myInDesign.ChangeGlyphPreferences = idNothingEnum.idNothing
   ```

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.)

```
Set myDocument = myInDesign.Documents.Item(1)
Rem Clear the find/change text preferences.
myInDesign.FindTextPreferences = idNothingEnum.idNothing
myInDesign.ChangeTextPreferences = idNothingEnum.idNothing
Rem Search the document for the string "text".
myInDesign.FindTextPreferences.FindWhat = "text"
Rem Set the find options.
myInDesign.FindChangeTextOptions.CaseSensitive = False
myInDesign.FindChangeTextOptions.IncludeFootnotes = False
myInDesign.FindChangeTextOptions.IncludeHiddenLayers = False
myInDesign.FindChangeTextOptions.IncludeLockedLayersForFind = False
myInDesign.FindChangeTextOptions.IncludeLockedStoriesForFind = False
myInDesign.FindChangeTextOptions.IncludeMasterPages = False
myInDesign.FindChangeTextOptions.WholeWord = False
Set myFoundItems = myInDesign.Documents.Item(1).FindText
MsgBox ("Found " & CStr(myFoundItems.Count) & " instances of the search string.")
Rem Clear the find/change text preferences after the search.
myInDesign.FindTextPreferences = idNothingEnum.idNothing
myInDesign.ChangeTextPreferences = idNothingEnum.idNothing
```

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):

```
Set myDocument = myInDesign.Documents.Item(1)
Rem Clear the find/change text preferences.
myInDesign.FindTextPreferences = idNothingEnum.idNothing
myInDesign.ChangeTextPreferences = idNothingEnum.idNothing
Rem Search the document for the string "copy" and replace with "text".
myInDesign.FindTextPreferences.FindWhat = "copy"
myInDesign.ChangeTextPreferences.ChangeTo = "text"
Rem Set the find options.
myInDesign.FindChangeTextOptions.CaseSensitive = False
myInDesign.FindChangeTextOptions.IncludeFootnotes = False
myInDesign.FindChangeTextOptions.IncludeHiddenLayers = False
myInDesign.FindChangeTextOptions.IncludeLockedLayersForFind = False
myInDesign.FindChangeTextOptions.IncludeLockedStoriesForFind = False
myInDesign.FindChangeTextOptions.IncludeMasterPages = False
myInDesign.FindChangeTextOptions.WholeWord = False
Set myFoundItems = myInDesign.Documents.Item(1).ChangeText
MsgBox ("Changed " & CStr(myFoundItems.Count) & " instances of the search string.")
Rem Clear the find/change text preferences.
myInDesign.FindTextPreferences = idNothingEnum.idNothing
myInDesign.ChangeTextPreferences = idNothingEnum.idNothing
```

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):

```
Rem Clear the find/change preferences.
myInDesign.FindTextPreferences = idNothingEnum.idNothing
myInDesign.ChangeTextPreferences = idNothingEnum.idNothing
Rem Set the find options.
myInDesign.FindChangeTextOptions.CaseSensitive = false
myInDesign.FindChangeTextOptions.IncludeFootnotes = false
myInDesign.FindChangeTextOptions.IncludeHiddenLayers = false
myInDesign.FindChangeTextOptions.IncludeLockedLayersForFind = false
myInDesign.FindChangeTextOptions.IncludeLockedStoriesForFind = false
myInDesign.FindChangeTextOptions.IncludeMasterPages = false
myInDesign.FindChangeTextOptions.WholeWord = false
Rem Search the document for the 24 point text and change it to 10 point text.
myInDesign.findTextPreferences.pointSize = 24
myInDesign.changeTextPreferences.pointSize = 10
myInDesign.documents.item(1).changeText
Rem Clear the find/change preferences after the search.
myInDesign.FindTextPreferences = idNothingEnum.idNothing
myInDesign.ChangeTextPreferences = idNothingEnum.idNothing
```

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):

```
Set myDocument = myInDesign.Documents.Item(1)
Rem Clear the find/change grep preferences.
myInDesign.FindGrepPreferences = idNothingEnum.idNothing
myInDesign.ChangeGrepPreferences = idNothingEnum.idNothing
Rem Set the find options.
myInDesign.FindChangeGrepOptions.IncludeFootnotes = False
myInDesign.FindChangeGrepOptions.IncludeHiddenLayers = False
myInDesign.FindChangeGrepOptions.IncludeLockedLayersForFind = False
myInDesign.FindChangeGrepOptions.IncludeLockedStoriesForFind = False
myInDesign.FindChangeGrepOptions.IncludeMasterPages = False
Rem Regular expression for finding an email address.
myInDesign.FindGrepPreferences.FindWhat = "(?i)[A-Z]*?@[A-Z]*?[.]..."
Rem Apply the change to 24-point text only.
myInDesign.FindGrepPreferences.PointSize = 24
myInDesign.ChangeGrepPreferences.Underline = True
myDocument.ChangeGrep
Rem Clear the find/change grep preferences after the search.
myInDesign.FindGrepPreferences = idNothingEnum.idNothing
myInDesign.ChangeGrepPreferences = idNothingEnum.idNothing
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem Access the active document.
Set myDocument = myInDesign.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
myReadPMTags myInDesign, myStory
```

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

```
Function myReadPMTags(myInDesign, myStory)
    Set myDocument = myStory.Parent
    Rem Reset the findGrepPreferences to ensure that previous settings
    Rem do not affect the search.
    myInDesign.FindGrepPreferences = idNothingEnum.idNothing
    myInDesign.ChangeGrepPreferences = idNothingEnum.idNothing
    Rem Find the tags.
    myInDesign.FindGrepPreferences.findWhat = "(?i)^<\s*\w+\s*>"
    Set myFoundItems = myStory.findGrep
    If myFoundItems.Count <> 0 Then
        Set myFoundTags = CreateObject("Scripting.Dictionary")
        For myCounter = 1 To myFoundItems.Count
            If Not (myFoundTags.Exists(myFoundItems.Item(myCounter).Contents)) Then
                myFoundTags.Add myFoundItems.Item(myCounter).Contents,
                myFoundItems.Item(myCounter).Contents
            End If
        Next
        Rem At this point, we have a list of tags to search for.
        For Each myFoundTag In myFoundTags
            myString = myFoundTag
            Rem Find the tag using findWhat.
            myInDesign.FindTextPreferences.findWhat = myString
            Rem Extract the style name from the tag.
            myStyleName = Mid(myString, 2, Len(myString) - 2)
            Rem Create the style if it does not already exist.
            Set myStyle = myAddStyle(myDocument, myStyleName)
            Rem Apply the style to each instance of the tag.
            myInDesign.ChangeTextPreferences.AppliedParagraphStyle = myStyle
            myStory.ChangeText
            Rem Reset the changeTextPreferences.
            myInDesign.ChangeTextPreferences = idNothingEnum.idNothing
            Rem Set the changeTo to an empty string.
            myInDesign.ChangeTextPreferences.ChangeTo = ""
            Rem Search to remove the tags.
            myStory.ChangeText
            Rem Reset the find/change preferences again.
            myInDesign.ChangeTextPreferences = idNothingEnum.idNothing
        Next
    End If
    myInDesign.FindGrepPreferences = idNothingEnum.idNothing
    myInDesign.ChangeGrepPreferences = idNothingEnum.idNothing
End Function
```

Using glyph search

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

```
Rem Clear the find/change glyph preferences.
myInDesign.FindGlyphPreferences = idNothingEnum.idNothing
myInDesign.ChangeGlyphPreferences = idNothingEnum.idNothing
Rem Set the find options.
myInDesign.FindChangeGrepOptions.IncludeFootnotes = False
myInDesign.FindChangeGrepOptions.IncludeHiddenLayers = False
myInDesign.FindChangeGrepOptions.IncludeLockedLayersForFind = False
myInDesign.FindChangeGrepOptions.IncludeLockedStoriesForFind = False
myInDesign.FindChangeGrepOptions.IncludeMasterPages = False
Rem You must provide a font that is used in the document for the
Rem AppliedFont property of the FindGlyphPreferences object.
myInDesign.FindGlyphPreferences.AppliedFont = myDocument.Fonts.Item("Times New Roman
Regular");
Rem Provide the glyph ID, not the glyph Unicode value.
myInDesign.FindGlyphPreferences.GlyphID = 374;
Rem The appliedFont of the changeGlyphPreferences object can be
Rem any font available to the application.
myInDesign.changeGlyphPreferences.AppliedFont = myInDesign.Fonts.Item("ITC Zapf
DingbatsMedium");
myInDesign.Documents.Item(1).ChangeGlyph
Rem Clear the find/change glyph preferences after the search.
myInDesign.FindGlyphPreferences = idNothingEnum.idNothing
myInDesign.ChangeGlyphPreferences = idNothingEnum.idNothing
```

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):

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Set myTextFrame = myPage.TextFrames.Add
Rem Set the bounds of the text frame.
myTextFrame.GeometricBounds = myGetBounds(myDocument, myPage)
Rem Fill the text frame with placeholder text.
myString = "Table 1" & vbCr
myString = myString & "Column 1" & vbTab & "Column 2" & vbTab & "Column 3" & vbCr & "1a"
& vbTab & "1b" & vbTab & "1c" & vbCr & "2a" & vbTab & "2b" & vbTab & "2c" & vbCr & "3a" &
vbTab & "3b" & vbTab & "3c" & vbCr
myString = myString & "Table 2" & vbCr
myString = myString & "Column 1,Column 2,Column 3;1a,1b,1c;2a,2b,2c;3a,3b,3c" & vbCr
myString = myString & "Table 3" & vbCr
myTextFrame.Contents = myString
Set myStory = myTextFrame.ParentStory
Set myStartCharacter = myStory.Paragraphs.Item(7).Characters.Item(1)
Set myEndCharacter = myStory.Paragraphs.Item(7).Characters.Item(-2)
Set myText = myStory.Texts.ItemByRange(myStartCharacter, myEndCharacter).Item(1)
Rem The convertToTable method takes three parameters:
Rem [ColumnSeparator as string]
```

```
Rem [RowSeparator as string]
Rem [NumberOfColumns as integer] (only used if the ColumnSeparator
Rem and RowSeparator values are the same)
Rem In the last paragraph in the story, columns are separated by commas
Rem and rows are separated by semicolons, so we provide those characters
Rem to the method as parameters.
Set myTable = myText.ConvertToTable(",", ";")
Set myStartCharacter = myStory.Paragraphs.Item(2).Characters.Item(1)
Set myEndCharacter = myStory.Paragraphs.Item(5).Characters.Item(-2)
Set myText = myStory.Texts.ItemByRange(myStartCharacter, myEndCharacter).Item(1)
Rem In the second through the fifth paragraphs, colums are separated by
Rem tabs and rows are separated by returns. These are the default delimiter
Rem parameters, so we don't need to provide them to the method.
Set myTable = myText.ConvertToTable
Rem You can also explicitly add a table--you don't have to convert text to a table.
Set 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.)

```
Set myDocument = myInDesign.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myTable = myStory.Tables.Item(1)
Rem Merge all of the cells in the first column.
myTable.Cells.Item(1).Merge myTable.Columns.Item(1).Cells.Item(-1)
Rem Convert column 2 into 2 cells (rather than 4).
myTable.Columns.Item(2).Cells.Item(-1).Merge myTable.Columns.Item(2).Cells.Item(-2)
myTable.Columns.Item(2).Cells.Item(1).Merge myTable.Columns.Item(2).Cells.Item(2)
Rem Merge the last two cells in row 1.
myTable.Rows.Item(1).Cells.Item(-1).Merge myTable.Rows.Item(1).Cells.Item(-1)
Rem Merge the last two cells in row 3.
myTable.Rows.Item(3).Cells.Item(-2).Merge myTable.Rows.Item(3).Cells.Item(-1)
```

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

```
Set myDocument = myInDesign.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myTable = myStory.InsertionPoints.Item(-1).Tables.Add
myTable.ColumnCount = 1
myTable.BodyRowCount = 1
myArray = myGetBounds(myDocument, myDocument.Pages.Item(1))
myWidth = myArray(3) - myArray(1)
myTable.Columns.Item(1).Width = myWidth
myTable.Cells.Item(1).Split idHorizontalOrVertical.idHorizontal
myTable.Columns.Item(1).Split idHorizontalOrVertical.idVertical
myTable.Cells.Item(1).Split idHorizontalOrVertical.idVertical
myTable.Rows.Item(-1).Split idHorizontalOrVertical.idHorizontal
myTable.Cells.Item(-1).Split idHorizontalOrVertical.idVertical
For myRowCounter = 1 To myTable.Rows.Count
    Set myRow = myTable.Rows.Item(myRowCounter)
    For myCellCounter = 1 To myRow.Cells.Count
        myString = "Row: " & myRowCounter & " Cell: " & myCellCounter
        myRow.Cells.Item(myCellCounter).contents = myString
    Next
Next
```

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

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Rem Create a text frame on page 1.
Set myTable = myDocument.Stories.Item(1).Tables.Item(1)
Rem Convert the first row to a header row.
myTable.Rows.Item(1).RowType = idRowTypes.idHeaderRow
Rem Convert the last row to a footer row.
myTable.Rows.Item(-1).RowType = idRowTypes.idFooterRow
```

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

```
Set myDocument = myInDesign.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myTable = myStory.Tables.Item(1)
Rem Convert the first row to a header row.
myTable.Rows.Item(1).RowType = idRowTypes.idHeaderRow
Rem Use a reference to a swatch, rather than to a color.
myTable.Rows.Item(1).FillColor = myDocument.Swatches.Item("DGC1_446b")
myTable.Rows.Item(1).FillTint = 40
myTable.Rows.Item(2).FillColor = myDocument.Swatches.Item("DGC1_446a")
myTable.Rows.Item(2).FillTint = 40
myTable.Rows.Item(3).FillColor = myDocument.Swatches.Item("DGC1_446a")
myTable.Rows.Item(3).FillTint = 20
myTable.Rows.Item(4).FillColor = myDocument.Swatches.Item("DGC1_446a")
myTable.Rows.Item(4).FillTint = 40
Rem Iterate through the cells to apply the cell stroke formatting.
For myCounter = 1 To myTable.Cells.Count
    myTable.Cells.Item(myCounter).TopEdgeStrokeColor =
    myDocument.Swatches.Item("DGC1_446b")
    myTable.Cells.Item(myCounter).TopEdgeStrokeWeight = 1
    myTable.Cells.Item(myCounter).BottomEdgeStrokeColor =
    myDocument.Swatches.Item("DGC1_446b")
    myTable.Cells.Item(myCounter).BottomEdgeStrokeWeight = 1
    Rem When you set a cell stroke to a swatch, make certain that
    Remyou also set the stroke weight.
    myTable.Cells.Item(myCounter).LeftEdgeStrokeColor =
    myDocument.Swatches.Item("None")
    myTable.Cells.Item(myCounter).LeftEdgeStrokeWeight = 0
    myTable.Cells.Item(myCounter).RightEdgeStrokeColor =
    myDocument.Swatches.Item("None")
    myTable.Cells.Item(myCounter).RightEdgeStrokeWeight = 0
Next
```

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

```
Set myDocument = myInDesign.Documents.Item(1)
Set myTable = myDocument.stories.Item(1).tables.Item(1)
Rem Apply alternating fills to the table.
myTable.alternatingFills = idAlternatingFillsTypes.idAlternatingRows
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.)

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.Documents.Count <> 0 Then
    If myInDesign.Selection.Count <> 0 Then
        Select Case TypeName(myInDesign.Selection.Item(1))
            Rem When a row, a column, or a range of cells is selected,
            Rem the type returned is "Cell"
            Case "Cell"
                MsgBox ("A cell is selected.")
            Case "Table"
                MsgBox ("A table is selected.")
            Case "InsertionPoint", "Character", "Word", "TextStyleRange",
            "Line", "Paragraph", "TextColumn", "Text"
                If TypeName(myInDesign.Selection.Item(1).Parent) = "Cell" Then
                    MsgBox ("The selection is inside a table cell.")
                End If
            Case "Rectangle", "Oval", "Polygon", "GraphicLine"
                If TypeName(myInDesign.Selection.Item(1).Parent.Parent) = "Cell" Then
                    MsgBox ("The selection is inside a table cell.")
                End If
            Case "Image", "PDF", "EPS"
                If TypeName(myInDesign.Selection.Item(1).Parent.Parent.Parent) =
                "Cell" Then
                    MsgBox ("The selection is inside a table cell.")
                End If
            Case Else
                MsgBox ("The selection is not inside a table.")
        End Select
    End If
End If
```

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):

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
Rem Create a rectangle on the first page.
Set myTextFrame = myPage.TextFrames.Add
myTextFrame.geometricBounds = Array(72, 72, 288, 288)
Set myTextPath = myTextFrame.TextPaths.Add
myTextPath.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 79).

Autocorrect

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

```
ReDim myNewWordPairList(0)
Rem Add a word pair to the autocorrect list. Each
Rem AutoCorrectTable is linked to a specific language.
Set myAutoCorrectTable = myInDesign.AutoCorrectTables.Item("English: USA")
Rem To safely add a word pair to the auto correct table, get the current
Rem word pair list, then add the new word pair to that array, and then
Rem set the autocorrect word pair list to the array.
myWordPairList = myAutoCorrectTable.AutoCorrectWordPairList
ReDim myNewWordPairList(UBound(myWordPairList))
For myCounter = 0 To UBound(myWordPairList) - 1
    myNewWordPairList(myCounter) = myWordPairList(myCounter)
Next
Rem Add a new word pair to the array.
myNewWordPairList(UBound(myNewWordPairList)) = (Array("paragarph", "paragraph"))
Rem Update the word pair list.
myAutoCorrectTable.AutoCorrectWordPairList = myNewWordPairList
Rem To clear all autocorrect word pairs in the current dictionary:
Rem myAutoCorrectTable.autoCorrectWordPairList = array(())
Rem Turn autocorrect on if it's not on already.
If myInDesign.AutoCorrectPreferences.AutoCorrect = False Then
    myInDesign.AutoCorrectPreferences.AutoCorrect = True
End If
myInDesign.AutoCorrectPreferences.AutoCorrectCapitalizationErrors = True
```

Footnotes

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

```
Set myDocument = myInDesign.Documents.Item(1)
Set myPage = myDocument.Pages.Item(1)
With myDocument.FootnoteOptions
    .SeparatorText = vbTab
    .MarkerPositioning = idFootnoteMarkerPositioning.idSuperscriptMarker
End With
Set myTextFrame = myDocument.Pages.Item(1).TextFrames.Item(1)
Rem Add four footnotes at random locations in the story.
For myCounter = 1 To 4
    myRandomNumber = CLng(myGetRandom(1, myTextFrame.ParentStory.Words.Count))
    Set myWord = myTextFrame.ParentStory.Words.Item(myRandomNumber)
    Set myFootnote = myWord.InsertionPoints.Item(-1).Footnotes.Add
    Rem Note: when you create a footnote, it contains text--the footnote marker
    Rem and the separator text (if any). If you try to set the text of the footnote
    Rem by setting the footnote contents, you will delete the marker. Instead, append
    Rem the footnote text, as shown below.
    myFootnote.InsertionPoints.Item(-1).Contents = "This is a footnote."
Next
```

Setting text preferences

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

```
With myInDesign.TextPreferences
    .AbutTextToTextWrap = True
    .AddPages = False
    Rem baseline shift key increment can range from .001 to 200 points.
    .BaselineShiftKeyIncrement = 1
    .DeleteEmptyPages = False
    .EnablePreviewStyleMode = False
    .HighlightCustomSpacing = False
    .HighlightHjViolations = True
    .HighlightKeeps = True
    .HighlightSubstitutedFonts = True
    .HighlightSubstitutedGlyphs = True
    .JustifyTextWraps = True
    Rem kerning key increment value is 1/1000 of an em.
    .KerningKeyIncrement = 10
    Rem leading key increment value can range from .001 to 200 points.
    .LeadingKeyIncrement = 1
    .LimitToMasterTextFrames = False
    .LinkTextFilesWhenImporting = False
    .PreserveFacingPageSpreads = False
    .ShowInvisibles = True
    .SmallCap = 60
    .SmartTextReflow = False
    .SubscriptPosition = 30
    .SubscriptSize = 60
    .SuperscriptPosition = 30
    .SuperscriptSize = 60
    .TypographersQuotes = False
    .UseOpticalSize = False
    .UseParagraphLeading = False
    .ZOrderTextWrap = False
End With
Rem Text editing preferences are application-wide.
With myInDesign.TextEditingPreferences
    .AllowDragAndDropTextInStory = True
    .DragAndDropTextInLayout = True
    .SingleClickConvertsFramesToTextFrames = True
    .SmartCutAndPaste = True
    .TripleClickSelectsLine = False
End With
```

6 User Interfaces

VBScript 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.

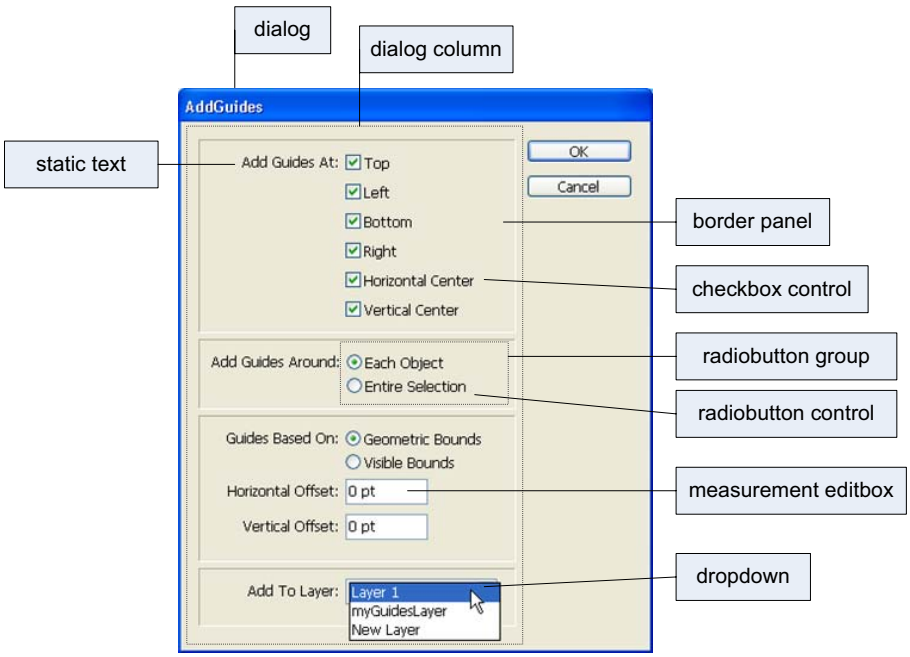
NOTE: InDesign scripts written in JavaScript also can include user interfaces created using the Adobe *ScriptUI* component. This chapter includes some ScriptUI scripting tutorials; for more information, see *Adobe Creative Suite® 4 JavaScript Tools Guide*.

NOTE: Although Visual Basic applications can create complete user interfaces, they run from a separate Visual Basic executable file. InDesign scripting includes the ability to create complex dialogs that appear inside InDesign and look very much like the program's standard user interface. VBScripts run from the Scripts palette are much faster than scripts run from an external application.

We assume you already read *Adobe InDesign CS4 Scripting Tutorial* and know how to create and run a script.

Dialog overview

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



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

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

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

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

Your first InDesign dialog

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

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDialog = myInDesign.Dialogs.Add
Rem Add a dialog column.
With myDialog.DialogColumns.Add
    With .StaticTexts.Add
        .StaticLabel = "This is a very simple dialog box."
    End With
End With
Rem Show the dialog box.
myResult = myDialog.Show
Rem If the user clicked OK, display one message;
Rem if they clicked Cancel, display a different message.
If myResult = True Then
    MsgBox("You clicked the OK button!")
Else
    MsgBox("You clicked the Cancel button!")
End If
```

```
Rem Remove the dialog 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 CS4 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:

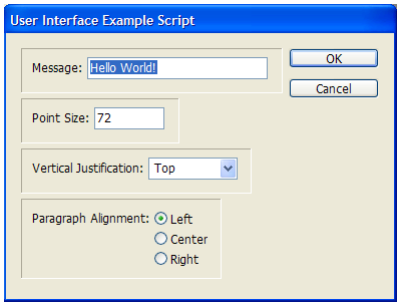
```
Function myDisplayDialog(myInDesign)
    Set myDialog = myInDesign.Dialogs.Add
    myDialog.CanCancel = True
    myDialog.Name = "Simple User Interface Example Script"
    Set myDialogColumn = myDialog.DialogColumns.Add
    Set myTextEditField = myDialogColumn.TextEditboxes.Add
    myTextEditField.EditContents = "Hello World!"
    myTextEditField.MinWidth = 180
    Rem Create a number (real) entry field.
    Set myPointSizeField = myDialogColumn.measurementEditboxes.Add
    myPointSizeField.EditValue = 72
    myResult = myDialog.Show
    If myResult = True Then
        Rem Get the values from the dialog box controls.
        myString = myTextEditField.EditContents
        myPointSize = myPointSizeField.EditValue
        Rem Remove the dialog box from memory.
        myDialog.Destroy
        myMakeDocument(myInDesign, myString, myPointSize)
    Else
        myDialog.Destroy
    End If
End Function
```

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

```
Function myMakeDocument(myInDesign, myString, myPointSize)
    Set myDocument = myInDesign.Documents.Add
    Set myTextFrame = myDocument.Pages.Item(1).TextFrames.Add
    Rem Resize the text frame to the "live" area of the page
    Rem (using the function "myGetBounds").
    myBounds = myGetBounds(myDocument, myDocument.Pages.Item(1))
    myTextFrame.GeometricBounds = myBounds
    Rem Enter the text from the dialog box in the text frame.
    myTextFrame.Contents = myString
    Rem Set the size of the text to the size you entered in the dialog box.
    myTextFrame.Texts.Item(1).PointSize = myPointSize
End Function
```

Creating a more complex user interface

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



For the complete script, see ComplexUI.

```
Set myDialog = myInDesign.Dialogs.Add
myDialog.CanCancel = True
myDialog.Name = "User Interface Example Script"
Rem Create a dialog column.
Set myDialogColumn = myDialog.DialogColumns.Add
Rem Create a border panel.
Set myBorderPanel = myDialogColumn.BorderPanels.Add
Rem Create a dialog column inside the border panel.
Set myTempDialogColumn = myBorderPanel.DialogColumns.Add
Set myStaticText = myTempDialogColumn.StaticTexts.Add
myStaticText.StaticLabel = "Message:"
Rem Create another dialog column inside the border panel.
Set myTempDialogColumn = myBorderPanel.DialogColumns.Add
Set myTextEditField = myTempDialogColumn.TextEditboxes.Add
myTextEditField.EditContents = "Hello World!"
myTextEditField.MinWidth = 180
Rem Create another border panel.
Set myBorderPanel = myDialogColumn.BorderPanels.Add
Rem Create a dialog column inside the border panel.
Set myTempDialogColumn = myBorderPanel.DialogColumns.Add
Set myStaticText = myTempDialogColumn.StaticTexts.Add
myStaticText.StaticLabel = "Point Size:"
Rem Create another dialog column inside the border panel.
Set myTempDialogColumn = myBorderPanel.DialogColumns.Add
Set myPointSizeField = myTempDialogColumn.RealEditboxes.Add
myPointSizeField.EditValue = 72
Rem Create another border panel.
Set myBorderPanel = myDialogColumn.BorderPanels.Add
Rem Create a dialog column inside the border panel.
Set myTempDialogColumn = myBorderPanel.DialogColumns.Add
Set myStaticText = myTempDialogColumn.StaticTexts.Add
myStaticText.StaticLabel = "Vertical Justification:"
Rem Create another dialog column inside the border panel.
Set myTempDialogColumn = myBorderPanel.DialogColumns.Add
Set myVerticalJustificationMenu = myTempDialogColumn.Dropdowns.Add
myVerticalJustificationMenu.StringList = Array("Top", "Center", "Bottom")
myVerticalJustificationMenu.SelectedIndex = 0
Rem Create another border panel.
Set myBorderPanel = myDialogColumn.BorderPanels.Add
```

```
Rem Create a dialog column inside the border panel.
Set myTempDialogColumn = myBorderPanel.DialogColumns.Add
Set myStaticText = myTempDialogColumn.StaticTexts.Add
myStaticText.StaticLabel = "Paragraph Alignment:"
Rem Create another dialog column inside the border panel.
Set myTempDialogColumn = myBorderPanel.DialogColumns.Add
Set myRadioButtonGroup = myTempDialogColumn.RadiobuttonGroups.Add
Set myLeftRadioButton = myRadioButtonGroup.RadiobuttonControls.Add
myLeftRadioButton.StaticLabel = "Left"
myLeftRadioButton.CheckedState = True
Set myCenterRadioButton = myRadioButtonGroup.RadiobuttonControls.Add
myCenterRadioButton.StaticLabel = "Center"
Set myRightRadioButton = myRadioButtonGroup.RadiobuttonControls.Add
myRightRadioButton.StaticLabel = "Right"
Rem If the user clicked OK, then create the example document.
If myDialog.Show = True Then
    Rem Get the values from the dialog box controls.
    myString = myTextEditField.EditContents
    myPointSize = myPointSizeField.EditValue
    Select Case myVerticalJustificationMenu.SelectedIndex
        Case 0
            myVerticalJustification = idVerticalJustification.idTopAlign
        Case 1
            myVerticalJustification = idVerticalJustification.idCenterAlign
        Case Else
            myVerticalJustification = idTopAlign.idBottomAlign
        End Select
    Rem set the paragraph alignment of the text to the dialog radio button choice.
    Select Case myRadioButtonGroup.SelectedButton
        Case 0
            myAlignment = idJustification.idLeftAlign
        Case 1
            myAlignment = idJustification.idCenterAlign
        Case Else
            myAlignment = idJustification.idRightAlign
    End Select
    Rem Remove the dialog box from memory.
    myDialog.Destroy
    Rem Create a new document.
    myMakeDocument myInDesign, myString, myPointSize, myParagraphAlignment,
    myVerticalJustification
Else
    myDialog.Destroy
End If
```

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

```
Function myMakeDocument(myInDesign, myString, myPointSize, myAlignment,
myVerticalJustification)
    Rem Create a new document.
    Set myDocument = myInDesign.Documents.Add
    Set myTextFrame = myDocument.Pages.Item(1).TextFrames.Add
    Rem Resize the text frame to the "live" area of the
    Rem page (using the function "myGetBounds").
    myBounds = myGetBounds(myDocument, myDocument.Pages.Item(1))
    myTextFrame.GeometricBounds = myBounds
    Rem Enter the text from the dialog box in the text frame.
    myTextFrame.Contents = myString
```

```
    Rem Set the size of the text to the size you entered
    Rem in the dialog box.
    myTextFrame.Texts.Item(1).PointSize = myPointSize
    Rem Set the paragraph alignment to the alignment you
    Rem selected in the dialog box.
    myTextFrame.Texts.Item(1).Justification = myAlignment
    Rem Set the text frame vertical justification to the vertical justification
    Rem you selected in the dialog box.
    myTextFrame.TextFramePreferences.VerticalJustification = myVerticalJustification
End Function
```

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 VBScript users. InDesign scripts can execute scripts written in other scripting languages using the DoScript 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 a VBScript (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 VBScript (for the complete script, see CallProgressBar):

```
Rem Create a document and add pages to it--
Rem if you do not do this, the progress bar
Rem will go by too quickly.
Set myDocument = myInDesign.Documents.Add
Rem Note that the JavaScripts must use the "session"
Rem engine for this to work.
myString = "#targetengine ""session""" & vbCr
myString = myString & "myCreateProgressPanel(100, 400);" & vbcr
myString = myString & "myProgressPanel.show();" & vbcr
myInDesign.DoScript myString, idScriptLanguage.idJavascript
```

```
For myCounter = 1 to 100
    Rem Add a page to the document.
    myInDesign.Documents.Item(1).Pages.Add
    myString = "#targetengine ""session""" & vbCr
    myString = myString & "myProgressPanel.myProgressBar.value = "
    myString = myString & cstr(myCounter) & "/myIncrement;" & vbcr
    myInDesign.DoScript myString, idScriptLanguage.idJavascript
    If(myCounter = 100) Then
        myString = "#targetengine ""session""" & vbCr
        myString = myString & "myProgressPanel.myProgressBar.value = 0;" & vbcr
        myString = myString & "myProgressPanel.hide();" & vbcr
        myInDesign.DoScript myString, idScriptLanguage.idJavascript
        myDocument.Close idSaveOptions.idNo
    End If
Next
```

7 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 CS4 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 8, "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 (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 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).

```
Set myInDesign = CreateObject("InDesign.Application.CS4"
Set myEventListener = myInDesign.EventListeners.Add("afterNew",
"c:\IDEventListeners\Message.vbs", false)
```

The script referred to in the above script contains the following code:

```
Rem "evt" is the event passed to this script by the event listener.
MsgBox ("This event is the " & evt.EventType & " event.")
```

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

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
Set myFile = myFileSystemObject.GetFile("c:\IDEventHandlers\message.vbs")
myResult = myInDesign.RemoveEventListener("afterNew", myFile, 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 CS4 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.

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myEventListener = myInDesign.EventListeners.Add("beforeImport",
"c:\EventInfo.vbs", True)
Set myDocument = myInDesign.Documents.Add
Set myEventListener = myDocument.EventListeners.Add("beforeImport",
"c:\EventInfo.vbs", False)
```

The EventInfo.vbs script referred to in the above script contains the following script code:

```
main evt
Function main(myEvent)
    myString = "Current Target: " & myEvent.CurrentTarget.Name
    MsgBox myString, vbOKOnly, "Event Details"
end function
```

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.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
myEventNames = Array("beforeQuit", "afterQuit", "beforeNew", "afterNew", "beforeOpen",
"afterOpen", "beforeClose", "afterClose", "beforeSave", "afterSave", "beforeSaveAs",
"afterSaveAs", "beforeSaveACopy", "afterSaveACopy", "beforeRevert", "afterRevert",
"beforePrint", "afterPrint", "beforeExport", "afterExport", "beforeImport",
"afterImport", "beforePlace", "afterPlace")
For myCounter = 0 To UBound(myEventNames)
    myInDesign.AddEventListener myEventNames(myCounter), "c:\GetEventInfo.vbs", False
    If myCounter < UBound(myEventNames) Then
        myInDesign.EventListeners.Add myEventNames(myCounter), "c:\GetEventInfo.vbs",
False
    End If
Next
```

The following script is the one referred to by the above script. The file reference in the script above must match the location of this script on your disk. For the complete script, see GetEventInfo.vbs.

```
main evt
Function main(myEvent)
    myString = "Handling Event: " & myEvent.EventType
    myString = myString & vbCr & vbCr & "Target: " & myEvent.Target & " " &
    myEvent.Target.Name
    myString = myString & vbCr & "Current: " & myEvent.CurrentTarget & " " &
    myEvent.CurrentTarget.Name
    myString = myString & vbCr & vbCr & "Phase: " &
    myGetPhaseName(myEvent.EventPhase)
    myString = myString & vbCr & "Captures: " & myEvent.Captures
    myString = myString & vbCr & "Bubbles: " & myEvent.Bubbles
    myString = myString & vbCr & vbCr & "Cancelable: " & myEvent.Cancelable
    myString = myString & vbCr & "Stopped: " & myEvent.PropagationStopped
    myString = myString & vbCr & "Canceled: " & myEvent.DefaultPrevented
    myString = myString & vbCr & vbCr & "Time: " & myEvent.TimeStamp
    MsgBox myString, vbOKOnly, "Event Details"
end function
Rem Function returns a string corresponding to the event phase enumeration.
Function myGetPhaseName(myEventPhase)
    Select Case myEventPhase
        Case idEventPhases.idAtTarget
            myPhaseName = "At Target"
        Case idEventPhases.idBubblingPhase
            myPhaseName = "Bubbling"
        Case idEventPhases.idCapturingPhase
            myPhaseName = "Capturing"
        Case idEventPhases.idDone
            myPhaseName = "Done"
        Case idEventPhases.idNotDispatching
            myPhaseName = "Not Dispatching"
    end select
    myGetPhaseName = myPhaseName
End Function
```

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

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
For myCounter = 1 To myInDesign.EventListeners.Count
    myInDesign.EventListeners.Item(1).Delete
Next
```

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).

```
Rem Adds an event listener to the afterNew event. Calls
Rem a script on disk to set up basic document parameters
Rem and XMP metadata.
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myEventListener = myInDesign.EventListeners.Add("afterNew",
"c:\IDEventHandlers\AfterNewHandler.vbs", False)
```

The following script is the one referred to by the above script. The file reference in the script above must match the location of this script on your disk. For the complete script, see `AfterNewHandler.vbs`.

```
Rem AfterNewHandler.vbs
Rem An InDesign CS4 VBScript
Rem
Rem This script is called by the AfterNew.vbs tutorial script. It
Rem Sets up a basic document layout and adds XMP information
Rem to the document.
AfterNewHandler(evt)
Function AfterNewHandler(myEvent)
    Set myInDesign = CreateObject("InDesign.Application.CS4")
    Set myDocument = myEvent.Parent
    Set myViewPreferences = myDocument.ViewPreferences
    myViewPreferences.HorizontalMeasurementUnits = idMeasurementUnits.idPoints
    myViewPreferences.VerticalMeasurementUnits = idMeasurementUnits.idPoints
    myViewPreferences.RulerOrigin = idRulerOrigin.idPageOrigin
    Rem mySlugOffset is the distance from the bottom of the
    Rem page to the top of the slug.
    mySlugOffset = 24
    Rem mySlugHeight is the height of the slug text frame.
    mySlugHeight = 72
    With myDocument.DocumentPreferences
        .SlugBottomOffset = mySlugOffset + mySlugHeight
        .SlugTopOffset = 0
        .SlugInsideOrLeftOffset = 0
        .SlugRightOrOutsideOffset = 0
    End With
    For myCounter = 1 To myDocument.MasterSpreads.Count
        Set myMasterSpread = myDocument.MasterSpreads.Item(myCounter)
        For myMasterPageCounter = 1 To myMasterSpread.Pages.Count
            Set myPage = myMasterSpread.Pages.Item(myMasterPageCounter)
            mySlugBounds = myGetSlugBounds(myDocument, myPage, mySlugOffset,
            mySlugHeight)
            Set mySlugFrame = myPage.TextFrames.Add
            mySlugFrame.GeometricBounds = mySlugBounds
            mySlugFrame.Contents = "Created: " & myEvent.TimeStamp & vbCr & "by: "
            & myInDesign.UserName
        Next
    Next
    With myDocument.MetadataPreferences
        .Author = "Adobe Systems"
        .Description = "This is a sample document with XMP metadata."
    End With
End Function
Function myGetSlugBounds(myDocument, myPage, mySlugOffset, mySlugHeight)
    myPageWidth = myDocument.DocumentPreferences.PageWidth
    myPageHeight = myDocument.DocumentPreferences.PageHeight
    myX1 = myPage.MarginPreferences.Left
    myY1 = myPageHeight + mySlugOffset
    myX2 = myPageWidth - myPage.MarginPreferences.Right
    myY2 = myY1 + mySlugHeight
    myGetSlugBounds = Array(myY1, myX1, myY2, myX2)
End Function
```

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):

```
Rem Adds an event listener that performs a preflight check on
Rem a document before printing. If the preflight check fails,
Rem the script gives the user the opportunity to cancel the print job.
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myEventListener = myInDesign.EventListeners.Add("beforePrint",
"c:\IDEventHandlers\BeforePrintHandler.vbs", False)
```

The following script is the one referred to by the above script. The file reference in the script above must match the location of this script on your disk. For the complete script, see `BeforePrintHandler.vbs`.

```
Rem BeforePrintHandler.vbs
Rem An InDesign CS4 VBScript
Rem
Rem Peforms a preflight check on a document. Called by the
Rem BeforePrint.applescript event listener example.
Rem "evt" is the event passed to this script by the event listener.
myBeforePrintHandler(evt)
Function myBeforePrintHandler(myEvent)
    Rem The parent of the event is the document.
    Set myDocument = myEvent.parent
    If myPreflight(myDocument) = False Then
        myEvent.stopPropagation
        myEvent.preventDefault
        myString = "Document did not pass preflight check." & vbCr
        myString = myString & "Please fix the problems and try again."
        msgbox(myString)
    Else
        msgbox("Document passed preflight check. Ready to print.")
        myDocument.print(true)
    End If
End Function
Function myPreflight(myDocument)
    myPreflightCheck = True
    myFontCheck = myCheckFonts(myDocument)
    myGraphicsCheck = myCheckGraphics(myDocument)
    If ((myFontCheck = false)Or(myGraphicsCheck = false)) Then
        myPreflightCheck = false
    End If
    myPreflight = myPreflightCheck
End function
Function myCheckFonts(myDocument)
    myFontCheck = true
    For myCounter = 1 To myDocument.fonts.count
        Set myFont = myDocument.fonts.item(myCounter)
        if myFont.status <> idFontStatus.idinstalled Then
            myFontCheck = false
        End If
    Next
    myCheckFonts = myFontCheck
End function
```

```
function myCheckGraphics(myDocument)
    myGraphicsCheck = true
    for myCounter = 1 To myDocument.allGraphics.count
        set myGraphic = myDocument.allGraphics.item(myCounter)
        If myGraphic.itemLink.status <> idLinkStatus.idnormal Then
            myGraphicsCheck = false
        End If
    Next
    myCheckGraphics = myGraphicsCheck
End function
```

8 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 CS4 Scripting Tutorial* and know how to create, install, and run a script.

Understanding the menu model

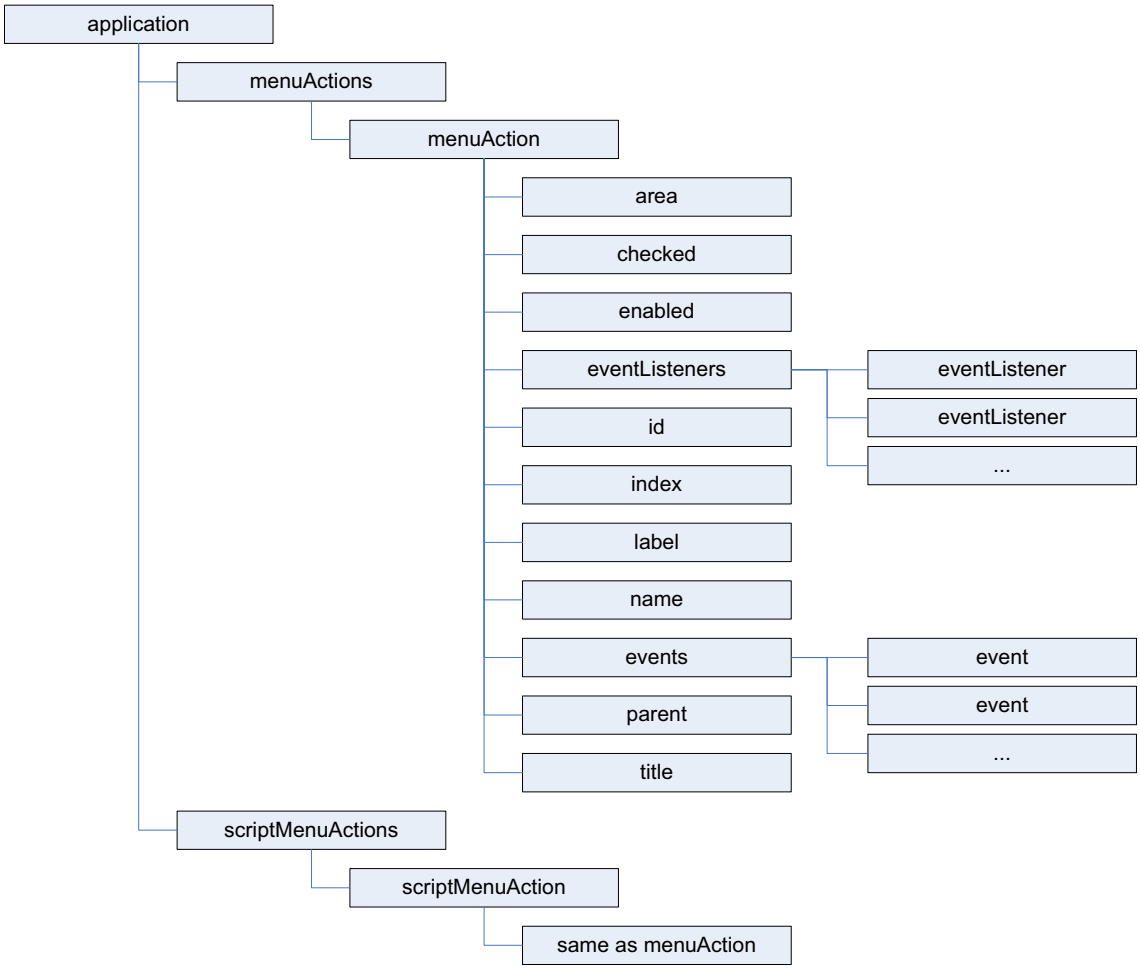
The InDesign menu-scripting model is made up of a series of objects that correspond to the menus you see in the application's user interface, including menus associated with panels as well as those displayed on the main menu bar. A `menu` object contains the following objects:

- `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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
Rem You'll need to fill in a valid file path on your system.
Set myTextFile = myFileSystemObject.CreateTextFile("c:\menuactions.txt", True, False)
For myCounter = 1 To myInDesign.MenuActions.Count
    Set myMenuAction = myInDesign.MenuActions.Item(myCounter)
    myTextFile.WriteLine myMenuAction.name
Next
myTextFile.Close
MsgBox "done!"
```

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.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
Set myTextFile = myFileSystemObject.CreateTextFile("c:\menunames.txt", True, False)
For myMenuCounter = 1 To myInDesign.Menus.Count
    Set myMenu = myInDesign.Menus.Item(myMenuCounter)
    myTextFile.WriteLine myMenu.Name
    myProcessMenu myMenu, myTextFile
Next
myTextFile.Close
MsgBox "done!"
Function myProcessMenu(myMenuItem, myTextFile)
    myString = ""
    myMenuName = myMenuItem.Name
    For myCounter = 1 To myMenuItem.MenuElements.Count
        If TypeName(myMenuItem.MenuElements.Item(myCounter)) <>
        "MenuSeparator" Then
            myString = myGetIndent(myMenuItem.MenuElements.Item(myCounter),
            myString, False)
            myTextFile.WriteLine myString &
            myMenuItem.MenuElements.Item(myCounter).Name
            myMenuElementName = myMenuItem.MenuElements.Item(myCounter).Name
            myString = ""
            If TypeName(myMenuItem.MenuElements.Item(myCounter)) = "Submenu" Then
                If myMenuItem.MenuElements.Count > 0 Then
                    myProcessMenu myMenuItem.MenuElements.Item(myCounter),
                    myTextFile
                End If
            End If
        End If
    Next
End Function
Function myGetIndent(myMenuItem, myString, myDone)
    Do While myDone = False
        If TypeName(myMenuItem.Parent) = "Application" Then
            myDone = True
        Else
            myString = myString & vbTab
            myGetIndent myMenuItem.Parent, myString, myDone
        End If
    Loop
    myGetIndent = myString
End Function
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem Fill in the name of the menu action you want.
Set myMenuAction = myInDesign.MenuActions.Item("Convert to Note")
myKeyStrings = myInDesign.FindKeyStrings(myMenuAction.Name)
myString = ""
For Each myKeyString In myKeyStrings
    myString = myString & myKeyString & vbCr
Next
MsgBox myString
```

NOTE: It is much better to get the locale-independent name of a `MenuAction` than of a `Menus`, `MenuItem`, or `Submenu`, because the title of a `MenuAction` is more likely to be a single string. Many of the other menu objects return multiple strings when you use the `FindKeyStrings` method.

Once you have the locale-independent string you want to use, you can include it in your scripts. Scripts that use these strings will function properly in locales other than that of your version of InDesign.

To translate a locale-independent string into the current locale, use the following script fragment (from the TranslateKeyString tutorial script):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem Fill in the appropriate key string in the following line.
myString = myInDesign.TranslateKeyString("$ID/NotesMenu.ConvertToNote")
MsgBox 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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem Get a reference to a menu action.
Set myMenuAction = myInDesign.MenuActions.Item("$ID/NotesMenu.ConvertToNote")
Rem Run the menu action. The example action will fail if you do not
Rem have text selected.
myMenuAction.Invoke
```

NOTE: In general, you should not try to automate InDesign processes by scripting menu actions and user-interface selections; InDesign's scripting object model provides a much more robust and powerful way to work. Menu actions depend on a variety of user-interface conditions, like the selection and the state of the window. Scripts using the object model work with the objects in an InDesign document directly, which means they do not depend on the user interface; this, in turn, makes them faster and more consistent.

Adding menus and menu items

Scripts also can create new menus and menu items or remove menus and menu items, just as you can in the InDesign user interface. The following sample script shows how to duplicate the contents of a submenu to a new menu in another menu location (for the complete script, see CustomizeMenu):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myMainMenu = myInDesign.Menus.Item("Main")
Set myTypeMenu = myMainMenu.MenuElements.Item("Type")
Set myFontMenu = myTypeMenu.MenuElements.Item("Font")
Set myKozukaMenu = myFontMenu.Submenus.Item("Kozuka Mincho Pro ")
Set mySpecialFontMenu = myMainMenu.Submenus.Add("Kozuka Mincho Pro")
For myCounter = 1 To myKozukaMenu.MenuItems.Count
    Set myAssociatedMenuAction =
myKozukaMenu.MenuItems.Item(myCounter).AssociatedMenuAction
    mySpecialFontMenu.MenuItems.Add myAssociatedMenuAction
Next
```

To remove the custom menu item created by the above script, use RemoveCustomMenu.

```
Set myMainMenu = myInDesign.menus.item("$ID/Main")
On Error Resume Next
Set mySpecialFontMenu = myMainMenu.Submenus.Item("Kozuka Mincho Pro")
mySpecialFontMenu.Delete
On Error Goto 0
```

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 7, “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.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set mySampleScriptAction = myInDesign.ScriptMenuActions.Add("Display Message")
Set myEventListener = mySampleScriptAction.EventListeners.Add("onInvoke",
"c:\message.vbs")
Set mySampleScriptMenu = myInDesign.Menus.Item("$ID/Main").Submenus.Add("Script Menu
Action")
Set mySampleScriptMenuItem = mySampleScriptMenu.MenuItems.Add(mySampleScriptAction)
```

The `message.vbs` script file contains the following code:

```
    MsgBox("You selected an example script menu action.")
```

To remove the `Menu`, `Submenu`, `MenuItem`, and `ScriptMenuAction` created by the above script, run the following script fragment (from the RemoveScriptMenuAction tutorial script):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set mySampleScriptAction = myInDesign.ScriptMenuActions.Item("Display Message")
mySampleScriptAction.Delete
Set mySampleScriptMenu = myInDesign.Menus.Item("$ID/Main").Submenus.Item("Script Menu
Action")
mySampleScriptMenu.Delete
```

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.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
For myCounter = myInDesign.ScriptMenuActions.Count To 1 Step -1
    myInDesign.ScriptMenuActions.Item(myCounter).Delete
Next
```

You can create a list of all current `ScriptMenuActions`, as shown in the following script fragment (from the ListScriptMenuActions tutorial script):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
Rem You'll need to fill in a valid file path for your system.
Set myTextFile = myFileSystemObject.CreateTextFile("c:\scriptmenuactionnames.txt",
True, False)
For myCounter = 1 To myInDesign.ScriptMenuActions.Count
    Set myScriptMenuAction = myInDesign.ScriptMenuActions.Item(myMenuCounter)
    myTextFile.WriteLine myScriptMenuAction.Name
Next
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.)

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set mySampleScriptAction = myInDesign.ScriptMenuActions.Add("Display Message")
Set myEventListener = mySampleScriptAction.EventListeners.Add("onInvoke",
"c:\WhatIsSelected.vbs ")
Set mySampleScriptMenu = myInDesign.Menus.Item("$ID/Main").Submenus.Add("Script Menu
Action")
Set mySampleScriptMenuItem = mySampleScriptMenu.MenuItems.Add(mySampleScriptAction)
mySampleScriptMenu.EventListeners.Add "beforeDisplay", "c:\BeforeDisplayHandler.vbs"
```

The BeforeDisplayHander tutorial script file contains the following script:

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set mySampleScriptAction = myInDesign.ScriptMenuActions.Item("Display Message")
If myInDesign.Selection.Count > 0 Then
    mySampleScriptAction.Enabled = True
Else
    mySampleScriptAction.Enabled = False
End If
```

The WhatIsSelected tutorial script file contains the following script:

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
myString = TypeName(myInDesign.Selection.Item(1))
MsgBox "The first item in the selection is a " & myString & "."
```

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.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Rem The locale-independent name (aka "key string") for the
Rem Layout context menu is "$ID/RtMouseLayout".
Set myLayoutContextMenu = myInDesign.Menus.Item("$ID/RtMouseLayout")
Rem Create the event handler for the "beforeDisplay" event
Rem of the Layout context menu.
Set myBeforeDisplayListener = myLayoutContextMenu.addEventListener("beforeDisplay",
"c:\IDEventHandlers\LabelGraphicBeforeDisplay.vbs", false)
```

The `LabelGraphicBeforeDisplay.vbs` file referred to in the above example contains the following:

```
myBeforeDisplayHandler evt
function myBeforeDisplayHandler(myEvent)
    ReDim myObjectList(0)
    Set myInDesign = CreateObject("InDesign.Application.CS4")
    Set myLayoutContextMenu = myInDesign.Menus.Item("$ID/RtMouseLayout")
    Rem Check for open documents is a basic sanity check--
    Rem it should never be needed, as this menu won't be
    Rem displayed unless an item is selected. But it's best
    Rem to err on the side of safety.
    If myInDesign.Documents.Count > 0 Then
        If myInDesign.Selection.Count > 0 Then
            Rem Does the selection contain any graphics?
            for myCounter = 1 To myInDesign.Selection.Count
                Select Case TypeName(myInDesign.Selection.Item(myCounter))
                    Case "PDF","EPS", "Image":
                        If Not (IsEmpty(myObjectList(0))) Then
                            ReDim Preserve myObjectList(UBound(myObjectList) + 1)
                        End If
                        Set myObjectList(UBound(myObjectList)) =
                            myInDesign.Selection.Item(myCounter)
                    Case "Rectangle", "Oval", "Polygon":
                        If myInDesign.selection.Item(myCounter).
                        Graphics.Count > 0 Then
                            If Not (IsEmpty(myObjectList(0))) Then
                                ReDim Preserve myObjectList(UBound(myObjectList) + 1)
                            End If
                            Set myObjectList(UBound(myObjectList)) =
                            myInDesign.Selection.Item(myCounter).Graphics.Item(1)
                        End If
                End Select
            Next
            If Not (IsEmpty(myObjectList(0))) Then
                Rem Add the menu item if it does not already exist.
                If myCheckForMenuItem(myLayoutContextMenu,
                "Create Graphic Label") = False Then
                    myMakeLabelGraphicMenuItem myInDesign
                End If
            Else
                Rem Remove the menu item, if it exists.
                If myCheckForMenuItem(myLayoutContextMenu,
                "Create Graphic Label") = True Then
                    myLayoutContextMenu.MenuItems.Item("Create Graphic
                    Label").delete
                End If
            End If
        End If
    End If
End Function
```

```
Function myMakeLabelGraphicMenuItem(myInDesign)
    Rem alert("Got to the myMakeLabelGraphicMenuItem function!")
    If myCheckForScriptMenuItem(myInDesign, "Create Graphic Label") =
    False Then
        msgbox "Making a new script menu action!"
        Set myLabelGraphicMenuAction = myInDesign.ScriptMenuActions.add("Create
        Graphic Label")
        Set myLabelGraphicEventListener = myLabelGraphicMenuAction.
        EventListeners.Add("onInvoke", "c:\IDEventHandlers\
        LabelGraphicOnInvoke.vbs", false)
    End If
    Set myLabelGraphicMenuItem = myInDesign.Menus.Item("$ID/RtMouseLayout").
    menuItems.add(myInDesign.scriptMenuActions.item("Create Graphic Label"))
End Function
```

The `LabelGraphicOnInvoke.vbs` referred to in the above example defines the `script menu action` that is activated when the `menu item` is selected (`onInvoke` event):

```
myLabelGraphicEventHandler evt
Function myLabelGraphicEventHandler(myEvent)
    ReDim myObjectList(0)
    Set myInDesign = CreateObject("InDesign.Application.CS4")
    If myInDesign.Selection.Count > 0 Then
        Rem Does the selection contain any graphics?
        for myCounter = 1 To myInDesign.Selection.Count
            Select Case TypeName(myInDesign.Selection.Item(myCounter))
                Case "PDF","EPS", "Image":
                    If Not (IsEmpty(myObjectList(0))) Then
                        ReDim Preserve myObjectList(UBound(myObjectList) + 1)
                    End If
                    Set myObjectList(UBound(myObjectList)) =
                    myInDesign.Selection.Item(myCounter)
                Case "Rectangle", "Oval", "Polygon":
                    If myInDesign.selection.Item(myCounter).Graphics.Count > 0
                    Then
                        If Not (IsEmpty(myObjectList(0))) Then
                            ReDim Preserve myObjectList(UBound(myObjectList) + 1)
                        End If
                        Set myObjectList(UBound(myObjectList)) = myInDesign.
                        Selection.Item(myCounter).Graphics.Item(1)
                    End If
            End Select
        Next
        If Not (IsEmpty(myObjectList(0))) Then
            myDisplayDialog myInDesign, myObjectList
        End If
    End If
End Function
Rem Function that adds the label.
Function myAddLabel(myInDesign, myGraphic, myLabelType, myLabelHeight, myLabelOffset,
myLabelStyleName, myLayerName)
    Set myDocument = myInDesign.documents.Item(1)
    myLabelStyle = myDocument.paragraphStyles.item(myLabelStyleName)
    Set myLink = myGraphic.ItemLink
    Rem Create the label layer if it does not already exist.
    On Error Resume Next
    Set myLabelLayer = myDocument.layers.item(myLayerName)
```

```
    If Err.Number <> 0 Then
        Set myLabelLayer = myDocument.Layers.Add
        myLabelLayer.Name = myLayerName
        Err.Clear
    End If
    On Error Goto 0
    Rem Label type defines the text that goes in the label.
    Select Case myLabelType
        Rem File name
        case 0:
            myLabel = myLink.Name
        Rem File path
        case 1:
            myLabel = myLink.FilePath
        Rem XMP description
        case 2:
            On Error Resume Next
                myLabel = myLink.LinkXmp.Description
            If Err.Number <> 0 Then
                myLabel = "No description available."
                Err.Clear
            End If
            On Error Goto 0
        Rem XMP author
        case 3:
            On Error Resume Next
            myLabel = myLink.LinkXmp.Author
            If Err.Number <> 0 Then
                myLabel = "No author available."
                Err.Clear
            End If
            On Error Goto 0
    End Select
    Set myFrame = myGraphic.Parent
    myBounds = myFrame.GeometricBounds
    myX1 = myBounds(1)
    myY1 = myBounds(2) + myLabelOffset
    myX2 = myBounds(3)
    myY2 = myY1 + myLabelHeight
    Set myTextFrame = myFrame.Parent.TextFrames.Add(myLabelLayer)
    myTextFrame.GeometricBounds = Array(myY1, myX1, myY2, myX2)
    myTextFrame.Contents = myLabel
    myTextFrame.TextFramePreferences.FirstBaselineOffset =
    idFirstBaseline.idLeadingOffset
    myTextFrame.Paragraphs.Item(1).AppliedParagraphStyle =
    myInDesign.Documents.Item(1).ParagraphStyles.Item(myLabelStyle)
End Function
Function myDisplayDialog(myInDesign, myObjectList)
    myLabelWidth = 100
    myStyleNames = myGetParagraphStyleNames(myInDesign.Documents.Item(1))
    myLayerNames = myGetLayerNames(myInDesign.Documents.Item(1))
    Set myDialog = myInDesign.Dialogs.Add
    myDialog.Name = "LabelGraphics"
```

```
With myDialog.DialogColumns.Add
    Rem Label type
    With .DialogRows.Add
        With .DialogColumns.Add
            With .StaticTexts.add
                .StaticLabel = "Label Type"
                .MinWidth = myLabelWidth
            End With
        End With
        With .DialogColumns.Add
            Set myLabelTypeDropdown = .Dropdowns.Add
            myLabelTypeDropdown.StringList = Array("File name", "File path",
            "XMP description", "XMP author")
            myLabelTypeDropdown.SelectedIndex = 0
        End With
    End With
    Rem Text frame height
    With .DialogRows.Add
        With .DialogColumns.Add
            With .StaticTexts.add
                .StaticLabel="Label Height"
                .MinWidth=myLabelWidth
            End With
        End With
        With .DialogColumns.Add
            Set myLabelHeightField = .MeasurementEditboxes.Add
            myLabelHeightField.EditValue = 24
            myLabelHeightField.EditUnits = idMeasurementUnits.idPoints
        End With
    End With
    Rem Text frame offset
    With .DialogRows.Add
        With .DialogColumns.Add
            With .staticTexts.add
                .staticLabel="Label Offset"
                .minWidth=myLabelWidth
            End With
        End With
        With .DialogColumns.Add
            Set myLabelOffsetField = .MeasurementEditboxes.Add
            myLabelOffsetField.editValue=0
            myLabelOffsetField.editUnits=idMeasurementUnits.idPoints
        End With
    End With
    Rem Style to apply
    With .DialogRows.Add
        With .DialogColumns.Add
            With .StaticTexts.Add
                .StaticLabel="Label Style"
                .MinWidth=myLabelWidth
            End With
        End With
        With .DialogColumns.Add
            Set myLabelStyleDropdown = .Dropdowns.Add
            myLabelStyleDropdown.StringList=myStyleNames
            myLabelStyleDropdown.SelectedIndex=0
        End With
    End With
```

```
        Rem Layer
        With .DialogRows.Add
            With .DialogColumns.Add
                With .StaticTexts.Add
                    .StaticLabel="Layer"
                    .MinWidth=myLabelWidth
                End With
            End With
            With .DialogColumns.Add
                Set myLayerDropdown = .Dropdowns.Add
                myLayerDropdown.StringList=myLayerNames
                myLayerDropdown.SelectedIndex=0
            End With
        End With
    End With
    myResult = myDialog.show
    If myResult = True Then
        myLabelType = myLabelTypeDropdown.selectedIndex
        myLabelHeight = myLabelHeightField.editValue
        myLabelOffset = myLabelOffsetField.editValue
        myLabelStyle = myStyleNames(myLabelStyleDropdown.selectedIndex)
        myLayerName = myLayerNames(myLayerDropdown.selectedIndex)
        myDialog.Destroy
        myOldXUnits = myInDesign.documents.item(1).viewPreferences.
        horizontalMeasurementUnits
        myOldYUnits = myInDesign.documents.item(1).viewPreferences.
        verticalMeasurementUnits
        myInDesign.documents.item(1).viewPreferences.horizontalMeasurementUnits
        = idMeasurementUnits.idPoints
        myInDesign.documents.item(1).viewPreferences.verticalMeasurementUnits =
        idMeasurementUnits.idPoints
        for myCounter = 0 To UBound(myObjectList)
            Set myGraphic = myObjectList(myCounter)
            myAddLabel myInDesign, myGraphic, myLabelType, myLabelHeight,
            myLabelOffset, myLabelStyle, myLayerName
        Next
        myInDesign.documents.item(1).viewPreferences.horizontalMeasurementUnits
        = myOldXUnits
        myInDesign.documents.item(1).viewPreferences.verticalMeasurementUnits =
        myOldYUnits
    Else
        myDialog.Destroy
    End If
End Function
```

9 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 CS4 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 10, "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 137.

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):

```
Set myXMLViewPreferences = myDocument.XMLViewPreferences
myXMLViewPreferences.ShowAttributes = True
myXMLViewPreferences.ShowStructure = True
myXMLViewPreferences.ShowTaggedFrames = True
myXMLViewPreferences.ShowTagMarkers = True
myXMLViewPreferences.ShowTagOptions = 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):

```
Set myXMLPreferences = myDocument.XMLPreferences
myXMLPreferences.DefaultCellTagColor = idUIColors.idBlue
myXMLPreferences.DefaultCellTagName = "cell"
myXMLPreferences.DefaultImageTagColor = idUIColors.idBrickRed
myXMLPreferences.DefaultImageTagName = "image"
myXMLPreferences.DefaultStoryTagColor = idUIColors.idCharcoal
myXMLPreferences.DefaultStoryTagName = "text"
myXMLPreferences.DefaultTableTagColor = idUIColors.idCuteTeal
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):

```
Set myXMLImportPreferences = myDocument.XMLImportPreferences
myXMLImportPreferences.AllowTransform = False
myXMLImportPreferences.CreateLinkToXML = False
myXMLImportPreferences.IgnoreUnmatchedIncoming = True
myXMLImportPreferences.IgnoreWhitespace = True
myXMLImportPreferences.ImportCALSTables = True
myXMLImportPreferences.ImportStyle = idXMLImportStyles.idMergeImport
myXMLImportPreferences.ImportTextIntoTables = False
myXMLImportPreferences.ImportToSelected = False
myXMLImportPreferences.RemoveUnmatchedExisting = False
myXMLImportPreferences.RepeatTextElements = True
Rem The following properties are only used when the
Rem AllowTransform property is set to True.
Rem myXMLImportPreferences.TransformFilename = "c:\myTransform.xsl"
Rem If you have defined parameters in your XSL file, then you can pass
Rem parameters to the file during the XML import process. For each parameter,
Rem enter an array containing two strings. The first string is the name of the
Rem parameter, the second is the value of the parameter.Rem
myXMLImportPreferences.TransformParameters = Array(Array("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 "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 "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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
myDocument.ImportXML "c:\test.xml"
Set myRootXMLElement = myDocument.XMLElements.Item(1)
Set myLastXMLElement = myRootXMLElement.XMLElements.Item(-1)
Rem Select the XML element
myDocument.Select myLastXMLElement, idSelectionOptions.idReplaceWith
myDocument.XMLImportPreferences.ImportToSelected = True
myDocument.ImportXML "c:\test.xml"
Set myTextFrame = myDocument.Pages.Item(1).TextFrames.Add
myTextFrame.GeometricBounds = myGetBounds(myDocument, myDocument.Pages.Item(1))
myRootXMLElement.PlaceXML myTextFrame
```

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):

```
Rem You can create an XML tag without specifying a color for the tag.
Set myXMLTagA = myDocument.XMLTags.Add("XML_tag_A")
Rem You can define the highlight color of the XML tag using the UIColors enumeration...
Set myXMLTagB = myDocument.XMLTags.Add("XML_tag_B", UIColors.Gray)
Rem ...or you can provide an RGB array to set the color of the tag.
Set myXMLTagC = myDocument.XMLTags.Add("XML_tag_C", Array(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("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("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):

```
Set myXMLTagA = myDocument.XMLTags.Add("XML_tag_A")
Set myXMLElementA = myDocument.XMLElements.Item(1).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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
Set myXMLTagA = myDocument.XMLTags.Add("myXMLTagA")
Set myXMLTagB = myDocument.XMLTags.Add("myXMLTagB")
Set myXMLElementA = myDocument.XMLElements.Item(1).XMLElements.Add(myXMLTagA)
myXMLElementA.Contents = "This is XML element A."
Set myXMLElementB = myDocument.XMLElements.Item(1).XMLElements.Add(myXMLTagB)
myXMLElementB.Contents = "This is XML element B."
myXMLElementA.Move idLocationOptions.idAfter, myXMLElementB
```

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(1).Delete
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
Set myXMLTagA = myDocument.XMLTags.Add("myXMLTagA")
Set myXMLTagB = myDocument.XMLTags.Add("myXMLTagB")
Set myXMLElementA = myDocument.XMLElements.Item(1).XMLElements.Add(myXMLTagA)
myXMLElementA.Contents = "This is XML element A."
Set myXMLElementB = myDocument.XMLElements.Item(1).XMLElements.Add(myXMLTagB)
myXMLElementB.Contents = "This is XML element B."
myXMLElementA.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.)

```
Set myXMLElement = myDocument.XMLElements.item(1).xmlElements.item(1)
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):

```
Set myRootXMLElement = myDocument.XMLElements.item(1)
Set myXMLElementB = myRootXMLElement.xmlElements.item(2)
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):

```
Set myRootXMLElement = myDocument.XMLElements.item(1)
Set 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").

```
Set myDocument = myInDesign.Documents.Item(1)
Set myRootXMLElement = myDocument.XMLElements.item(1)
Set myXMLElementB = myRootXMLElement.xmlElements.item(2)
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):

```
Set myDocument = myInDesign.Documents.Item(1)
Set myRootXMLElement = myDocument.XMLElements.item(1)
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):

```
Set myRootXMLElement = myDocument.XMLElements.item(1)
Set myXMLElementB = myRootXMLElement.xmlElements.item(1)
Rem The "at" parameter can be either idLocationOptions.idAtEnd or
Rem idLocationOptions.idAtBeginning, but cannot
Rem be idLocationOptions.idAfter or idLocationOptions.idBefore.
myXMLElementB.XMLAttributes.item(1).convertToElement idLocationOptions.idAtEnd,
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):

```
Set myXMLStory = myDocument.XmlStories.Item(1)
Rem Though the text has not yet been placed in the layout,
Rem all text properties are available.
myXMLStory.Paragraphs.Item(1).PointSize = 72
Rem Place the XML element in the layout to see the result.
myDocument.XMLElements.Item(1).XMLElements.Item(1).PlaceXML
myDocument.Pages.Item(1).TextFrames.Item(1)
```

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):

```
Rem Export the entire XML structure in the document.
myDocument.Export idExportFormat.idXML, "c:\completeDocumentXML.xml"
Rem Export a specific XML element and its child XML elements.
Set myXMLElement = myDocument.XMLElements.Item(1).XMLElements.Item(-1)
myXMLElement.Export idExportFormat.idXML, "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(1).XMLElements.Item(2)
myDocument.XMLExportPreferences.ExportFromSelected = True
Rem Export the entire XML structure in the document.
myDocument.Export idExportFormat.idXML, "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(1).placeXML myDocument.pages.item(1).textFrames.item(1)
```

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(1).XMLElements.item(0).markup
myDocument.pages.item(1).textFrames.item(1)
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add()
myDocument.ViewPreferences.HorizontalMeasurementUnits = idMeasurementUnits.idPoints
myDocument.ViewPreferences.VerticalMeasurementUnits = idMeasurementUnits.idPoints
myDocument.ViewPreferences.RulerOrigin = idRulerOrigin.idPageOrigin
Rem PlaceIntoFrame has two parameters:
Rem On: The page, spread, or master spread on which to create the frame
Rem GeometricBounds: The bounds of the new frame (in page coordinates).
myDocument.XMLElements.Item(1).XMLElements.Item(1).PlaceIntoFrame
myDocument.Pages.Item(1), Array(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):

```
Set myPage = myDocument.Pages.Item(1)
Set myXMLElement = myDocument.XMLElements.Item(1)
myXMLElement.PlaceIntoCopy myPage, Array(288, 72), myPage.TextFrames.Item(1), 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):

```
Set myTextFrame = myDocument.Pages.Item(1).TextFrames.Add
myTextFrame.GeometricBounds = myGetBounds(myDocument, myDocument.Pages.Item(1))
myTextFrame.InsertionPoints.Item(-1).Contents = vbCr & vbCr
myDocument.XMLElements.Item(1).PlaceXML myTextFrame
Set myXMLElement = myDocument.XMLElements.Item(1).XMLElements.Add(myXMLTag)
myXMLElement.Contents = "This is the second XML element."
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):

```
Set myXMLElement = myDocument.XMLElements.Item(1).XMLElements.Item(3)
Rem Specify width and height as you create the inline frame.
myXMLElement.PlaceIntoInlineFrame Array(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):

```
Rem Shows how to add text before, after, and at the beginning/end
Rem of XML elements.
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Add
Set myRootXMLElement = myDocument.XMLElements.Item(1)
Set myXMLTag = myDocument.XMLTags.Add("myXMLElement")
Set myXMLElementA = myRootXMLElement.XMLElements.Add(myXMLTag)
myXMLElementA.Contents = "This is a paragraph in an XML story."
Set myXMLElementB = myRootXMLElement.XMLElements.Add(myXMLTag)
myXMLElementB.Contents = "This is another paragraph in an XML story."
Set myXMLElementC = myRootXMLElement.XMLElements.Add(myXMLTag)
myXMLElementC.Contents = "This is the third paragraph in an example XML story."
Set myXMLElementD = myRootXMLElement.XMLElements.Add(myXMLTag)
myXMLElementD.Contents = "This is the last paragraph in the XML story."
Set myXMLElement = myDocument.XMLElements.Item(1).XMLElements.Item(1)
Rem By inserting the return character after the XML element, the character
Rem becomes part of the content of the parent XML element, not of the element itself.
myXMLElement.InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
Set myXMLElement = myDocument.XMLElements.Item(1).XMLElements.Item(2)
myXMLElement.InsertTextAsContent "Static text: ", idXMLElementPosition.idBeforeElement
myXMLElement.InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
Rem To add text inside the element, set the location option to beginning or end.
Set myXMLElement = myDocument.XMLElements.Item(1).XMLElements.Item(3)
myXMLElement.InsertTextAsContent "Text at the start of the element: ",
idXMLElementPosition.idElementStart
myXMLElement.InsertTextAsContent " Text at the end of the element.",
idXMLElementPosition.idElementEnd
myXMLElement.InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
Rem Add static text outside the element.
Set myXMLElement = myDocument.XMLElements.Item(1).XMLElements.Item(4)
myXMLElement.InsertTextAsContent "Text before the element: ",
idXMLElementPosition.idBeforeElement
myXMLElement.InsertTextAsContent " Text after the element.",
idXMLElementPosition.idAfterElement
Rem To insert text inside the text of an element, work with the text objects contained
by the element.
myXMLElement.Words.Item(2).InsertionPoints.Item(1).Contents = "(the third word of) "
Set myStory = myDocument.Stories.Item(1)
myRootXMLElement.PlaceXML (myStory)
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Item(1)
Rem 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")
Rem Apply the tag to style mapping.
myDocument.MapXMLTagsToStyles
Set myTextFrame = myDocument.TextFrames.Add
myTextFrame.GeometricBounds = myGetBounds(myDocument, myDocument.Pages.Item(1))
Rem Associate the root XML element with the text frame
Rem so that you can see the effect of the tag to style mapping.
myDocument.XMLElements.Item(1).PlaceXML myDocument.Pages.Item(1).TextFrames.Item(1)
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Item(1)
Rem Create a tag to style 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")
Rem 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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Item(1)
Rem Create tags that match the style names in the document,
Rem creating an XMLExportMap for each tag/style pair.
For myCounter = 1 To myDocument.ParagraphStyles.Count
    Set myParagraphStyle = myDocument.ParagraphStyles.Item(myCounter)
    myParagraphStyleName = myParagraphStyle.Name
    myXMLTagName = Replace(myParagraphStyleName, " ", "_")
    myXMLTagName = Replace(myXMLTagName, "[", "")
    myXMLTagName = Replace(myXMLTagName, "]", "")
    Set myXMLTag = myDocument.XMLTags.Add(myXMLTagName)
    myDocument.XMLExportMaps.Add myParagraphStyle, myXMLTag
Next
Rem Apply the tag to style 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):

```
Set myXMLTag = myDocument.XMLTags.Add("graphic")
Set myGraphic = myDocument.Pages.Item(1).Place("c:\test.tif")
Rem Associate the graphic with a new XML element as you create the element.
Set myXMLElement = myDocument.XMLElements.Item(1).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.)

```
Rem Create a series of XML tags.
Set myHeading1XMLTag = myDocument.XMLTags.Add("heading_1")
Set myHeading2XMLTag = myDocument.XMLTags.Add("heading_2")
Set myPara1XMLTag = myDocument.XMLTags.Add("para_1")
Set myBodyTextXMLTag = myDocument.XMLTags.Add("body_text")
Rem Create a series of paragraph styles.
Set myHeading1Style = myDocument.ParagraphStyles.Add
myHeading1Style.Name = "heading 1"
myHeading1Style.PointSize = 24
Set myHeading2Style = myDocument.ParagraphStyles.Add
myHeading2Style.Name = "heading 2"
myHeading2Style.PointSize = 14
myHeading2Style.SpaceBefore = 12
Set myPara1Style = myDocument.ParagraphStyles.Add
myPara1Style.Name = "para 1"
myPara1Style.PointSize = 12
myPara1Style.FirstLineIndent = 0
Set myBodyTextStyle = myDocument.ParagraphStyles.Add
myBodyTextStyle.Name = "body text"
myBodyTextStyle.PointSize = 12
myBodyTextStyle.FirstLineIndent = 24
Set myCharacterStyle = myDocument.CharacterStyles.Add
myCharacterStyle.Name = "Emphasis"
myCharacterStyle.FontStyle = "Italic"
```

```
Set myTextFrameStyle = myDocument.ObjectStyles.Add
myTextFrameStyle.Name = "Text Frame Style"
myTextFrameStyle.CornerEffect = idCornerEffects.idRoundedCorner
myTextFrameStyle.StrokeColor = myDocument.Colors.Item("Black")
myTextFrameStyle.StrokeWeight = 2
Rem Add XML elements.
Set myRootXMLElement = myDocument.XMLElements.Item(1)
Set myXMLElementA = myRootXMLElement.XMLElements.Add(myHeading1XMLTag)
myXMLElementA.Contents = "Heading 1"
myXMLElementA.ApplyParagraphStyle myHeading1Style, True
myXMLElementA.InsertTextAsContent vbCr, idLocationOptions.idAfter
Set myXMLElementB = myRootXMLElement.XMLElements.Add(myPara1XMLTag)
myXMLElementB.Contents = "This is the first paragraph in the article."
myXMLElementB.ApplyParagraphStyle myPara1Style, True
myXMLElementB.InsertTextAsContent vbCr, idLocationOptions.idAfter
Set myXMLElementC = myRootXMLElement.XMLElements.Add(myBodyTextXMLTag)
myXMLElementC.Contents = "This is the second paragraph in the article."
myXMLElementC.ApplyParagraphStyle myBodyTextStyle, True
myXMLElementC.InsertTextAsContent vbCr, idLocationOptions.idAfter
Set myXMLElementD = myRootXMLElement.XMLElements.Add(myHeading2XMLTag)
myXMLElementD.Contents = "Heading 2"
myXMLElementD.ApplyParagraphStyle myHeading2Style, True
myXMLElementD.InsertTextAsContent vbCr, idLocationOptions.idAfter
Set myXMLElementE = myRootXMLElement.XMLElements.Add(myPara1XMLTag)
myXMLElementE.Contents = "This is the first paragraph following the subhead."
myXMLElementE.ApplyParagraphStyle myPara1Style, True
myXMLElementE.InsertTextAsContent vbCr, idLocationOptions.idAfter
Set myXMLElementF = myRootXMLElement.XMLElements.Add(myBodyTextXMLTag)
myXMLElementF.Contents = "This is the second paragraph following the subhead."
myXMLElementF.ApplyParagraphStyle myBodyTextStyle, True
myXMLElementF.InsertTextAsContent vbCr, idLocationOptions.idAfter
Set myXMLElementG = myXMLElemetF.XMLElements.Add(myBodyTextXMLTag)
myXMLElementG.Contents = "Note:"
Set myXMLElementG = myXMLElementG.Move(idLocationOptions.idAtBeginning, myXMLElementF)
myXMLElementG.InsertTextAsContent " ", idLocationOptions.idAfter
myXMLElementG.ApplyCharacterStyle myCharacterStyle, True
Set myTextFrame = myDocument.TextFrames.Add
myTextFrame.GeometricBounds = myGetBounds(myDocument, myDocument.Pages.Item(1))
Rem Associate the root XML element with the text frame.
myRootXMLElement.PlaceXML myDocument.Pages.Item(1).TextFrames.Item(1)
myRootXMLElement.ApplyObjectStyle myTextFrameStyle, True
```

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.

```
Rem Create a series of XML tags.
Set myRowTag = myDocument.XMLTags.Add("row")
Set myCellTag = myDocument.XMLTags.Add("cell")
Set myTableTag = myDocument.XMLTags.Add("table")
Rem Add XML elements.
Set myRootXMLElement = myDocument.XMLElements.Item(1)
With myRootXMLElement
    Set myTableXMLElement = .XMLElements.Add(myTableTag)
    With myTableXMLElement
        For myRowCounter = 1 To 6
            With .XMLElements.Add(myRowTag)
                .Contents = "Row " + CStr(myRowCounter)
                For myCellCounter = 1 To 4
                    With .XMLElements.Add(myCellTag)
                        .Contents = "Cell " + CStr(myCellCounter)
                    End With
                Next
            End With
        Next
    End With
End With
Set myTable = myTableXMLElement.ConvertElementToTable(myRowTag, myCellTag)
Set myTextFrame = myDocument.TextFrames.Add
myTextFrame.GeometricBounds = myGetBounds(myDocument, myDocument.Pages.Item(1))
myDocument.XMLElements.Item(1).XMLElements.Item(1).PlaceXML myTextFrame
```

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):

```
Rem Create a series of XML tags.
Set myRowTag = myDocument.XMLTags.Add("row")
Set myCellTag = myDocument.XMLTags.Add("cell")
Set myTableTag = myDocument.XMLTags.Add("table")
Rem Create a table style and a cell style.
Set myTableStyle = myDocument.TableStyles.Add
myTableStyle.StartColumnFillColor = myDocument.Colors.Item("Black")
myTableStyle.StartColumnFillTint = 25
Set myCellStyle = myDocument.CellStyles.Add
myCellStyle.FillColor = myDocument.Colors.Item("Black")
myCellStyle.FillTint = 45
Rem Add XML elements.
Set myRootXMLElement = myDocument.XMLElements.Item(1)
With myRootXMLElement
    Set myTableXMLElement = .XMLElements.Add(myTableTag)
    With myTableXMLElement
        For myRowCounter = 1 To 6
            With .XMLElements.Add(myRowTag)
                .Contents = "Row " + CStr(myRowCounter)
                For myCellCounter = 1 To 4
                    With .XMLElements.Add(myCellTag)
                        .Contents = "Cell " + CStr(myCellCounter)
                    End With
                Next
            End With
        Next
    End With
End With
```

```
Set myTable = myTableXMLElement.ConvertElementToTable(myRowTag, myCellTag)
Set myTableXMLElement = myDocument.XMLElements.Item(1).XMLElements.Item(1)
myTableXMLElement.ApplyTableStyle myTableStyle
myTableXMLElement.XMLElements.Item(1).ApplyCellStyle myCellStyle
myTableXMLElement.XMLElements.Item(6).ApplyCellStyle myCellStyle
myTableXMLElement.XMLElements.Item(11).ApplyCellStyle myCellStyle
myTableXMLElement.XMLElements.Item(16).ApplyCellStyle myCellStyle
myTableXMLElement.XMLElements.Item(17).ApplyCellStyle myCellStyle
myTableXMLElement.XMLElements.Item(22).ApplyCellStyle myCellStyle
myDocument.XMLElements.Item(1).PlaceIntoFrame myDocument.Pages.Item(1),
myGetBounds(myDocument, myDocument.Pages.Item(1))
myTable.AlternatingFills = idAlternatingFillsTypes.idAlternatingColumns
```

10 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 7, "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 CS4 Scripting Tutorial* and know how to create and run a script. We also assume you have some knowledge of XML and have read Chapter 9, "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:

```
Class RuleName
    Public Property Get name
        name = "RuleNameAsString"
    End Property
    Public Property Get xpath
        xpath = "ValidXPathSpecifier"
    End Property
    Public Function  apply (myXMLElement, myRulesProcessor)
        Rem Do something here.
        Rem Return true to stop further processing of the XML element.
        apply = False
    End Function
End Class
```

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 150."

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:

```
myRuleSet = 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.vbs` 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.

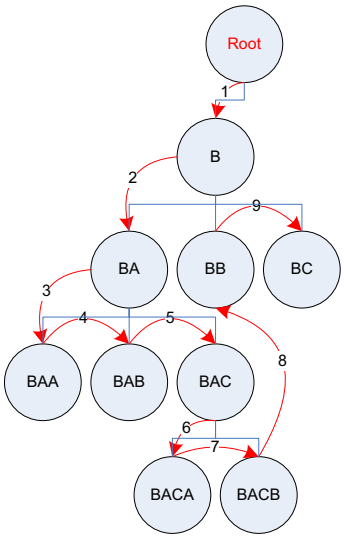
The `__processRuleSet` function applies rules to XML elements in "depth first" order; that is, XML elements and their child elements are processed in the order in which they appear in the XML structure. For each "branch" of the XML structure, the XML-rules processor visits each XML element before moving 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.

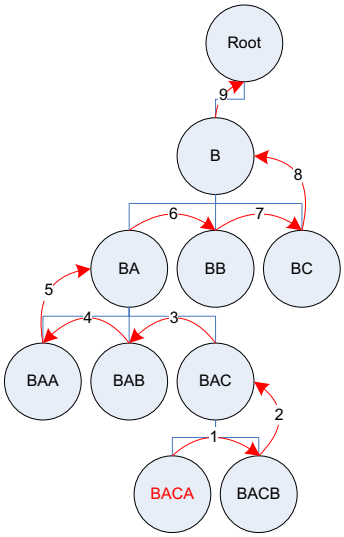
To see how all these functions work together, import the `DepthFirstProcessingOrder.xml` file into a new document, then run the `DepthFirstProcessingOrder.jsx` script. InDesign creates a text frame, that lists the attribute names of each element in the sample XML file in the order in which they were 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:

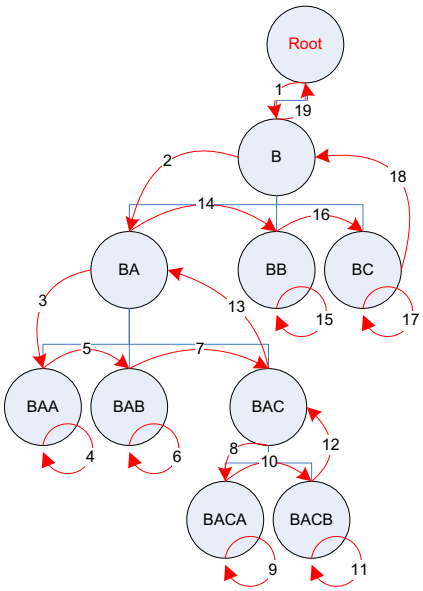


Iteration with `__processChildren` (assuming a rule that matches every XML element in the structure) is shown in the following figure:



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.)

```
Class NormalRule
    Public Property Get name
        name = "NormalRule"
    End Property
    Public Property Get xpath
        xpath = "//XMLElement"
    End Property
    Public Function apply(myXMLElement, myRuleProcessor)
        With myXMLElement
            myStory.InsertionPoints.Item(-1).Contents =
            .XMLAttributes.Item(1).Value & vbCr
        End With
        apply = false
    End Function
End Class
Class ProcessChildrenRule
    Public Property Get name
        name = "ProcessChildrenRule"
    End Property
    Public Property Get xpath
        xpath = "//XMLElement"
    End Property
    Public Function apply(myXMLElement, myRuleProcessor)
        glueCode_processChildren(myRuleProcessor)
        With myXMLElement
            myXMLElement.XMLattributes.Item(1).Value
            myStory.InsertionPoints.Item(-1).Contents =
            .XMLAttributes.Item(1).Value & vbCr
        End With
        apply = false
    End Function
End Class
```



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

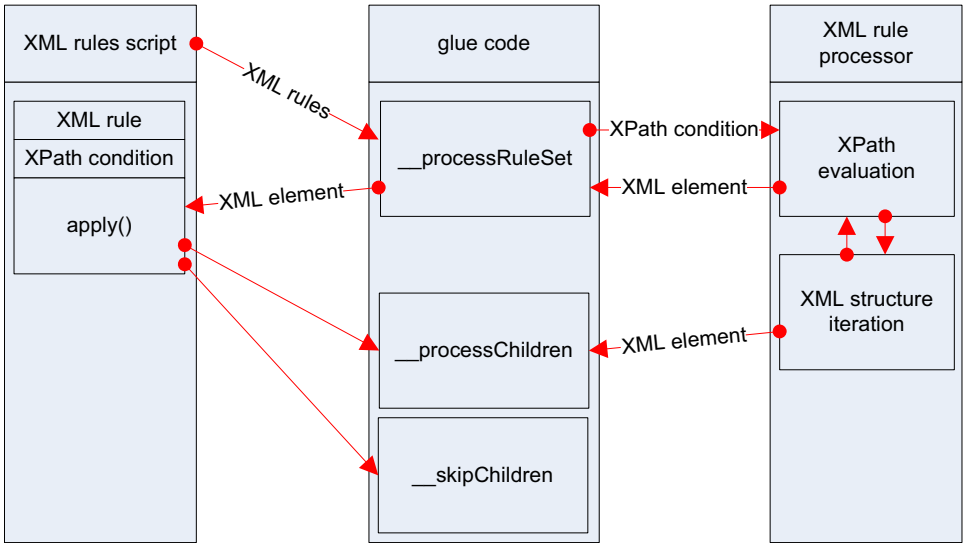
Because XML rules are part of the InDesign scripting model, scripts that use rules do not differ in nature from ordinary scripts, and they benefit from the same error-handling mechanism. When InDesign generates an error, an XML-rules script behaves no differently than any other script. InDesign errors 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 150). 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.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Item(1)
myFilePath = myInDesign.FilePath
myFilePath = myFilePath & "\Scripts\Xml rules\glue code.vbs"
Rem Use the Include function to load the glue code file.
Include myFilePath
Set myAddReturns = new AddReturns
myRuleSet = Array(myAddReturns)
Rem The third parameter of __processRuleSet is a
Rem prefix mapping table; we'll leave it empty.
glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet, Array()
Rem XML rule "AddReturns"
Class AddReturns
    Public Property Get name
        name = "AddReturns"
    End Property
    Public Property Get xpath
        xpath = "//*"
    End Property
```

```
    Public Function  apply (myXMLElement, myRulesProcessor)
        With myXMLElement
            Rem Add a return character after the end of the XML element
            Rem (this means that the return does not become part of the
            Rem XML element data, but becomes text data associated with the
            Rem parent XML element).
            .InsertTextAsContent vbcr, idXMLElementPosition.idAfterElement
        End With
        Rem Enter true to stop further processing of this element.
        apply = False
    End Function
End Class
Function Include(myScriptFilePath)
    Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
    Set myScriptFile = myFileSystemObject.OpenTextFile(myScriptFilePath)
    myScriptContents = myScriptFile.ReadAll
    ExecuteGlobal myScriptContents
End 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.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
Set myDocument = myInDesign.Documents.Item(1)
myFilePath = myInDesign.FilePath
myFilePath = myFilePath & "\Scripts\Xml rules\glue code.vbs"
Rem Use the Include function to load the glue code file.
Include myFilePath
myRuleSet = Array(new ProcessDevice,new ProcessName,new ProcessType,_
new ProcessPartNumber,new ProcessSupplyVoltage,new ProcessPackageType,_
new ProcessPackageOne,new ProcessPackages,new ProcessPrice)
Rem The third parameter of __processRuleSet is a
Rem prefix mapping table; we'll leave it empty.
glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet, Array()
Class ProcessDevice
    Public Property Get name
        name = "ProcessDevice"
    End Property
    Public Property Get xpath
        xpath = "/devices/device"
    End Property
    Public Function  apply (myXMLElement, myRulesProcessor)
        With myXMLElement
            .InsertTextAsContent vbcr, idXMLElementPosition.idAfterElement
        End With
        apply = False
    End Function
End Class
Class ProcessName
    Public Property Get name
        name = "ProcessName"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/name"
    End Property
```

```
    Public Function  apply (myXMLElement, myRulesProcessor)
        With myXMLElement
            Rem Add static text at the beginning of the element.
            .InsertTextAsContent "Device Name:", idXMLElementPosition.idBeforeElement
            Rem Add a return character at the end of the element.
            .InsertTextAsContent vbcr, idXMLElementPosition.idAfterElement
        End With
        apply = False
    End Function
End Class
Class ProcessType
    Public Property Get name
        name = "ProcessType"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/type"
    End Property
    Public Function  apply (myXMLElement, myRulesProcessor)
        With myXMLElement
            Rem Add static text at the beginning of the element.
            .InsertTextAsContent "Circuit Type:", idXMLElementPosition.idBeforeElement
            Rem Add a return character at the end of the element.
            .InsertTextAsContent vbcr, idXMLElementPosition.idAfterElement
        End With
        apply = False
    End Function
End Class
Class ProcessPartNumber
    Public Property Get name
        name = "ProcessPartNumber"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/part_number"
    End Property
    Public Function  apply (myXMLElement, myRulesProcessor)
        With myXMLElement
            Rem Add static text at the beginning of the element.
            .InsertTextAsContent "Part Number:", idXMLElementPosition.idBeforeElement
            Rem Add a return character at the end of the element.
            .InsertTextAsContent vbcr, idXMLElementPosition.idAfterElement
        End With
        apply = False
    End Function
End Class
Rem Adds static text around the "minimum" and "maximum"
Rem XML elements of the "supply_voltage" XML element.

Class ProcessSupplyVoltage
    Public Property Get name
        name = "ProcessSupplyVoltage"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/supply_voltage"
    End Property
```

```
    Public Function  apply (myXMLElement, myRulesProcessor)
        With myXMLElement
            .InsertTextAsContent "Supply Voltage: From ",
            idXMLElementPosition.idBeforeElement
            With myXMLElement.XMLElements.Item(1)
                .InsertTextAsContent " to ", idXMLElementPosition.idAfterElement
            End with
            With myXMLElement.XMLElements.Item(-1)
                Rem Add static text to the end of the voltage range.
                .InsertTextAsContent " volts", idXMLElementPosition.idAfterElement
            End with
            Rem Add a return at the end of the XML element.
            .InsertTextAsContent vbcr, idXMLElementPosition.idAfterElement
        End With
        apply = True
    End Function
End Class
Rem Insert a dash between the "type" and "pins" elements.
Class ProcessPackageType
    Public Property Get name
        name = "ProcessPackageType"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/package/type"
    End Property
    Public Function  apply (myXMLElement, myRulesProcessor)
        With myXMLElement
            .InsertTextAsContent "-", idXMLElementPosition.idAfterElement
        End With
        apply = true
    End Function
End Class
Rem Process the first "package" element.
Class ProcessPackageOne
    Public Property Get name
        name = "ProcessPackageOne"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/package[1]"
    End Property
    Public Function  apply (myXMLElement, myRulesProcessor)
        With myXMLElement
            .InsertTextAsContent "Package: ", idXMLElementPosition.idBeforeElement
        End With
        apply = true
    End Function
End Class
Rem Process the remaining "package" elements.
Class ProcessPackages
    Public Property Get name
        name = "ProcessPackages"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/package"
    End Property
```

```
    Public Function  apply (myXMLElement, myRulesProcessor)
        With myXMLElement
            .InsertTextAsContent", ", idXMLElementPosition.idBeforeElement
        End With
        apply = True
    End Function
End Class
Class ProcessPrice
    Public Property Get name
        name = "ProcessPrice"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/price"
    End Property
    Public Function  apply (myXMLElement, myRulesProcessor)
        With myXMLElement
            Rem Add a return at the start of the XML element.
            .InsertTextAsContent vbCr & "Price: $",
            idXMLElementPosition.idBeforeElement
            Rem .InsertTextAsContent "Price: $", idXMLElementPosition.idBeforeElement
            .InsertTextAsContent vbcr, idXMLElementPosition.idAfterElement
        End With
        apply = False
    End Function
End Class
Function Include(myScriptFilePath)
    Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
    Set myScriptFile = myFileSystemObject.OpenTextFile(myScriptFilePath)
    myScriptContents = myScriptFile.ReadAll
    ExecuteGlobal myScriptContents
End 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):

```
    myFilePath = myInDesign.FilePath
    myFilePath = myFilePath & "\Scripts\Xml rules\glue code.vbs"
    Rem Use the Include function to load the glue code file.
    Include myFilePath
    myRuleSet = Array(new ListItems)
    glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet,
    Array()
    Rem Match all following sibling XML elements
    Rem of the first "item" XML element.
    Class ListItems
        Public Property Get name
            name = "ListItems"
        End Property
        Public Property Get xpath
            xpath = "/xmlElement/item[1]/following-sibling::*"
        End Property
        Public Function  apply (myXMLElement, myRulesProcessor)
            With myXMLElement
                .InsertTextAsContent vbcr, idXMLElementPosition.idBeforeElement
            End With
            apply = False
        End Function
    End Class
```

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.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.Documents.Count > 0 Then
    Set myDocument = myInDesign.Documents.Item(1)
    myFilePath = myInDesign.FilePath
    myFilePath = myFilePath & "\Scripts\Xml rules\glue code.vbs"
    Rem Use the Include function to load the glue code file.
    Include myFilePath
    myRuleSet = Array(new MoveElement)
    glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet,
Array()
End If
Class MoveElement
    Public Property Get name
        name = "MoveElement"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/part_number"
    End Property
```

```
    Rem Moves the part_number XML element to the start of
    Rem the device XML element (the parent).
    Public Function  apply (myXMLElement, myRuleProcessor)
        Rem Because this rule makes changes to the XML structure,
        Rem you must use _skipChildren to avoid invalidating
        Rem the XML element references.
        glueCode_skipChildren(myRuleProcessor)
        With myXMLElement
            Set myParent = .Parent
            Set myNameElement = myParent.XMLElements.Item(1)
            .Move idLocationOptions.idBefore, myNameElement
        End With
        apply = false
    End Function
End Class
```

Duplicating XML elements with XML rules

As discussed in Chapter 9, "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.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.Documents.Count > 0 Then
    Set myDocument = myInDesign.Documents.Item(1)
    myFilePath = myInDesign.FilePath
    myFilePath = myFilePath & "\Scripts\Xml rules\glue code.vbs"
    Rem Use the Include function to load the glue code file.
    Include myFilePath
    myRuleSet = Array(new DuplicateElement)
    glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet,
Array()
End If
Class DuplicateElement
    Public Property Get name
        name = "DuplicateElement"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/part_number"
    End Property
    Rem Moves the part_number XML element to the start of
    Rem the device XML element (the parent).
    Public Function  apply (myXMLElement, myRuleProcessor)
        Rem Because this rule makes changes to the XML structure,
        Rem you must use _skipChildren to avoid invalidating
        Rem the XML element references.
        glueCode_skipChildren(myRuleProcessor)
        With myXMLElement
            .Duplicate
        End With
        apply = false
    End Function
End Class
```

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.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.Documents.Count > 0 Then
    Set myDocument = myInDesign.Documents.Item(1)
    myFilePath = myInDesign.FilePath
    myFilePath = myFilePath & "\Scripts\Xml rules\glue code.vbs"
    Rem Use the Include function to load the glue code file.
    Include myFilePath
    myRuleSet = Array(new AddAttribute)
    glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet,
Array()
End If
Class AddAttribute
    Public Property Get name
        name = "AddAttribute"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/part_number"
    End Property
    Rem Adds the content of the XML element to an attribute
    Rem of the parent of the XML element. This can make finding
    Rem the element by its content much easier and faster.
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            Set myParent = .Parent
            myString = myXMLElement.Texts.Item(1).Contents
            Set myXMLAttribute = .Parent.XMLAttributes.Add("part_number", myString)
        End With
        apply = false
    End Function
End Class
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.Documents.Count > 0 Then
    Set myDocument = myInDesign.Documents.Item(1)
    myFilePath = myInDesign.FilePath
    myFilePath = myFilePath & "\Scripts\Xml rules\glue code.vbs"
    Rem Use the Include function to load the glue code file.
    Include myFilePath
    myRuleSet = Array(new ConvertToAttribute)
    glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet,
Array()
End If
Class ConvertToAttribute
    Public Property Get name
        name = "ConvertToAttribute"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/part_number"
    End Property
    Rem Converts an XML element to an attribute
    Rem of the parent of the XML element.
    Public Function  apply (myXMLElement, myRuleProcessor)
        glueCode_skipChildren(myRuleProcessor)
        With myXMLElement
            .ConvertToAttribute
        End With
        apply = true
    End Function
End Class
```

To move data from an XML attribute to an XML element, use the `convertToElement` method, as described in Chapter 9, "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.

```
myCounter = 0
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.Documents.Count > 0 Then
    Set myDocument = myInDesign.Documents.Item(1)
    Rem Define two colors.
    Set myColorA = myAddColor(myDocument, "ColorA",
    idColorModel.idProcess, Array(0, 100, 80, 0))
    Set myColorB = myAddColor(myDocument, "ColorB",
    idColorModel.idProcess, Array(100, 0, 80, 0))
    myFilePath = myInDesign.FilePath
    myFilePath = myFilePath & "\Scripts\Xml rules\glue code.vbs"
    Rem Use the Include function to load the glue code file.
    Include myFilePath
    myRuleSet = Array(new ReturnFalse, new ReturnTrue)
    glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet, Array()
End If
```

```
Rem Adds a color to the text of every element in the structure.
Class ReturnFalse
    Public Property Get name
        name = "ReturnFalse"
    End Property
    Public Property Get xpath
        xpath = "//*"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            .Texts.Item(1).FillColor = myColorA
        End With
        Rem Leaves the XML element available to further processing.
        apply = false
    End Function
End Class
Rem Adds a color to the text of every other element in the structure.
Class ReturnTrue
    Public Property Get name
        name = "ReturnTrue"
    End Property
    Public Property Get xpath
        xpath = "//*"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            Rem Test based on the global variable "myCounter"
            If myCounter Mod 2 = 0 Then
                .Texts.Item(1).FillColor = myColorB
            End If
            myCounter = myCounter + 1
        End With
        Rem Do not process the element with any further matching rules.
        apply = true
    End Function
End Class
Function Include(myScriptFilePath)
    Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
    Set myScriptFile = myFileSystemObject.OpenTextFile(myScriptFilePath)
    myScriptContents = myScriptFile.ReadAll
    ExecuteGlobal myScriptContents
End Function
Function myAddColor(myDocument, myColorName, myColorModel, myColorValue)
    On Error Resume Next
    Set myColor = myDocument.colors.Item(myColorName)
    If Err.Number <> 0 Then
        Set myColor = myDocument.colors.Add
        myColor.Name = myColorName
        Err.Clear
    End If
    On Error GoTo 0
    myColor.model = myColorModel
    myColor.colorValue = myColorValue
    Set myAddColor = myColor
End Function
```

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.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.Documents.Count > 0 Then
    Set myDocument = myInDesign.Documents.Item(1)
    Rem Define two colors.
    Set myColorA = myAddColor(myDocument, "ColorA", idColorModel.idProcess,
    Array(0, 100, 80, 0))
    myFilePath = myInDesign.FilePath
    myFilePath = myFilePath & "\Scripts\Xml rules\glue code.vbs"
    Rem Use the Include function to load the glue code file.
    Include myFilePath
    myRuleSet = Array(new AddAttribute)
    glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet,
Array()
    Rem Now that the attributes have been added, find and format
    Rem the XML element whose attribute content matches a specific string.
    myRuleSet = Array(new FindAttribute)
    glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet,
Array()
End If
Class AddAttribute
    Public Property Get name
        name = "AddAttribute"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/part_number"
    End Property
    Rem Adds the content of the XML element to an attribute
    Rem of the parent of the XML element. This can make finding
    Rem the element by its content much easier and faster.
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            Set myParent = .Parent
            myString = myXMLElement.Texts.Item(1).Contents
            Set myXMLAttribute = .Parent.XMLAttributes.Add("part_number", myString)
        End With
        apply = false
    End Function
End Class
Class FindAttribute
    Public Property Get name
        name = "FindAttribute"
    End Property
    Public Property Get xpath
        xpath = "/devices/device[@part_number = 'DS001']"
    End Property
```

```
    Rem Applies a color to the text of an XML element
    Rem (to show that we found it).
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            .Texts.Item(1).FillColor = myColorA
        End With
        apply = false
    End Function
End Class
```

The following script shows how to use the `findText` method to find and format XML content (for the complete script, see FindXMLElementByFindText):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.Documents.Count > 0 Then
    Set myDocument = myInDesign.Documents.Item(1)
    Rem Define two colors.
    Set myColorA = myAddColor(myDocument, "ColorA", idColorModel.idProcess,
    Array(0, 100, 80, 0))
    myFilePath = myInDesign.FilePath
    myFilePath = myFilePath & "\Scripts\Xml rules\glue code.vbs"
    Rem Use the Include function to load the glue code file.
    Include myFilePath
    myRuleSet = Array(new FindByFindText)
    glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet,
Array()
End If
Class FindByFindText
    Public Property Get name
        name = "FindByFindText"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/description"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            myResetFindText
            If .Texts.Item(1).contents <> "" Then
                Rem Set the find options.
                myInDesign.FindChangeTextOptions.CaseSensitive = False
                myInDesign.FindChangeTextOptions.IncludeFootnotes = False
                myInDesign.FindChangeTextOptions.IncludeHiddenLayers = False
                myInDesign.FindChangeTextOptions.IncludeLockedLayersForFind =
                False
                myInDesign.FindChangeTextOptions.IncludeLockedStoriesForFind =
                False
                myInDesign.FindChangeTextOptions.IncludeMasterPages = False
                myInDesign.FindChangeTextOptions.WholeWord = False
                Rem Search for the word "triangle" in the content of the element.
                myInDesign.FindTextPreferences.FindWhat = "triangle"
                Set myFoundItems = .FindText
                If myFoundItems.Count > 0 Then
                    .Texts.Item(1).FillColor = myColorA
                End If
                myResetFindText
            End If
        End With
        apply = false
    End Function
End Class
```

```
Function myResetFindText
    myInDesign.FindTextPreferences = idNothingEnum.idNothing
    myInDesign.ChangeTextPreferences = idNothingEnum.idNothing
End Function
```

The following script shows how to use the `findGrep` method to find and format XML content (for the complete script, see FindXMLElementByFindGrep):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.Documents.Count > 0 Then
    Set myDocument = myInDesign.Documents.Item(1)
    Rem Define two colors.
    Set myColorA = myAddColor(myDocument, "ColorA", idColorModel.idProcess,
    Array(0, 100, 80, 0))
    myFilePath = myInDesign.FilePath
    myFilePath = myFilePath & "\Scripts\Xml rules\glue code.vbs"
    Rem Use the Include function to load the glue code file.
    Include myFilePath
    myRuleSet = Array(new FindByFindGrep)
    glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet,
Array()
End If
Class FindByFindGrep
    Public Property Get name
        name = "FindByFindGrep"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/description"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            myResetFindGrep
            If .Texts.Item(1).contents <> "" Then
                Rem Search for the regular expression:
                Rem "(?i)pulse.*?triangle|triangle.*?pulse"
                Rem in the content of the element.
                myInDesign.FindGrepPreferences.FindWhat =
                "(?i)pulse.*?triangle|triangle.*?pulse"
                Set myFoundItems = .FindGrep
                If myFoundItems.Count > 0 Then
                    .Texts.Item(1).FillColor = myColorA
                End If
                myResetFindGrep
            End If
        End With
        apply = false
    End Function
End Class
Function myResetFindGrep
    myInDesign.FindTextPreferences = idNothingEnum.idNothing
    myInDesign.ChangeTextPreferences = idNothingEnum.idNothing
End Function
```

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.

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.Documents.Count > 0 Then
    Set myDocument = myInDesign.Documents.Item(1)
    myFilePath = myInDesign.FilePath
    myFilePath = myFilePath & "\Scripts\Xml rules\glue code.vbs"
    Rem Use the Include function to load the glue code file.
    Include myFilePath
    Set myXMLTag = myMakeXMLTag(myDocument, "VCOs")
    Set myContainerElement =
    myDocument.XMLElements.Item(1).XMLElements.Add(myXMLTag)
    myRuleSet = Array(new ExtractVCO)
    glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet,
Array()
End If
Class ExtractVCO
    Public Property Get name
        name = "ExtractVCO"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/type"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            If .Texts.Item(1).Contents = "VCO" Then
                Set myNewElement = .Parent.Duplicate
                myNewElement.Move idLocationOptions.idAtEnd,
                myDocument.XMLElements.Item(1).XMLElements.Item(-1)
            End If
        End With
        apply = false
    End Function
End Class
```

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):

```
Set myInDesign = CreateObject("InDesign.Application.CS4")
If myInDesign.Documents.Count > 0 Then
    myFilePath = myInDesign.FilePath
    myFilePath = myFilePath & "\Scripts\Xml rules\glue code.vbs"
    Rem Use the Include function to load the glue code file.
    Include myFilePath
    Set myDocument = myInDesign.Documents.Item(1)
    Rem Document setup
    With myDocument.ViewPreferences
        .HorizontalMeasurementUnits = idMeasurementUnits.idPoints
        .VerticalMeasurementUnits = idMeasurementUnits.idPoints
    End With
    Rem Create a color.
    Set myColor = myAddColor(myDocument, "Red", idColorModel.idProcess,
    Array(0, 100, 100, 0))
    Rem Create a series of paragraph styles.
    Set myParagraphStyle = myMakeParagraphStyle(myDocument, "DeviceName")
    myParagraphStyle.PointSize =24
    myParagraphStyle.Leading = 24
    myParagraphStyle.FillColor = myColor
    myParagraphStyle.SpaceBefore = 24
    Set myParagraphStyle = myMakeParagraphStyle(myDocument, "DeviceType")
    myParagraphStyle.PointSize =12
    myParagraphStyle.Leading = 12
    myParagraphStyle.FontStyle = "Bold"
    Set myParagraphStyle = myMakeParagraphStyle(myDocument, "PartNumber")
    myParagraphStyle.PointSize =12
    myParagraphStyle.Leading = 12
    myParagraphStyle.FontStyle = "Bold"
    Set myParagraphStyle = myMakeParagraphStyle(myDocument, "Voltage")
    myParagraphStyle.PointSize =10
    myParagraphStyle.Leading = 12
    myParagraphStyle.FontStyle = "Bold"
    Set myParagraphStyle = myMakeParagraphStyle(myDocument, "DevicePackage")
    myParagraphStyle.PointSize =10
    myParagraphStyle.Leading = 12
    Set myParagraphStyle = myMakeParagraphStyle(myDocument, "Price")
    myParagraphStyle.PointSize =10
    myParagraphStyle.Leading = 12
    myParagraphStyle.FontStyle = "Bold"
    myRuleSet = Array(new ProcessDevice, new ProcessName, new ProcessType, new
ProcessPartNumber, new ProcessSupplyVoltage, new ProcessPrice, new ProcessPackageType,
new ProcessPackageOne, new ProcessPackages)
    glueCode_ProcessRuleSet myInDesign, myDocument.XMLElements.Item(1), myRuleSet,
Array()
End If
Class ProcessDevice
    Public Property Get name
        name = "ProcessDevice"
    End Property
    Public Property Get xpath
        xpath = "/devices/device"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            .InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
        End With
        apply = false
    End Function
End Class
```

```
Class ProcessName
    Public Property Get name
        name = "ProcessName"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/name"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            .InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
            .ApplyParagraphStyle myDocument.ParagraphStyles.Item("DeviceName")
        End With
        apply = true
    End Function
End Class
Class ProcessType
    Public Property Get name
        name = "ProcessType"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/type"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            .InsertTextAsContent "Circuit Type: ",
            idXMLElementPosition.idBeforeElement
            .InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
            .ApplyParagraphStyle myDocument.ParagraphStyles.Item("DeviceType")
        End With
        apply = true
    End Function
End Class
Class ProcessPartNumber
    Public Property Get name
        name = "ProcessPartNumber"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/part_number"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            .InsertTextAsContent "Part Number: ",
            idXMLElementPosition.idBeforeElement
            .InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
            .ApplyParagraphStyle myDocument.ParagraphStyles.Item("PartNumber")
        End With
        apply = true
    End Function
End Class
Class ProcessSupplyVoltage
    Public Property Get name
        name = "ProcessPartNumber"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/supply_voltage"
    End Property
```

```
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            .InsertTextAsContent "Supply Voltage From: ",
            idXMLElementPosition.idBeforeElement
            With .XMLElements.Item(1)
                .InsertTextAsContent " to ", idXMLElementPosition.idAfterElement
            End With
            With .XMLElements.Item(-1)
                .InsertTextAsContent " volts",
                idXMLElementPosition.idAfterElement
            End With
            .InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
            .ApplyParagraphStyle myDocument.ParagraphStyles.Item("Voltage")
        End With
        apply = true
    End Function
End Class
Class ProcessPackageType
    Public Property Get name
        name = "ProcessPackageType"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/package/type"
    End Property
    Public Function  apply (myXMLElement, myRulesProcessor)
        With myXMLElement
            .InsertTextAsContent "-", idXMLElementPosition.idAfterElement
        End With
        apply = true
    End Function
End Class
Rem Process the first "package" element.
Class ProcessPackageOne
    Public Property Get name
        name = "ProcessPackageOne"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/package[1]"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            .InsertTextAsContent "Package: ",
            idXMLElementPosition.idBeforeElement
            Rem Because we have already added a return to the
            Rem end of this element as part of the ProcessPrice
            Rem rule, we can savly apply a paragrpah style.
            .ApplyParagraphStyle
            myDocument.ParagraphStyles.Item("DevicePackage")
        End With
        apply = true
    End Function
End Class
Rem Process the remaining "package" elements.
Class ProcessPackages
    Public Property Get name
        name = "ProcessPackages"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/package"
    End Property
```

```
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            .InsertTextAsContent", ", idXMLElementPosition.idBeforeElement
        End With
        apply = True
    End Function
End Class
Class ProcessPrice
    Public Property Get name
        name = "ProcessPrice"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/price"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            Rem Add a return at the start of the XML element.
            .InsertTextAsContent vbCr & "Price: $",
            idXMLElementPosition.idBeforeElement
            .InsertTextAsContent vbcr, idXMLElementPosition.idAfterElement
            .ApplyParagraphStyle myDocument.ParagraphStyles.Item("Price")
        End With
        apply = False
    End Function
End Class
```

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:

```
Class ProcessDevice
    Public Property Get name
        name = "ProcessDevice"
    End Property
    Public Property Get xpath
        xpath = "/devices/device"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            .InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
            If myDocument.Pages.Item(1).TextFrames.Count > 0 Then
                Set myPage = myDocument.Pages.Add
                myBounds = myGetBounds(myDocument, myPage)
                Set myTextFrame = .PlaceIntoFrame(myPage, myBounds)
                myTextFrame.TextFramePreferences.FirstBaselineOffset =
                idFirstBaseline.idLeadingOffset
            Else
                Set myPage = myDocument.Pages.Item(1)
                myBounds = myGetBounds(myDocument, myPage)
                Set myTextFrame = .PlaceIntoFrame(myPage, myBounds)
                myTextFrame.TextFramePreferences.FirstBaselineOffset =
                idFirstBaseline.idLeadingOffset
            End If
        End With
        apply = false
    End Function
End Class
```

The "ProcessType" rule moves the "type" XML element to a new frame on the page:

```
Class ProcessType
    Public Property Get name
        name = "ProcessType"
    End Property
    Public Property Get xpath
        xpath = "/devices/device/type"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        With myXMLElement
            Set myPage = myDocument.Pages.Item(-1)
            myBounds = myGetBounds(myDocument, myPage)
            myX1 = myBounds(1)
            myY1 = myBounds(0)
            myBounds = Array(myY1-24, myX1, myY1, myX1 + 48)
            Set myTextFrame = .PlaceIntoFrame(myPage, myBounds)
            myTextFrame.TextFramePreferences.InsetSpacing = Array(6, 6, 6, 6)
            myTextFrame.FillColor = myDocument.Swatches.Item("Red")
            .ApplyParagraphStyle myDocument.ParagraphStyles.Item("DeviceType")
        End With
        apply = true
    End Function
End Class
```

Creating tables using XML rules

You can use the `ConvertElementToTable` method to turn an XML element into a table. This method has a limitation in that it assumes that all of the XML elements inside the table conform to a very specific set of XML tags—one tag for a row element; another for a cell, or column element. Typically, the XML data we

want to put into a table does not conform to this structure: it is likely that the XML elements we want to arrange in columns use heterogeneous XML tags (price, part number, etc.).

To get around this limitation, we can "wrap" each XML element we want to add to a table row using a container XML element, as shown in the following script fragments (see XMLRulesTable). In this example, a specific XML rule creates an XML element for each row.

```
Class ProcessDevice
    Public Property Get name
        name = "ProcessDevice"
    End Property
    Public Property Get xpath
        xpath = "//device[@type = 'VCO']"
    End Property
    Rem Create a new row for every device whose type is "VCO"
    Public Function  apply (myXMLElement, myRuleProcessor)
        Set myNewRowElement = myContainerElement.XMLElements.Add(myRowTag)
        apply = false
    End Function
End Class
```

Successive rules move and format their content into container elements inside the row XML element.

```
Class ProcessPrice
    Public Property Get name
        name = "ProcessPrice"
    End Property
    Public Property Get xpath
        xpath = "//device[@type = 'VCO']/price"
    End Property
    Public Function  apply (myXMLElement, myRuleProcessor)
        glueCode_skipChildren(myRuleProcessor)
        With myXMLElement
            Set myLastElement = myContainerElement.XMLElements.Item(-1)
            Set myNewElement = myLastElement.XMLElements.add(myCellTag)
            Set myPriceElement = .Move(idLocationOptions.idAtBeginning,
        myNewElement)
            myPriceElement.InsertTextAsContent "$",
        idXMLElementPosition.idBeforeElement
        End With
        apply = true
    End Function
End Class
```

Once all of the specified XML elements have been "wrapped," we can convert the container element to a table.

```
Set myTable = myContainerElement.ConvertElementToTable(myRowTag, myCellTag)
```

Scripting the XML-rules processor object

While we have provided a set of utility functions in `glue code.vbs`, 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.

```
myXPath = Array("/devices/device")
myXMLMatches = mySimulateXPath(myXPath)
Rem At this point, myXMLMatches contains all of the XML elements
Rem that matched the XPath expression provided in myXPath.
Rem In a real script, you could now process the elements.
Rem For this example, however, we'll simply display a message.
If IsEmpty(myXMLMatches(0)) = False Then
    MsgBox "Found " & CStr(UBound(myXMLMatches)+1) & " matching elements."
Else
    MsgBox "Did not find any matching XML elements."
End if
Function mySimulateXPath(myXPath)
    ReDim myMatchingElements(0)
    Set myInDesign = CreateObject("InDesign.Application.CS4")
    Set myRuleProcessor = myInDesign.XMLRuleProcessors.Add(myXPath)
    Set myDocument = myInDesign.Documents.Item(1)
    Set myRootXMLElement = myDocument.XMLElements.Item(1)
    Set myMatchData = myRuleProcessor.StartProcessingRuleSet(myRootXMLElement)
    Do While TypeName(myMatchData) <> "Nothing"
        Set myXMLElement = myMatchData.Element
        If IsEmpty(myMatchingElements(0)) = False Then
            ReDim Preserve myMatchingElements(UBound(myMatchingElements) + 1)
        End If
        Set myMatchingElements(UBound(myMatchingElements)) = myXMLElement
        Set myMatchData = myRuleProcessor.FindNextMatch
    Loop
    mySimulateXPath = myMatchingElements
End Function
```