ADOBE® INDESIGN® CS5

ADOBE INDESIGN CS5 SCRIPTING GUIDE: APPLESCRIPT

Adobe

Adobe® InDesign® CS5 Scripting Guide: AppleScript

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 CS5 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 CS5 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 handlers “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” handler.

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 `script preferences` object and its properties.
- Getting a reference to the executing script.
- Running scripts in prior versions of the scripting object model.
- Using the `do script` method to run scripts.
- Working with script labels.
- Running scripts at InDesign start-up.

We assume you already read *Adobe InDesign CS5 Scripting Tutorial* and know how to write, install, and run InDesign scripts in the scripting language of your choice.

Script Preferences

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

Property	Description
`enable redraw`	Turns screen redraw on or off while a script is running from the Scripts panel.
`scripts folder`	The path to the scripts folder.
`scripts list`	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
`user interaction level`	This property controls the alerts and dialogs InDesign presents to the user. When you set this property to `never interact`, InDesign does not display any alerts or dialogs. Set it to `interact with alerts` to enable alerts but disable dialogs. Set it to `interact with all` 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 `never interact` before opening the document, then restore user interaction (set the property to `interact with all`) before completing script execution.
`version`	The version of the scripting environment in use. For more information, see "Script Versioning" on page 12. 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 `active script` 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):

```
tell application "Adobe InDesign CS5"
    set myScript to active script
    display dialog ("The current script is: " & myScript)
    tell application "Finder"
        set myFile to file myScript
        set myParentFolder to container of myFile
    end tell
    display dialog ("The folder containing the active script is: " & myParentFolder)
end tell
```

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

Script Versioning

InDesign CS5 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 CS5 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 AppleScripts is done using the `tell` statement. You do not need a `tell` statement if you run the script from the Scripts Panel, because there is an implicit `tell` statement for the application launching the script.

```
--Target InDesign CS5
tell application "Adobe InDesign CS5"
```

Compilation

Typically, AppleScripts are compiled using the targeted application's dictionary. This behavior may be overridden by means of the `using terms from` statement, which substitutes another application's dictionary for compilation purposes:

```
tell application "Adobe InDesign CS5" --target CS5
    using terms from application "InDesign CS3" --compile using CS3
        --InDesign CS2 version script goes here.
    end using terms from
end tell
```

To generate a CS3 version of the AppleScript dictionary, use the `publish terminology` command, which is exposed on the application object. The dictionary 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, `/Users/`*`user-name`*`/Library/Preferences/Adobe InDesign/Version 4.0/Scripting Support/4.0-J` (where *`user-name`* is your user name.

```
tell application "Adobe InDesign CS5"
    --publish the InDesign CS3 dictionary (version 5.0 DOM)
    publish terminology version 5.0
end tell
```

Interpretation

The InDesign application object contains a `script preferences` 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 to 5.0 scripting object model
set version of script preferences to 5.0
```

Using the do script Method

The `do script` 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 `do script` 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 `do script` 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 `do script` 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 `do script` method to run scripts that were saved as strings in the `label` property of objects. Using this technique, an object can contain a script that controls its layout properties or updates its content according to certain parameters. Scripts also can be embedded in XML elements as an attribute of the element or as the contents of an element. See "Running Scripts at Startup" on page 18.

Sending parameters to do script

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

```
tell application "Adobe InDesign CS5"
    --Create a list of parameters.
    set myParameters to {"Hello from do script", "Your message here."}
    --Create a JavaScript as a string.
    set myJavaScript to "alert(\"First argument: \" + arguments[0] +
    \"\\rSecond argument: \" + arguments[1])"
    --Run the JavaScript using the do script command.
    do script myJavaScript language javascript with arguments myParameters
    --Create an AppleScript as a string.
    set myAppleScript to "tell application \"Adobe InDesign CS5\"" & return
    set myAppleScript to myAppleScript & "display dialog (\"First argument: \" &
    item 1 of arguments & return & \"Second argument: \" & item 2 of arguments)" & return
    set myAppleScript to myAppleScript & "end tell"
    --Run the AppleScript using the do script command.
    do script myAppleScript language applescript language with arguments myParameters
end tell
```

Returning values from do script

The following script fragment shows how to return a value from a script executed by `do script`. 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 to document 1
set myPage to page 1 of myDocument
set myTextFrame to text frame 1 of myPage
tell myDocument
    set myDestinationPage to make page
end tell
set myPageIndex to name of myDestinationPage
set myID to id of myTextFrame
set myJavaScript to "var myDestinationPage = arguments[1];" & return
set myJavaScript to myJavaScript & "myID = arguments[0];" & return
set myJavaScript to myJavaScript & "var myX = arguments[2];" & return
set myJavaScript to myJavaScript & "var myY = arguments[3];" & return
set myJavaScript to myJavaScript & "var myPageItem =
app.documents.item(0).pages.item(0).pageItems.itemByID(myID);" & return
set myJavaScript to myJavaScript &
"myPageItem.duplicate(app.documents.item(0).pages.item(myDestinationPage));" & return
--Create an array for the parameters we want to pass to the JavaScript.
set myArguments to {myID, myPageIndex, 0, 0}
set myDuplicate to do script myJavaScript language javascript with arguments
myArguments
--myDuplicate now contains a reference to the duplicated text frame.
--Change the text in the duplicated text frame.
set contents of myDuplicate to "Duplicated text frame."
```

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

```
tell application "Adobe InDesign CS5"
    --Create a string to be run as an AppleScript.
    set myAppleScript to "tell application \"Adobe InDesign CS5\"" & return
    set myAppleScript to myAppleScript & "tell script args" & return
    set myAppleScript to myAppleScript & "set value name \"ScriptArgumentA\"
    value \"This is the first AppleScript script argument value.\"" & return
    set myAppleScript to myAppleScript & "set value name \"ScriptArgumentB\"
    value \"This is the second AppleScript script argument value.\"" & return
    set myAppleScript to myAppleScript & "end tell" & return
    set myAppleScript to myAppleScript & "end tell"
    --Run the AppleScript string.
    do script myAppleScript language applescript language
    --Retrieve the script argument values set by the script.
    tell script args
        set myScriptArgumentA to get value name "ScriptArgumentA"
        set myScriptArgumentB to get value name "ScriptArgumentB"
    end tell
    --Display the script argument values in a dialog box.
    display dialog "ScriptArgumentA: " & myScriptArgumentA & return & "ScriptArgumentB:
" & myScriptArgumentB
    --Create a string to be run as a JavaScript.
```

```
    set myJavaScript to "app.scriptArgs.setValue(\"ScriptArgumentA\", \"This is the
first JavaScript script argument value.\");" & return
    set myJavaScript to myJavaScript & "app.scriptArgs.setValue(\"ScriptArgumentB\",
\"This is the second JavaScript script argument value.\");" & return
    --Run the JavaScript string.
    do script myJavaScript language javascript
    --Retrieve the script argument values set by the script.
    tell script args
        set myScriptArgumentA to get value name "ScriptArgumentA"
        set myScriptArgumentB to get value name "ScriptArgumentB"
    end tell
    --Display the script argument values in a dialog box.
    display dialog "ScriptArgumentA: " & myScriptArgumentA & return & "ScriptArgumentB:
" & myScriptArgumentB
end tell
```

Controlling Undo with do Script

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

```
--Given a script "myAppleScript" and an array of parameters "myParameters"...
tell application "Adobe InDesign CS5"
do script myJavaScript language javascript with arguments myArguments undo mode fast
entire script undo name "Script Action"
end tell
--undo modes can be:
--auto unto: Add no events to the Undo queue.
--entire script: Put a single event in the Undo queue.
--fast entire script: Put a single event in the Undo queue.
--script request: Undo each script action as a separate event.
--The last parameter is the text that appears in the Undo menu item.
```

Working with Script Labels

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

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

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

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

```
set myDocument to make document
set myPage to page 1 of myDocument
set myPageWidth to page width of document preferences of myDocument
set myPageHeight to page height of document preferences of myDocument
--Create 10 random page items.
repeat with i from 1 to 10
    set myX1 to my myGetRandom(0, myPageWidth, false)
    set myY1 to my myGetRandom(0, myPageHeight, false)
    set myX2 to my myGetRandom(0, myPageWidth, false)
    set myY2 to my myGetRandom(0, myPageHeight, false)
    tell myPage
        set myRectangle to make rectangle with properties {geometric bounds:{myY1, myX1,
myY2, myX2}}
    end tell
    if my myGetRandom(0, 1, true) = 1 then
        set label of myRectangle to "myScriptLabel"
    end if
end repeat
set myCount to 0
repeat with i from 1 to count of page items of myPage
    if label of page item i of myPage is "myScriptLabel" then
        set myCount to myCount + 1
    end if
end repeat
display dialog ("Found " & myCount & " page items with the label.")
--This function gets a random number in the range myStart to myEnd.
on myGetRandom(myStart, myEnd, myInteger)
    set myRange to myEnd - myStart
    if myInteger = true then
        set myRandom to myStart + (random number from myStart to myEnd)
    else
        set myRandom to myStart + (random number from myStart to myEnd) as integer
    end if
    return myRandom
end myGetRandom
```

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

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell view preferences of myDocument
        set horizontal measurement units to points
        set vertical measurement units to points
    end tell
    set myPage to page 1 of myDocument
    tell myPage
        set myRectangle to make rectangle with properties
        {geometric bounds:{72, 72, 144, 144}}
        --Insert a custom label using insert label. The first parameter is the
        --name of the label, the second is the text to add to the label.
        tell myRectangle
            insert label key "CustomLabel" value "This is some text stored
            in a custom label."
            --Extract the text from the label and display it in an alert.
            set myString to extract label key "CustomLabel"
        end tell
        display dialog ("Custom label contained: " & myString)
    end tell
end tell
```

Running Scripts at Startup

To run a script when InDesign starts, put the script in the Startup Scripts folder in the Scripts folder (for more information, see “Installing Scripts” in *Adobe InDesign CS5 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.
- Create watermarks.
- Apply different sizes to different pages (multiple pages sizes).
- Print a document.
- Export a document as Adobe PDF.
- Export pages of a document as EPS.

We assume that you have already read *Adobe InDesign CS5 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.)

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
end tell
```

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

```
tell application "Adobe InDesign CS5"
    --Replace "myDocumentPreset" in the following line with the name
    --of the document preset you want to use.
    set myDocument to make document with properties
    {document preset:"myDocumentPreset"}
end tell
```

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

```
--Creates a new document without showing the document window.
--The "showing window" parameter controls the visibility of the document.
--Hidden documents are not minimized, and will not appear until
--you tell the document to create a new window.
tell application "Adobe InDesign CS5"
    set myDocument to make document with properties {showing window:false}
    --To show the window:
    --tell myDocument
    --set myWindow to make window
    --end tell
end tell
```

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.)

```
tell application "Adobe InDesign CS5"
    --You'll have to fill in your own file path.
    set myDocument to open "yukino:myTestDocument.indd"
end tell
```

You can choose to prevent the document from displaying (that is, hide it) by setting the `showing window` parameter of the `open` command 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):

```
tell application "Adobe InDesign CS5"
    --You can use the "showing window" parameter to open files
    --without displaying them. This can speed up many scripting
    --operations, and makes it possible for a script to operate
    --on a file in the background. To display a document you've
    --opened this way, tell the document to create a new window.
    --You'll have to fill in your own file path.
    set myDocument to open "yukino:myTestDocument.indd" <lb>
    without showing window
    --At this point, your script could change or get information
    --from the hidden document. Once you've done that, you can show
    --the document window:
    tell myDocument to make window
end tell
```

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` command can do either operation, as shown in the following script fragment (from the SaveDocument tutorial script):

```
--Saves the active document.
--If the active document has been changed since it was last saved, save it.
tell application "Adobe InDesign CS5"
    if modified of active document is true then
        tell active document to save
    end if
end tell
```

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

```
--If the active document has not been saved (ever), save it.
tell application "Adobe InDesign CS5"
    if saved of active document is false then
        --If you do not provide a file name, InDesign displays the Save dialog box.
        tell active document to save saving in "yukino:myTestDocument.indd"
    end if
end tell
```

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

```
--Save the active document as a template.
tell application "Adobe InDesign CS5"
    set myDocument to active document
    tell myDocument
        if saved is true then
            --Convert the file name to a string.
            set myFileName to full name
            set myFileName to my myReplace(myFileName, ".indd", ".indt")
        else
            --If the document has not been saved, then give it a default file
            --name/file path. You'll have to fill in the file path.
            set myFileName to "yukino:myTestDocument.indt"
        end if
        save to myFileName with stationery
    end tell
end tell
on myReplace(myString, myFindString, myChangeString)
    set AppleScript's text item delimiters to myFindString
    set myTextList to every text item of (myString as text)
    set AppleScript's text item delimiters to myChangeString
    set myString to myTextList as string
    set AppleScript's text item delimiters to ""
    return myString
end myReplace
```

Closing a document

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

```
tell application "Adobe InDesign CS5"
    close document 1
    --document 1 always refers to the front-most document.
    --Note that you could also use:
    --close active document
end tell
```

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

```
tell application "Adobe InDesign CS5"
--Use "saving yes" to save the document,
--or "saving no" to close the document without saving,
--or "saving ask" to display a prompt. If you use
--"saving yes",you'll need to provide a reference
--to a file to save to in the second parameter (saving in).
    --If the file has never been saved (it's an untitled file),
    --display a prompt.
    if saved of active document is not equal to true then
        close active document saving ask
        --Or, to save to a specific file name
        --(you'll have to fill in the file path):
        --set myFile to "yukino:myTestDocument.indd"
        --close active document saving yes saving in myFile
    else
        --If the file has already been saved to a file, save it.
        close active document saving yes
    end if
end tell
```

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

```
tell application "Adobe InDesign CS5"
    tell documents to close without saving
end tell
```

Basic Page Layout

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

Defining page size and document length

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

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell document preferences of myDocument
        set page height to "800pt"
        set page width to "600pt"
        set page orientation to landscape
        set pages per document to 16
    end tell
end tell
```

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

Defining bleed and slug areas

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

```
tell application "Adobe InDesign CS5"
    --Create a new document.
    set myDocument to make document
    --The bleed and slug properties belong to the document preferences object.
    tell document preferences of myDocument
        --Bleed
        set document bleed bottom offset to "3p"
        set document bleed top offset to "3p"
        set document bleed inside or left offset to "3p"
        set document bleed outside or right offset to "3p"
        --Slug
        set slug bottom offset to "18p"
        set slug top offset to "3p"
        set slug inside or left offset to "3p"
        set slug right or outside offset to "3p"
    end tell
end tell
```

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

```
tell application "Adobe InDesign CS5"
    --Create a new document.
    set myDocument to make document
    --The bleed properties belong to the document preferences object.
    tell document preferences of myDocument
        --Bleed
        set document bleed top offset to "3p"
        set document bleed uniform size to true
    end tell
end tell
```

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

```
tell application "Adobe InDesign CS5"
    --Create a new document.
    set myDocument to make document
    --The bleed properties belong to the document preferences object.
    tell document preferences of myDocument
        --Slug
        set document slug uniform size to true
        set slug top offset to "3p"
    end tell
end tell
```

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 `document preferences` object; instead, it is in the `pasteboard preferences` object, as shown in the following script fragment (from the BleedSlugGuideColors tutorial script):

```
tell application "Adobe InDesign CS5"
    --Assumes you have a document open.
    tell pasteboard preferences of active document
        --Any of InDesign's guides can use the UIColors constants...
        set bleed guide color to cute teal
        set slug guide color to charcoal
        --...or you can specify a list of RGB values
        --(with values from 0 to 255)
        set bleed guide color to {0, 198, 192}
        set slug guide color to {192, 192, 192}
    end tell
end tell
```

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 `margin preferences` 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.)

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell view preferences of myDocument
        set horizontal measurement units to points
        set vertical measurement units to points
    end tell
    tell master spread 1 of myDocument
        tell margin preferences of pages
            set top to 36
            set left to 36
            set bottom to 48
            set right to 36
        end tell
    end tell
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell view preferences of myDocument
        set horizontal measurement units to points
        set vertical measurement units to points
    end tell
    tell margin preferences of page 1 of myDocument
        set top to 36
        set left to 36
        set bottom to 48
        set right to 36
    end tell
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell margin preferences of page 1 of myDocument
        set top to 0
        set left to 0
        set bottom to 0
        set right to 0
    end tell
    tell master spread 1 of myDocument
        tell margin preferences of pages
            set top to 0
            set left to 0
            set bottom to 0
            set right to 0
        end tell
    end tell
    --At this point, you can set your page size to a small width
    --and height (1x1 picas minimum).
    set page height of document preferences of myDocument to "1p"
    set page width of document preferences of myDocument to "6p"
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    tell margin preferences
        --Save the current application default margin preferences.
        set myY1 to top
        set myX1 to left
        set myY2 to bottom
        set myX2 to right
        --Set the application default margin preferences.
        set top to 0
        set left to 0
        set bottom to 0
        set right to 0
    end tell
    --At this point, you can create a new document.
    set myDocument to make document
    --At this point, you can set your page size to a small width and height
    --(1x1 picas minimum).
    set page height of document preferences of myDocument to "1p"
    set page width of document preferences of myDocument to "1p"
    --Reset the application default margin preferences to their former state.
    tell margin preferences
        set top to myY1
        set left to myX1
        set bottom to myY2
        set right to myX2
    end tell
end tell
```

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 `preview background color` property sets the color of the pasteboard in Preview mode, as shown in the following script fragment (from the PasteboardPreferences tutorial script):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell pasteboard preferences of myDocument
        --You can use either a number or a measurement string to set the
        --space above/below.
        set minimum space above and below to "12p"
        --You can set the pasteboard background color to any
        --of the predefined UIColor constants...
        set preview background color to gray
        --...or you can specify an array of RGB values
        --(with values from 0 to 255)
        --set preview Background Color to {192, 192, 192}
    end tell
end tell
```

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.)

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    set myPageWidth to page width of document preferences of myDocument
    set myPageHeight to page height of document preferences of myDocument
    tell page 1 of myDocument
        set myMarginPreferences to margin preferences
        --Place guides at the margins of the page.
        make guide with properties {orientation:vertical,
        location:left of myMarginPreferences}
        make guide with properties {orientation:vertical,
        location:(myPageWidth - (right of myMarginPreferences))}
        make guide with properties {orientation:horizontal,
        location:top of myMarginPreferences}
        make guide with properties {orientation:horizontal,
        location:(myPageHeight - (bottom of myMarginPreferences))}
        --Place a guide at the vertical center of the page.
        make guide with properties {orientation:vertical,
        location:(myPageWidth / 2)}
        --Place a guide at the horizontal center of the page.
        make guide with properties {orientation:horizontal,
        location:(myPageHeight / 2)}
    end tell
end tell
```

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 `fit to page` 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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        --Create a layer named "guide layer".
        set myLayer to make layer with properties {name:"guide layer"}
        --Add a series of guides to page 1.
        tell page 1
            --Create a guide on the layer we created above.
            make guide with properties {orientation:horizontal, <lb>
            location:"12p", item layer:myLayer}
            make guide with properties {item layer:myLayer, <lb>
            orientation:horizontal, location:"14p"}
            --Make a locked guide.
            make guide with properties {locked:true, <lb>
            orientation:horizontal, location:"16p"}
            --Set the view threshold of a guide.
            make guide with properties {view threshold:100, <lb>
            orientation:horizontal, location:"18p"}
            --Set the guide color of a guide using a UIColors constant.
            make guide with properties {guide color:gray, <lb>
            orientation:horizontal, location:"20p"}
            --Set the guide color of a guide using an RGB array.
            make guide with properties {guide color:{192, 192, 192}, <lb>
            orientation:horizontal, location:"22p"}
        end tell
    end tell
end tell
```

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

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell spread 1 of myDocument
        --Parameters (all optional): row count, column count, row gutter,
        --column gutter, guide color, fit margins, remove existing, layer.
        --Note that the create guides command does not take an RGB
        --array for the guide color parameter.
        create guides number of rows 4 number of columns 4 row gutter "1p"   <lb>
        column gutter "1p" guide color gray with fit margins and remove existing
    end tell
end tell
```

Setting grid preferences

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

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    set horizontal measurement units of view preferences of myDocument to points
    set vertical measurement units of view preferences of myDocument to points
    tell grid preferences of myDocument
        set baseline start to 56
        set baseline division to 14
        set baseline grid shown to true
        set horizontal gridline division to 14
        set horizontal grid subdivision to 5
        set vertical gridline division to 14
        set vertical grid subdivision to 5
        set document grid shown to true
    end tell
end tell
```

Snapping to guides and grids

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

```
tell application "Adobe InDesign CS5"
    set myDocument to active document
    tell guide preferences of myDocument
        set guides in back to true
        set guides locked to false
        set guides shown to true
        set guides snapto to true
    end tell
    tell grid preferences of myDocument
        set document grid shown to false
        set document grid snapto to true
        --Objects "snap" to the baseline grid when guidePreferences.guideSnapTo
        --is set to true.
        set baseline grid shown to true
    end tell
end tell
```

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 `view preferences` object, as shown in the following script fragment (from the ViewPreferences tutorial script):

```
tell application "Adobe InDesign CS5"
    set myDocument to active document
    tell view preferences of myDocument
        --Measurement unit choices are:
        --picas, points, inches, inches decimal, millimeters, centimeters, or ciceros
        --Set horizontal and vertical measurement units to points.
        set horizontal measurement units to points
        set vertical measurement units to points
    end tell
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    set myDocument to active document
    tell view preferences of myDocument
        set myOldXUnits to horizontal measurement units
        set myOldYUnits to vertical measurement units
        set horizontal measurement units to points
        set vertical measurement units to points
    end tell
    --At this point, you can perform any series of script actions that depend on
    --the measurement units you've set. At the end of the script, reset
    --the measurement units to their original state.
    tell view preferences of myDocument
        set horizontal measurement units to myOldXUnits
        set vertical measurement units to myOldYUnits
    end tell
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    if (count documents) > 0 then
        set myDocument to active document
        --If the document preset "myDocumentPreset"
        --does not already exist, create it.
        try
            set myDocumentPreset to document preset "myDocumentPreset"
        on error
            set myDocumentPreset to make document preset with properties
            {name:"myDocumentPreset"}
        end try
        --Fill in the properties of the document preset
        --with the corresponding
        --properties of the active document.
        tell myDocumentPreset
            --Note that the following gets the page margins from the margin preferences
            --of the document; to get the margin preferences from the active page,
            --replace "myDocument" with "active page of active window" in the
            --following line (assuming the active window is a layout window).
            set myMarginPreferences to margin preferences of myDocument
            set left to left of myMarginPreferences
            set right to right of myMarginPreferences
            set top to top of myMarginPreferences
            set bottom to bottom of myMarginPreferences
            set column count to column count of myMarginPreferences
            set column gutter to column gutter of myMarginPreferences
            set document bleed bottom offset to document bleed bottom offset
            of document preferences of myDocument
            set document bleed top offset to document bleed top offset
            of document preferences of myDocument
            set document bleed inside or left offset to document bleed inside
            or left offset of document preferences of myDocument
            set document bleed outside or right offset to document bleed outside
            or right offset of document preferences of myDocument
            set facing pages to facing pages of document preferences of myDocument
            set page height to page height of document preferences of myDocument
            set page width to page width of document preferences of myDocument
            set page orientation to page orientation of document preferences
            of myDocument
            set pages per document to pages per document of document preferences
            of myDocument
            set slug bottom offset to slug bottom offset of document preferences
            of myDocument
            set slug top offset to slug top offset of document preferences
            of myDocument
            set slug inside or left offset to slug inside or left offset
            of document preferences of myDocument
            set slug right or outside offset to slug right or outside offset
            of document preferences of myDocument
        end tell
    end if
end tell
```

Creating a document preset

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

```
tell application "Adobe InDesign CS5"
    --If the document preset "myDocumentPreset" does not already exist, create it.
    try
        set myDocumentPreset to document preset "myDocumentPreset"
    on error
        set myDocumentPreset to make document preset
        with properties {name:"myDocumentPreset"}
    end try
    --Fill in the properties of the document preset.
    tell myDocumentPreset
        set page height to "9i"
        set page width to "7i"
        set left to "4p"
        set right to "6p"
        set top to "4p"
        set bottom to "9p"
        set column count to 1
        set document bleed bottom offset to "3p"
        set document bleed top offset to "3p"
        set document bleed inside or left offset to "3p"
        set document bleed outside or right offset to "3p"
        set facing pages to true
        set page orientation to portrait
        set pages per document to 1
        set slug bottom offset to "18p"
        set slug top offset to "3p"
        set slug inside or left offset to "3p"
        set slug right or outside offset to "3p"
    end tell
end tell
```

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.)

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    --Set up the document.
    tell document preferences of myDocument
        set page height to "11i"
        set page width to "8.5i"
        set facing pages to true
        set page orientation to portrait
    end tell
    --Set the document's ruler origin to page origin. This is very important--
    --if you don't do this, getting objects to the correct position on the
    --page is much more difficult.
    set ruler origin of view preferences of myDocument to page origin
    tell master spread 1 of myDocument
        --Set up the left page (verso).
        tell margin preferences of page 1
            set column count to 3
            set column gutter to "1p"
            set bottom to "6p"
            --"left" means inside, "right" means outside.
            set left to "6p"
            set right to "4p"
```

```
            set top to "4p"
        end tell
        --Add a simple footer with a section number and page number.
        tell page 1
            set myTextFrame to make text frame   ¬
                  with properties {geometric bounds:{"61p", "4p", "62p", "45p"}}
            tell myTextFrame
                set contents of insertion point 1 to section marker
                set contents of insertion point 1 to Em space
                set contents of insertion point 1 to auto page number
                set justification of paragraph 1 to left align
            end tell
        end tell
        --Set up the right page (recto).
        tell margin preferences of page 2
            set column count to 3
            set column gutter to "1p"
            set bottom to "6p"
            --"left" means inside, "right" means outside.
            set left to "6p"
            set right to "4p"
            set top to "4p"
        end tell
        --Add a simple footer with a section number and page number.
        tell page 2
            set myTextFrame to make text frame   ¬
                  with properties {geometric bounds:{"61p", "6p", "62p", "47p"}}
            tell myTextFrame
                set contents of insertion point 1 to auto page number
                set contents of insertion point 1 to Em space
                set contents of insertion point 1 to section marker
                set justification of paragraph 1 to right align
            end tell
        end tell
    end tell
end tell
```

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

```
--Assumes that the active document has a master page named "B-Master"
--and at least three document pages.
tell application "Adobe InDesign CS5"
    tell active document
        set applied master of page 2 to master spread "B-Master"
    end tell
end tell
```

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

```
--Assumes that the active document has master spread named "B-Master"
--that is not the same as the first master spread in the document.
tell application "Adobe InDesign CS5"
    tell active document
        set applied master of page 2 of master spread 1 to
        master spread "B-Master"
    end tell
end tell
```

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.)

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell metadata preferences of myDocument
        set author to "Adobe"
        set copyright info URL to "http://www.adobe.com"
        set copyright notice to "This document is copyrighted."
        set copyright status to yes
        set description to "Example of xmp metadata scripting in InDesign CS"
        set document title to "XMP Example"
        set job name to "XMP_Example_2004"
        set keywords to {"animal", "mineral", "vegetable"}
        --The metadata preferences object also includes the read-only
        --creator, format, creationDate, modificationDate,
        --and serverURL properties that are automatically entered
        --and maintained by InDesign.
        --Create a custom XMP container, "email"
        set myNewContainer to create container item   ¬
                  namespace "http://ns.adobe.com/xap/1.0/" path "email"
        set property namespace "http://ns.adobe.com/xap/1.0/"   ¬
                  path "email/*[1]" value "someone@adobe.com"
    end tell
end tell
```

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.)

```
tell application "Adobe InDesign CS5"
    --Make a new document.
    set myDocument to make document
    tell myDocument
        tell document preferences
            set page width to "7i"
            set page height to "9i"
            set page orientation to portrait
        end tell
        tell margin preferences
            set top to ((14 * 4) & "pt") as string
            set left to ((14 * 4) & "pt") as string
            set bottom to "74pt"
            set right to ((14 * 5) & "pt") as string
        end tell
        --Set up the bleed and slug areas.
        tell document preferences
            --Bleed
            set document bleed bottom offset to "3p"
            set document bleed top offset to "3p"
            set document bleed inside or left offset to "3p"
            set document bleed outside or right offset to "3p"
            --Slug
            set slug bottom offset to "18p"
            set slug top offset to "3p"
            set slug inside or left offset to "3p"
            set slug right or outside offset to "3p"
        end tell
        --Create a color.
        try
            set myColor to color "PageNumberRed"
        on error
            set myColor to make color with properties {name:"PageNumberRed",
            model:process, color value:{20, 100, 80, 10}}
        end try
        --Next, set up some default styles.
        --Create up a character style for the page numbers.
        try
            set myCharacterStyle to character style "page_number"
        on error
            set myCharacterStyle to make character style
            with properties {name:"page_number"}
        end try
        set fill color of myCharacterStyle to color "PageNumberRed"
        --Create up a pair of paragraph styles for the page footer text.
        --These styles have only basic formatting.
        try
            set myParagraphStyle to paragraph style "footer_left"
        on error
            set myParagraphStyle to make paragraph style
            with properties {name:"footer_left"}
        end try
        --Create up a pair of paragraph styles for the page footer text.
        try
            set myParagraphStyle to paragraph style "footer_right"
        on error
            set myParagraphStyle to make paragraph style  ¬
                  with properties {name:"footer_right",  ¬
                  based on:paragraph style "footer_left", justification:right align}
        end try
```

```
--Create a layer for guides.
try
    set myLayer to layer "GuideLayer"
on error
    set myLayer to make layer with properties {name:"GuideLayer"}
end try
--Create a layer for the footer items.
try
    set myLayer to layer "Footer"
on error
    set myLayer to make layer with properties {name:"Footer"}
end try
--Create a layer for the slug items.
try
    set myLayer to layer "Slug"
on error
    set myLayer to make layer with properties {name:"Slug"}
end try
--Create a layer for the body text.
try
    set myLayer to layer "BodyText"
on error
    set myLayer to make layer with properties {name:"BodyText"}
end try
tell view preferences
    set ruler origin to page origin
    set horizontal measurement units to points
    set vertical measurement units to points
end tell
--Document baseline grid and document grid
tell grid preferences
    set baseline start to 56
    set baseline division to 14
    set baseline grid shown to false
    set horizontal gridline division to 14
    set horizontal grid subdivision to 5
    set vertical gridline division to 14
    set vertical grid subdivision to 5
    set document grid shown to false
end tell
--Document XMP information.
tell metadata preferences
    set author to "Adobe"
    set copyright info URL to "http:--www.adobe.com"
    set copyright notice to "This document is not copyrighted."
    set copyright status to no
    set description to "Example 7 x 9 book layout"
    set document title to "Example"
    set job name to "7 x 9 book layout template"
    set keywords to {"7 x 9", "book", "template"}
    set myNewContainer to create container item
    namespace "http://ns.adobe.com/xap/1.0/" path "email"
    set property namespace "http://ns.adobe.com/xap/1.0/"
    path "email/*[1]" value "someone@adobe.com"
end tell
--Set up the master spread.
tell master spread 1
    tell page 1
        set myMarginPreferences to margin preferences
        set myBottomMargin to (page height of document preferences of
```

```
    myDocument - (bottom of myMarginPreferences)
    set myLeftMargin to right of myMarginPreferences
    set myRightMargin to (page width of document preferences of myDocument)
    - (left of myMarginPreferences)
    make guide with properties {orientation:vertical,
    location:myRightMargin,item layer:layer "GuideLayer" of myDocument}
    make guide with properties {orientation:vertical,
    location:myLeftMargin,item layer:layer "GuideLayer" of myDocument}
    make guide with properties {orientation:horizontal,
    location:top of myMarginPreferences,
    item layer:layer "GuideLayer" of myDocument, fit to page:false}
    make guide with properties {orientation:horizontal,
    location:myBottomMargin,
    item layer:layer "GuideLayer" of myDocument, fit to page:false}
    make guide with properties {orientation:horizontal,
    location:myBottomMargin + 14,
    item layer:layer "GuideLayer" of myDocument, fit to page:false}
    make guide with properties {orientation:horizontal,
    location:myBottomMargin + 28,
    item layer:layer "GuideLayer" of myDocument, fit to page:false}
    set myLeftFooter to make text frame with properties
    {item layer:layer "Footer" of myDocument,
    geometric bounds:{myBottomMargin + 14, right of myMarginPreferences,
    myBottomMargin + 28, myRightMargin}}
    set contents of insertion point 1 of parent story of myLeftFooter
    to section marker
    set contents of insertion point 1 of parent story of myLeftFooter
    to Em space
    set contents of insertion point 1 of parent story of myLeftFooter
    to auto page number
    set applied character style of character 1 of parent story of
    myLeftFooter to character style "page_number" of myDocument
    set applied paragraph style of paragraph 1 of parent story of
    myLeftFooter to paragraph style "footer_left" of myDocument
    --Slug information.
    tell metadata preferences of myDocument
        set myEmail to get property
        namespace "http://ns.adobe.com/xap/1.0/" path "email/*[1]"
        set myDate to current date
        set myString to "Author:" & tab & author & tab & "Description:"
        & tab & description & return & "Creation Date:" & tab & myDate
        & tab & "Email Contact" & tab & myEmail
    end tell
    set myLeftSlug to make text frame with properties
    {item layer:layer "Slug" of myDocument,
    geometric bounds:{(page height of document preferences of myDocument)
     + 36, right of myMarginPreferences,
    (page height of document preferences of myDocument) + 144,
    myRightMargin}, contents:myString}
    tell parent story of myLeftSlug
        convert to table text 1
    end tell
    --Body text master text frame.
    set myLeftFrame to make text frame with properties   ¬
    {item layer:layer "BodyText" of myDocument,
    geometric bounds:{top of myMarginPreferences,
    right of myMarginPreferences, myBottomMargin, myRightMargin}}
end tell
tell page 2
    set myMarginPreferences to margin preferences
```

```
                set myLeftMargin to left of myMarginPreferences
                set myRightMargin to (page width of document preferences of myDocument)
                - (right of myMarginPreferences)
                make guide with properties {orientation:vertical,
                location:myLeftMargin,item layer:layer "GuideLayer" of myDocument}
                make guide with properties {orientation:vertical,
                location:myRightMargin, item layer:layer "GuideLayer" of myDocument}
                set myRightFooter to make text frame with properties
                {item layer:layer "Footer" of myDocument,
                geometric bounds:{myBottomMargin + 14, left of myMarginPreferences,
                myBottomMargin + 28, myRightMargin}}
                set contents of insertion point 1 of parent story of myRightFooter
                to auto page number
                set contents of insertion point 1 of parent story of myRightFooter
                to Em space
                set contents of insertion point 1 of parent story of myRightFooter
                to section marker
                set applied character style of character -1 of parent story of
                myRightFooter to character style "page_number" of myDocument
                set applied paragraph style of paragraph 1 of parent story of
                myRightFooter to paragraph style "footer_right" of myDocument
                --Slug information.
                set myRightSlug to make text frame with properties
                {item layer:layer "Slug" of myDocument,
                geometric bounds:{(page height of document preferences of myDocument)
                + 36, left of myMarginPreferences,(page height of document preferences
                of myDocument) + 144,myRightMargin}, contents:myString}
                tell parent story of myRightSlug
                    convert to table text 1
                end tell
                --Body text master text frame.
                set myRightFrame to make text frame with properties
                {item layer:layer "BodyText" of myDocument,
                geometric bounds:{top of myMarginPreferences,
                    left of myMarginPreferences, myBottomMargin, myRightMargin}}
            end tell
        end tell
        --Add section marker text--this text will appear in the footer.
        set marker of section 1 of myDocument to "Section 1"
        --When you link the master page text frames, one of the frames sometimes
        --becomes selected. Deselect it.
        select nothing
    end tell
end tell
```

Creating watermarks

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

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

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

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

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

Setting watermark preferences

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

```
tell watermark preferences
    set watermark visibility to true
    set watermark do print to true
    set watermark draw in back to true
    set watermark text to "Confidential"
    set watermark font family to "Arial"
    set watermark font style to "Bold"
    set watermark font point size to 72
    set watermark font color to red
    set watermark opacity to 60
    set watermark rotation to -45
    set watermark horizontal position to watermark h center
    set watermark horizontal offset to 0
    set watermark vertical position to watermark v center
    set watermark vertical offset to 0
end tell
```

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

```
tell watermark preferences of document 1
    set watermark visibility to true
    set watermark do print to true
    set watermark draw in back to true
    set watermark text to "Confidential"
    set watermark font family to "Arial"
    set watermark font style to "Bold"
    set watermark font point size to 72
    set watermark font color to blue
    set watermark opacity to 60
    set watermark rotation to -45
    set watermark horizontal position to watermark h center
    set watermark horizontal offset to 0
    set watermark vertical position to watermark v center
    set watermark vertical offset to 0
end tell
```

Disabling watermarks

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

```
tell application "Adobe InDesign CS5"
    set watermark visibility of watermark preferences to false
end tell
```

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

```
tell application "Adobe InDesign CS5"
    tell document 1
        set watermark visibility of watermark preferences to false
    end tell
end tell
```

Adjusting Page Sizes and Layout

Prior to InDesign CS5, pages in a document were limited to a single page size. InDesign CS5 removes this limitation and allows different page sizes within a single InDesign document. For information on setting the default page size, see "Defining page size and document length".

You can also apply geometric transformations to individual pages.

Selecting pages

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

```
--Given a document with four pages (1, 2, 3, 4)...
set myPages to pages of active document
--Select page 2 and 3.
select item 2 of myPages
select item 3 of myPages existing selection add to
--Select last page.
select item -1 of myPages existing selection add to
```

Resizing and reframing pages

You can resize or reframe page items on a page by scripting. With InDesign CS5, you can also apply the resize and reframe operations to pages to change their sizes.

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

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

```
--Given a document with four pages (1, 2, 3, 4)...
tell pages of active document
    --Resize page to two times bigger
    resize item 2 in inner coordinates from center anchor by multiplying current
dimensions by values {2, 2}
    --Resize page to 400 points width and 600 points height.
    resize item 3 in inner coordinates from center anchor by replacing current
dimensions with values {400, 600}
end tell
```

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

```
--Given a document with four pages (1, 2, 3, 4)...
tell item 2 of pages of active document
    --Make the page one inch wider and one inch higher.
    set myBounds to bounds
    set myY1 to item 1 of myBounds
    set myX1 to item 2 of myBounds
    set myY2 to (item 3 of myBounds) + 72
    set myX2 to (item 4 of myBounds) + 72
    reframe in inner coordinates opposing corners {{myX1, myY1}, {myX2, myY2}}
end tell
```

Transforming pages

Operations that change the geometry of objects are called transformations. Prior to InDesign CS5, the transform method could rotate, scale, shear, and move (translate) page items on a page. In InDesign CS5, the transform method can also be used on pages. For technical details about transformation architecture, refer to "Transforming Page Items".

To transform a page:

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

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

```
--Given a document with four pages (1, 2, 3, 4)...
set myDocument to active document
set myPages to pages of myDocument

--Rotate a page around its center point.
set myRotateMatrix to make transformation matrix with properties {counterclockwise
rotation angle:27}
my myTransform(item 1 of myPages, myRotateMatrix)

--Scale a page around its center point.
set myScaleMatrix to make transformation matrix with properties {horizontal scale
factor:0.8, vertical scale factor:0.8}
my myTransform(item 2 of myPages, myScaleMatrix)

--Shear a page around its center point.
set myShearMatrix to make transformation matrix with properties {clockwise shear
angle:30}
my myTransform(item 3 of myPages, myShearMatrix)

on myTransform(myPage, myTransformationMatrix)
    tell application "Adobe InDesign CS5"
        transform myPage in pasteboard coordinates from center anchor with matrix
myTransformationMatrix
    end tell
end myTransform
```

Master page overlay

Because pages can have multiple sizes, it is possible for a page and its master page to be different sizes. In addition to tracking which master is applied, pages now also maintain a matrix that determines how the

master page draws on the page. This is called the Master Page Overlay. When you select a page using the Page Tool on the Tools Panel, you can see how the master page is positioned by checking the Show Master Page Overlay checkbox on the control panel. You can move the overlay around with the mouse. InDesign achieves this by applying a transform to the master overlay matrix. Although the user interface allows only translation (moving *x* and *y*), you can do more by scripting. The following script shows how to transform a master page overlay. (For the complete script, see MasterPageTransform.)

```
--Given a document with four pages (1, 2, 3, 4)...
set myDocument to active document
set myPages to pages of myDocument

--Rotate master page overlay around its top-left corner.
set myRotateMatrix to make transformation matrix with properties {counterclockwise
rotation angle:27}
set master page transform of item 1 of myPages to myRotateMatrix

--Scale master page overlay around its top-left corner.
set myScaleMatrix to make transformation matrix with properties {horizontal scale
factor:0.5, vertical scale factor:0.5}
set master page transform of item 2 of myPages to myScaleMatrix

--Shear master page overlay around its top-left corner.
set myShearMatrix to make transformation matrix with properties {clockwise shear
angle:30}
set master page transform of item 3 of myPages to myShearMatrix

--Translate master page overlay 1 inch right and 2 inches down.
set myTranslateMatrix to make transformation matrix with properties {horizontal
translation:72, vertical translation:144}
set master page transform of item 4 of myPages to myTranslateMatrix
```

Printing a Document

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

```
tell application "Adobe InDesign CS5"
   print active document
end tell
```

Printing using page ranges

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

```
tell application "Adobe InDesign CS5"
   set page range of print preferences to "1-3, 10"
   print
end tell
```

Setting print preferences

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

```
--PrintPreferences.as
--An InDesign CS5 AppleScript
--Sets the print preferences of the active document.
tell application "Adobe InDesign CS5"
    --Get the bleed amounts from the document's bleed and add a bit.
    tell document preferences of active document
        set myX1Offset to document bleed inside or left offset + 3
        set myY1Offset to document bleed top offset + 3
        set myX2Offset to document bleed outside or right offset + 3
        set myY2Offset to document bleed bottom offset + 3
    end tell
    tell print preferences of active document
        --Properties corresponding to the controls in the General panel
        --of the Print dialog box.
        --activePrinterPreset is ignored in this example--we'll set our
        --own print preferences.
        --printer can be either a string (the name of the printer) or
        --postscript file.
        set printer to postscript file
        --Here's an example of setting the printer to a specific printer.
        --set printer to "AGFA-SelectSet 5000SF v2013.108"
        --If the printer property is the name of a printer, then the ppd property
        --is locked (and will return an error if you try to set it).
        try
            --set PPD to "Device Independent"
        end try
        --If the printer property is set to postscript file, the copies
        --property is unavailable. Attempting to set it will generate an error.
        --set copies to 1
        --If the printer property is set to Printer.postscript file, or if the
        --selected printer does not support collation, then the collating
        --property is unavailable. Attempting to set it will generate an error.
        --set collating to false
        set reverse order to false
        --The setting of color output determines the settings available
        --to almost all other properties in the print preferences.
        try
            set color output to separations
        end try
        --pageRange can be either PageRange.allPages or a page range string.
        set page range to all pages
        set print spreads to false
        set print master pages to false
        --If the printer property is set to postScript file, then
        --the print file property contains the file path to the output file.
        --set printFile to "yukino:test.ps"
        set sequence to all
        --If trapping is on, setting the following properties
        --will produce an error.
        try
            if trapping is off then
                set print blank pages to false
                set print guides grids to false
                set print nonprinting to false
            end if
        end try
        ------------------------------------------------------------------------
        --Properties corresponding to the controls in the Setup panel of
        --the Print dialog.
        ------------------------------------------------------------------------
```

```
set paper size to custom
--Page width and height are ignored if paper Size is not custom.
--set paper height to 1200
--set paper width to 1200
set print page orientation to portrait
set page position to centered
set paper gap to 0
set paper offset to 0
set paper transverse to false
set scale height to 100
set scale width to 100
set scale mode to scale width height
set scale proportional to true
--If trapping is on (application built in or Adobe inRip),
--attempting to set the following properties will produce an error.
if trapping is off then
    set thumbnails to false
    --The following properties are not needed because thumbnails is set to false.
    --set thumbnails per page to 4
    set tile to false
    --The following properties are not needed because tile is set to false.
    --set tiling overlap to 12
    --set tiling type to auto
end if
-------------------------------------------------------------------------
--Properties corresponding to the controls in the Marks and Bleed
--panel of the Print dialog.
-------------------------------------------------------------------------
--Set the following property to true to print all printer's marks.
--set all Printer Marks to true;
set use document bleed to print to false
--If use document bleed to print is true then setting any of the
-- bleed properties
--will result in an error.
set bleed bottom to myY2Offset
set bleed top to myY1Offset
set bleed inside to myX1Offset
set bleed outside to myX2Offset
--If any bleed area is greater than zero, then export the bleed marks.
if bleed bottom is equal to 0 and bleed top is equal to 0 and ¬
          bleed inside is equal to 0 and bleed outside is equal to 0 then
    set bleed marks to true
else
    set bleed marks to false
end if
set color bars to true
set crop marks to true
set include slug to print to false
set mark line weight to p125pt
set mark offset to 6
--set mark Type to default
set page information marks to true
set registration marks to true
-------------------------------------------------------------------------
--Properties corresponding to the controls in the Output panel
--of the Print dialog.
-------------------------------------------------------------------------
set negative to true
set color output to separations
--Note the lowercase "i" in "Builtin"
```

```
set trapping to application builtin
set screening to "175 lpi/2400 dpi"
set flip to none
--If trapping is on, attempting to set the following properties
--will generate an error.
if trapping is off then
    set print black to true
    set print cyan to true
    set print magenta to true
    set print yellow to true
end if
--Only change the ink angle and frequency when you want to override the
--screening set by the screening specified by the screening property.
--set black angle to 45
--set black frequency to 175
--set cyan angle to 15
--set cyan frequency to 175
--set magenta angle to 75
--set magenta frequency to 175
--set yellow angle to 0
--set yellow frequency to 175
--The following properties are not needed (because colorOutput
--is set to separations).
--set composite angle to 45
--set composite frequency to 175
--set simulate overprint to false
-------------------------------------------------------------------------
--Properties corresponding to the controls in the Graphics panel
--of the Print dialog.
-------------------------------------------------------------------------
set send image data to all image data
set font downloading to complete
try
    set download PPD fonts to true
end try
try
    set data format to binary
end try
try
    set PostScript level to level 3
end try
-------------------------------------------------------------------------
--Properties corresponding to the controls in the Color Management panel
--of the Print dialog.
-------------------------------------------------------------------------
--If the use color management property of color settings is false,
--attempting to set the following properties will return an error.
try
    set source space to use document
    set intent to use color settings
    set CRD to use document
    set profile to PostScript CMS
end try
-------------------------------------------------------------------------
```

```
        --Properties corresponding to the controls in the Advanced panel
        --of the Print dialog.
        ------------------------------------------------------------------------
        set OPI image replacement to false
        set omit bitmaps to false
        set omit EPS to false
        set omit PDF to false
        --The following line assumes that you have a flattener preset
        --named "high quality flattener".
        try
            set flattener preset name to "high quality flattener"
        end try
        set ignore spread overrides to false
    end tell
end tell
```

Printing with printer presets

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

```
tell application "Adobe InDesign CS5"
    --The following line assumes that you have defined a print preset
    --named "myPrintPreset".
    print active document using "myPrintPreset"
end tell
```

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.)

```
tell application "Adobe InDesign CS5"
    tell active document
        --You'll have to fill in a valid file path for your system.
        export format PDF type to "yukino:test.pdf" without showing options
    end tell
end tell
```

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

```
tell application "Adobe InDesign CS5"
    tell active document
        --You'll have to fill in a valid file path for your system and
        --a valid PDF export preset name.
        export format PDF type to "yukino:test.pdf" using PDF export preset
        "myTestPreset" without showing options
    end tell
end tell
```

Setting PDF export options

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

```
tell application "Adobe InDesign CS5"
    --Get the bleed amounts from the document's bleed.
    tell document preferences of active document
        set myX1Offset to document bleed inside or left offset
        set myY1Offset to document bleed top offset
        set myX2Offset to document bleed outside or right offset
        set myY2Offset to document bleed bottom offset
    end tell
    tell PDF export preferences
        --Basic PDF output options.
        set page range to all pages
        set acrobat compatibility to acrobat 6
        set export guides and grids to false
        set export layers to false
        set export nonprinting objects to false
        set export reader spreads to false
        set generate thumbnails to false
        try
            set ignore spread overrides to false
        end try
        set include bookmarks to true
        set include hyperlinks to true
        try
            set include ICC profiles to true
        end try
        set include slug with PDF to false
        set include structure to false
        set interactive elements option to do not include
        --Setting subset fonts below to zero disallows font subsetting
        --set subset fonts below to some other value to use font subsetting.
        set subset fonts below to 0
        --Bitmap compression/sampling/quality options.
        set color bitmap compression to zip
        set color bitmap quality to eight bit
        set color bitmap sampling to none
        --threshold to compress color is not needed in this example.
        --color bitmap sampling dpi is not needed when color bitmap sampling
        --is set to none.
        set grayscale bitmap compression to zip
        set grayscale bitmap quality to eight bit
        set grayscale bitmap sampling to none
        --threshold to compress gray is not needed in this example.
        --grayscale bitmap sampling dpi is not needed when grayscale bitmap
        --sampling is set to none.
        set monochrome bitmap compression to zip
        set monochrome bitmap sampling to none
        --threshold to compress monochrome is not needed in this example.
        --monochrome bitmap sampling dpi is not needed when monochrome bitmap
        --sampling is set to none.
        --Other compression options.
        set compression type to Compress None
        set compress text and line art to true
        set crop images to frames to true
        set optimize PDF to true
```

```
        --Printers marks and prepress options.
        set bleed bottom to myY2Offset
        set bleed top to myY1Offset
        set bleed inside to myX1Offset
        set bleed outside to myX2Offset
        --If any bleed area is greater than zero, then export the bleed marks.
        if bleed bottom is 0 and bleed top is 0 and bleed inside is 0 and
        bleed outside is 0 then
            set bleed marks to true
        else
            set bleed marks to false
        end if
        set color bars to true
        --Color tile size and gray tile size are not used
        --unless the compression method chosen is JPEG 2000.
        --set color tile size to 256
        --set Gray tile size to 256
        set crop marks to true
        set omit bitmaps to false
        set omit EPS to false
        set omit PDF to false
        set page information marks to true
        set page marks offset to "12 pt"
        set PDF color space to unchanged color space
        set PDF mark type to default
        set printer mark weight to p125pt
        set registration marks to true
        --simulate overprint is only available when the export standard
        --is PDF/X-1a or PDF/X-3
        --set simulate overprint to false
        set use document bleed with PDF to true
        --Set viewPDF to true to open the PDF in Acrobat or Adobe Reader.
        set view PDF to false
    end tell
    --Now export the document.
    tell active document
        --You'll have to fill in a valid file path for your system.
        export format PDF type to "yukino:test.pdf" without showing options
    end tell
end tell
```

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.)

```
--Assumes you have a document open, and that that document
--contains at least 12 pages.
tell application "Adobe InDesign CS5"
    tell PDF export preferences
        --page range can be either all pages or a page range string
        --(just as you would enter it in the Print or Export PDF dialog box).
        set page range to "1, 3-6, 7, 9-11, 12"
    end tell
    tell active document
        --You'll have to fill in a valid file path for your system and
        --a valid PDF export preset name.
        export format PDF type to "yukino:test.pdf" using
        PDF export preset "myTestPreset" without showing options
    end tell
end tell
```

Exporting individual pages to PDF

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

```
--Display a "choose folder" dialog box.
tell application "Adobe InDesign CS5"
    if (count documents) is not equal to 0 then
        my myChooseFolder()
    else
        display dialog "Please open a document and try again."
    end if
end tell
on myChooseFolder()
    set myFolder to choose folder with prompt "Choose a Folder"
    --Get the folder name (it'll be returned as a Unicode string)
    set myFolder to myFolder as string
    --Unofficial technique for changing Unicode folder name to plain text string.
    set myFolder to «class ktxt» of (myFolder as record)
    if myFolder is not equal to "" then
        my myExportPages(myFolder)
    end if
end myChooseFolder
on myExportPages(myFolder)
    tell application "Adobe InDesign CS5"
        set myDocument to active document
        set myDocumentName to name of myDocument
        set myDialog to make dialog with properties {name:"File Naming Options"}
        tell myDialog
            tell (make dialog column)
                tell (make dialog row)
                    make static text with properties {static label:"Base name:"}
                    set myBaseNameField to make text editbox with properties
                    {edit contents:myDocumentName, min width:160}
                end tell
            end tell
        end tell
        set myResult to show myDialog
        if myResult is true then
            --The name of the exported files will be the base name + the value
            --of the counter + ".pdf".
            set myBaseName to edit contents of myBaseNameField
            --Remove the dialog box from memory.
```

```
            destroy myDialog
            repeat with myCounter from 1 to (count pages in myDocument)
                set myPageName to name of page myCounter of myDocument
                set page range of PDF export preferences to name of page
                myCounter of myDocument
                --Generate a file path from the folder name, the base document
                --name, and the page name.
                --Replace any colons in the page name (e.g., "Sec1:1") so that
                --they don't cause problems with file naming.
                set myPageName to my myReplace(myPageName, ":", "_")
                set myFilePath to myFolder & myBaseName & "_" & myPageName & ".pdf"
                tell myDocument
                    --The export command will fail if you provide the file path
                    --as Unicode text--that's why we had to convert the folder name
                    --to plain text.
                    export format PDF type to myFilePath
                end tell
            end repeat
        else
            destroy myDialog
        end if
    end tell
end myExportPages
on myReplace(myString, myFindString, myChangeString)
    set AppleScript's text item delimiters to myFindString
    set myTextList to every text item of (myString as text)
    set AppleScript's text item delimiters to myChangeString
    set myString to myTextList as string
    set AppleScript's text item delimiters to ""
    return myString
end myReplace
```

Exporting PDF with Interactive Features

The following script shows how to export a document with interactive features as a PDF. (For the complete script, see ExportInteractivePDF.)

```
--Given a document "myDocument" containing at least two spreads...
repeat with myCounter from 1 to (count spreads of myDocument)
    set page transition type of spread myCounter of myDocument to wipe transition
    set page transition direction of spread myCounter of myDocument to left to right
    set page transition duration of spread myCounter of myDocument to medium
end repeat
set myDesktopFolder to path to desktop as string
set myFile to myDesktopFolder & "InteractivePDF.pdf"
tell myDocument
    export format interactive PDF to myFile without showing options
end tell
```

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.)

```
tell application "Adobe InDesign CS5"
    set page range of EPS export preferences to all pages
    tell active document
        --You'll have to fill in a valid file name for your system.
        --Files will be named "myFile_01.eps", "myFile_02.eps", and so on.
        set myFileName to "yukino:myFile.eps"
        export format EPS type to myFileName without showing options
    end tell
end tell
```

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.)

```
--Assumes you have a document open, and that that document
--contains at least 12 pages.
tell application "Adobe InDesign CS5"
    tell EPS export preferences
        --page range can be either all pages or a page range string
        --(just as you would enter it in the Print or Export EPS dialog box).
        set page range to "1, 3-6, 7, 9-11, 12"
    end tell
    tell active document
        --You'll have to fill in a valid file path for your system.
        export format EPS type to "yukino:test.eps" without showing options
    end tell
end tell
```

Exporting as EPS with file naming

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

```
--Display a "choose folder" dialog box.
tell application "Adobe InDesign CS5"
    if (count documents) is not equal to 0 then
        my myChooseFolder()
    else
        display dialog "Please open a document and try again."
    end if
end tell
on myChooseFolder()
    set myFolder to choose folder with prompt "Choose a Folder"
    --Get the folder name (it'll be returned as a Unicode string)
    set myFolder to myFolder as string
    --Unofficial technique for changing Unicode folder name to plain text string.
    set myFolder to «class ktxt» of (myFolder as record)
    if myFolder is not equal to "" then
        my myExportPages(myFolder)
    end if
```

```
end myChooseFolder
on myExportPages(myFolder)
    tell application "Adobe InDesign CS5"
        set myDocument to active document
        set myDocumentName to name of myDocument
        set myDialog to make dialog with properties {name:"ExportPages"}
        tell myDialog
            tell (make dialog column)
                tell (make dialog row)
                    make static text with properties {static label:"Base Name:"}
                    set myBaseNameField to make text editbox with properties
                    {edit contents:myDocumentName, min width:160}
                end tell
            end tell
        end tell
        set myResult to show myDialog
        if myResult is true then
            --The name of the exported files will be the base name + the
            --value of the counter + ".pdf".
            set myBaseName to edit contents of myBaseNameField
            --Remove the dialog box from memory.
            destroy myDialog
            repeat with myCounter from 1 to (count pages in myDocument)
                --Get the name of the page and assign it to the variable "myPageName"
                set myPageName to name of page myCounter of myDocument
                --Set the page range to the name of the specific page.
                set page range of EPS export preferences to myPageName
                --Generate a file path from the folder name, the base document
                --name, and the page name.
                --Replace any colons in the page name (e.g., "Sec1:1") so that
                --they don't cause problems with file naming.
                set myPageName to my myReplace(myPageName, ":", "_")
                set myFilePath to myFolder & myBaseName & "_" & myPageName & ".eps"
                tell myDocument
                    export format EPS type to myFilePath without showing options
                end tell
            end repeat
        else
            destroy myDialog
        end if
    end tell
end myExportPages
on myReplace(myString, myFindString, myChangeString)
    set AppleScript's text item delimiters to myFindString
    set myTextList to every text item of (myString as text)
    set AppleScript's text item delimiters to myChangeString
    set myString to myTextList as string
    set AppleScript's text item delimiters to ""
    return myString
end myReplace
```

4 Working with Layers

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

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

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

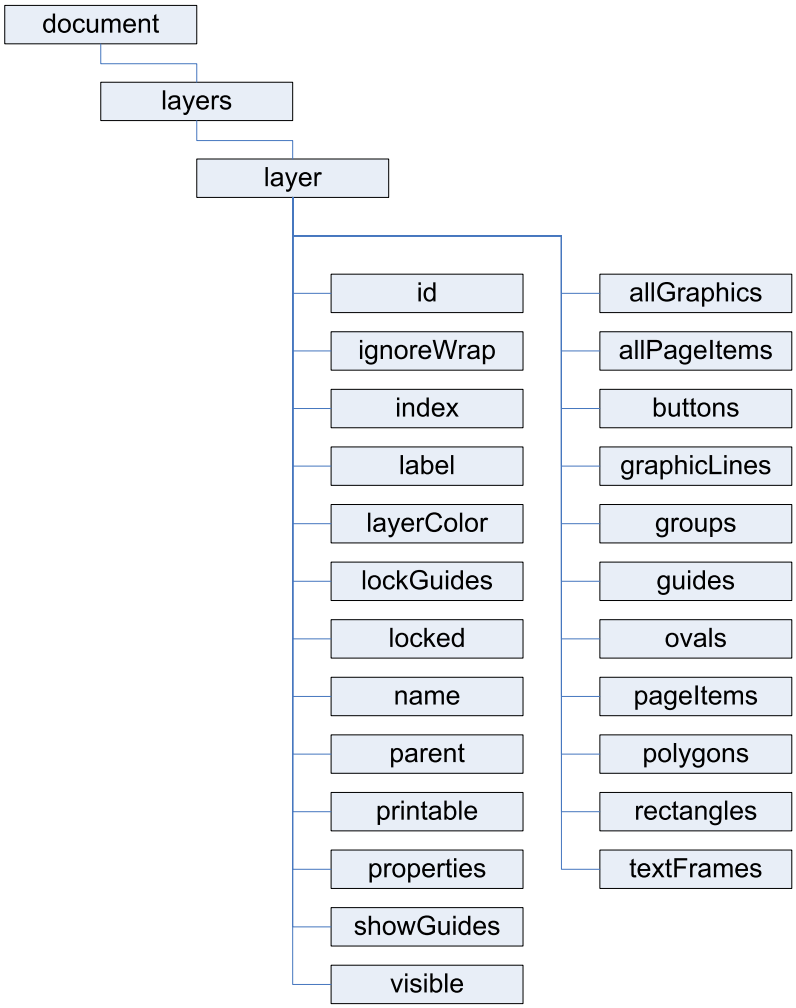
Understanding the Layer Object Model

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

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

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

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



Scripting Layers

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

Creating layers

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

```
--Given a document "myDocument"...
tell myDocument
    set myLayer to make layer
end tell
```

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

Referring to layers

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

Getting the active layer

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

```
--Given a document "myDocument"...
tell myDocument
    set myLayer to active layer
end tell
```

Referring to layers by layer index

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

```
--Given a document "myDocument"...
tell myDocument
    set myTargetLayer to active layer
    set myLayerName to name of myTargetLayer
    repeat with myCounter from 1 to (count layers)
    --If the layer is not the target layer, hide it.
    set myName to name of layer myCounter
        if myName is not equal to myLayerName then
            set visible of layer myCounter of myDocument to false
        end if
    end repeat
end tell
```

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

Referring to layers by layer name

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

```
set myLayer to layer "Text Layer" of document 1
```

Deleting layers

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

```
--Given a document "myDocument" containing a layer named "Delete This Layer"...
set myLayer to layer "Delete This Layer" of myDocument
tell myLayer
    delete
end tell
```

Moving layers

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

```
--Given a document "myDocument" containing at least two layers...
set myLayerA to layer 1 of myDocument
set myLayerB to layer 2 of myDocument
tell myLayer1
    move to after myLayer2
end tell
```

Duplicating layers

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

```
-Given a layer "myLayer"...
tell myLayer
    set myNewLayer to duplicate
end tell
```

Merging layers

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

```
--Given the layers "myLayer1" and "myLayer2"...
tell myLayer1
    merge with myLayer2
end tell
```

Assigning page items to layers

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

```
--Given a reference to a page "myPage," and a document "myDocument"...
tell myPage
    --Create a text frame on a layer named "TextFrames"
    set myTextFrame to make text frame with properties
    {item layer:layer "TextFrames", geometric bounds:{72, 72, 144, 144}}
    --Create a rectangle on the current target layer.
    set myRectangle to make rectangle with properties
    {geometric bounds:{72, 144, 144, 216}}
    --Move the rectangle to a specific layer.
    set item layer of myRectangle to layer "Rectangles" of myDocument
    --Create a series of ovals.
    repeat with myCounter from 72 to 172 by 10
        make oval with properties {geometric bounds:{216, myCounter, 226,
        myCounter + 10}}
    end repeat
    --Move all of the ovals on the page to a specific layer.
    repeat with myCounter from 1 to (count ovals)
        set item layer of oval myCounter to layer "Ovals" of myDocument
    end repeat
end tell
```

Setting layer properties

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

Setting basic layer properties

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

```
--Given a document "myDocument"...
tell myDocument
    set myLayer to make layer
    set name of myLayer to "myLayer"
    set layer color of myLayer to charcoal
    set ignore wrap of myLayer to false
    set visible of myLayer to true
end tell
```

Working with layer guides

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

```
--Given a document "myDocument" and a page "myPage" containing at least one guide...
tell myDocument
    set myLayer to make layer
    --Move all of the guides on the page to the new layer.
    tell myPage
        repeat with myCounter from (count guides) to 1 by -1
            set item layer of guide myCounter to myLayer
        end repeat
    end tell
    set lock guides of myLayer to true
    set show guides of myLayer to true
end tell
```

Controlling layer printing and visibility

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

```
--Given a document "myDocument" containing layers named "Background,"
--"Language A,", "Language B," and "Language C," export the "Background"
--layer and each "Language" layer to PDF as separate PDF files...
set myList to {"A", "B", "C"}
set myPath to path to desktop as string
tell myDocument
    repeat with myCounter from 1 to 3
        set myVersion to "Language " & item myCounter of myList
        repeat with myLayerCounter from 1 to (count layers)
            if name of layer myLayerCounter is equal to myVersion or
            name of layer myLayerCounter is equal to "Background" then
                set visible of layer myLayerCounter to true
                set printable of layer myLayerCounter to true
            else
                set visible of layer myLayerCounter to false
                set printable of layer myLayerCounter to false
            end if
        end repeat
        set myFilePath to myPath & myVersion & ".pdf"
        export to myFilePath format PDF type
    end repeat
end tell
```

Locking layers

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

```
--Given a document "myDocument"...
tell myDocument
    set myLayer to active layer
    repeat with myCounter from ((index of myLayer) + 1) to (count layers)
        set locked of layer myCounter to true
    end repeat
end tell
```

5 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.

```
--Given a page "myPage", create a new rectangle at the default size and location...
tell myPage
    set myRectangle to make rectangle
end tell
```

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

```
--Given a page "myPage", create a new rectangle and specify its size and location...
tell myPage
    set myRectangle to make rectangle with properties
    {geometric bounds:{72, 72, 144, 144}}
end tell
```

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:

```
set myPageItemType to class of 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 class of to get the specific type.

```
--Given a generic page item "myPageItem"...
set myType to class of myPageItem
display dialog 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 `page items` collection of the container object. This gives you a collection of the top level page items inside the object. For example:

```
set myPageItems to page items of page 1 of document 1
```

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 `all page items` property.

```
set myAllPageItems to all page items of page 1 of document 1
```

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`.

```
set myPageItems to page items whose label is "myLabel" of page 1 of document 1
```

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 entire path 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 entire path property can contain anchor points only, or a anchor points and control handles. Here is an example array containing only anchor point locations:

```
{{x1, y1}, {x2, y2}, ...}
```

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

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

```
{{{xL1, YL1}, {x1, y1}, {xR1, yR1}}, {{xL2, YL2}, {x2, y2}, {xR2, yR2}}, ...}
```

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

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

```
{{{xL1, YL1}, {x1, y1}, {xR1, yR1}}, {x2, y2}, ...}
```

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

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

```
--Given a graphic line "myGraphicLine"...
tell path 1 of myGraphicLine
    set myPathPoint to add path point
end tell
--Move the path point to a specific location.
set anchor of myPathPoint to {144, 144}
```

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

```
--Given a polygon "myPolygon", remove the
--last path point in the first path.
tell path 1 of myPolgon
    delete path point -1
end tell
```

Grouping Page Items

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

```
--Given a page "myPage" containing at least two ovals and two rectangles...
tell myPage
    set myRectangleA to rectangle 1
    set myRectangleB to rectangle 2
    set Oval to oval 1
    set Oval to oval 2
    --Group the items.
    make group with properties{group items:
    {myRectangleA, myRectangleB, myOvalA, myOvalB})
end tell
```

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

```
--Given a group "myGroup"...
tell myGroup
set myPageItems to ungroup
end tell
```

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: move to and move by. Both parameters consist of an array of two measurement units, consisting of a horizontal value and a vertical value. move to specifies an absolute move to the location specified by the array, relative to the current location of the zero

point. move by specifies how far to move the page item relative to the current location of the page item itself. The Move script shows the difference between these two approaches.

```
--Given a reference to a rectangle "myRectangle"...
--Move the rectangle to the location (12, 12).
--Absolute move:
tell myRectangle
    move to {12, 12}
end tell
--Move the rectangle *by* 12 points horizontally, 12 points vertically.
--Relative move:
tell myRectangle
    move by {12, 12}
end tell
--Move the rectangle to another page (rectangle appears at (0,0);
tell document 1
set myPage to make page
end tell
tell myRectangle
    move to myPage
end tell
--To move a page item to another document, use the duplicate method.
```

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

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

```
--Given a reference to a rectangle "myRectangle"...
--Duplicate the rectangle and move the
--duplicate to the location (12, 12).
--Absolute move:
tell myRectangle
    set myDuplicate to duplicate to {12, 12}
end tell
--Duplicate the rectangle and move the duplicate *by* 12
--points horizontally, 12 points vertically.
--Relative move:
tell myRectangle
    set myDuplicate to duplicate by {12, 12}
end tell
--Duplicate the rectangle to another page (rectangle appears at (0,0).
tell document 1
    set myPage to make page
end tell
tell myRectangle
     set myDuplicate do duplicate to myPage
end tell
--Duplicate the rectangle to another document.
set myDocument to make document
tell myRectangle
    myDuplicate to page 1 of myDocument
end tell
```

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 make compound path command of a page item, as shown in the following script fragment (for the complete script, refer to the MakeCompoundPath script).

```
--Given a rectangle "myRectangle" and an Oval "myOval"...
tell myRectangle
    make compound path with myOval
end tell
```

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 release compound path command of a page item, as shown in the following script fragment (for the complete script, refer to the ReleaseCompoundPath script).

```
--Given a polygon "myPolygon"...
tell myPolygon
    set myPageItems to release compound path
end tell
```

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 a list of page items to use as the basis for the operation (just as you select a series of page items before choosing the Pathfinder operation in the user interface).

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

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

```
--Given a rectangle "myRectangle" and an Oval "myOval"...
tell myRectangle
    add path with myOval
end tell
```

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

```
--Given a rectangle "myRectangle" and an Oval "myOval"...
tell myRectangle
    exclude overlap path with myOval
end tell
```

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

```
--Given a rectangle "myRectangle" and an Oval "myOval"...
tell myRectangle
    intersect path with myOval
end tell
```

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

```
--Given a rectangle "myRectangle" and an Oval "myOval"...
tell myRectangle
    minus back with myOval
end tell
```

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

```
--Given a rectangle "myRectangle" and an Oval "myOval"...
tell myOval
    subtract with myRectangle
end tell
```

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 convert shape command, as demonstrated in the following script fragment (for the complete script, refer to ConvertShape).

```
--Given a rectangle "myRectangle"...
tell myRectangle
    convert shape given convert to rounded rectangle
end tell
```

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

```
--Given a rectangle "myRectangle"...
tell myRectangle
    convert shape given convert to open path
end tell
```

Arranging page items

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

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

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

```
--Given a rectangle "myRectangle" and a layer "myLayer",
--send the rectangle to the layer...
tell myRectangle
    set item layer to layer "myLayer" of document 1
end tell
```

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

```
--Given a layer "myLayer", move the layer behind
--the default layer.
tell myLayer
    move to end of layers of document 1
end tell
```

Transforming Page Items

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

Using the transform method

The `transform` method requires a transformation matrix (`transformation matrix`) 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 71. In addition, you specify the center of

transformation, or transformation origin. For more on specifying the transformation origin, see "Transformation origin" on page 72.

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

```
--Rotate a rectangle "myRectangle" around its center point.
set myRotateMatrix to make transformation matrix with properties {counterclockwise
rotation angle:27}
transform myRectangle in pasteboard coordinates from center anchor with matrix
myRotateMatrix
--Scale a rectangle "myRectangle" around its center point.
set myScaleMatrix to make transformation matrix with properties {horizontal scale
factor:.5, vertical scale factor:.5}
transform myRectangle in pasteboard coordinates from center anchor with matrix
myScaleMatrix
--Shear a rectangle "myRectangle" around its center point.
set myShearMatrix to make transformation matrix with properties {clockwise shear angle:
30}
transform myRectangle in pasteboard coordinates from center anchor with matrix
myShearMatrix
--Rotate a rectangle "myRectangle" around a specified ruler point ({72, 72}).
set myRotateMatrix to make transformation matrix with properties {counterclockwise
rotation angle:27}
transform myRectangle in pasteboard coordinates from {{72, 72}, center anchor} with
matrix myRotateMatrix with considering ruler units
--Scale a rectangle "myRectangle" around a specified ruler point ({72, 72}).
set myScaleMatrix to make transformation matrix with properties {horizontal scale
factor:.5, vertical scale factor:.5}
transform myRectangle in pasteboard coordinates from {{72, 72}, center anchor} with
matrix myScaleMatrix with considering ruler units
```

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

Working with transformation matrices

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

```
--Scale a transformation matrix by 50% in both
--horizontal and vertical dimensions.
set myTransformationMatrix to scale matrix myTransformationMatrix
horizontally by .5 vertically by .5
--Rotate a transformation matrix by 45 degrees.
set myTransformationMatrix to rotate matrix by angle 45
--Shear a transformation matrix by 15 degrees.
set myTransformationMatrix to shear matrix by angle 15
```

When you use the `rotate matrix` 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.

```
set myRectangle to rectangle 1 of page 1 of document 1
set myTransformationMatrix to make transformation matrix
--rotate matrix can take the following parameters: byAngle, byCosine, bySine;
--The following statements are equivalent (0.25881904510252 is the sine of 15 degrees,
0.96592582628907 is the cosine).
set myTransformationMatrix to rotate matrix myTransformationMatrix by angle 15
set myTransformationMatrix to rotate matrix myTransformationMatrix by cosine
0.965925826289
set myTransformationMatrix to rotate matrix myTransformationMatrix by sine
0.258819045103
--Rotate the rectangle by the rotated matrix--45 degrees.
transform myRectangle in pasteboard coordinates from center anchor with matrix
myTransformationMatrix
```

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

```
set myRectangle to rectangle 1 of page 1 of document 1
set myTransformationMatrix to make transformation matrix
--shear matrix can take the following parameters: by angle, by slope
--The following statements are equivalent. slope = rise/run
--so a 45 degree slope is 1.
set myTransformationMatrix to shear matrix myTransformationMatrix by slope 1
--Shear the rectangle.
transform myRectangle in pasteboard coordinates from center anchor with matrix
myTransformationMatrix
```

You can get the inverse of a transformation matrix using the `invert matrix`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 to rectangle 1 of page 1 of document 1
set myTransformationMatrix to make transformation matrix with properties
{counterclockwise rotation angle:30, horizontal translation:12, vertical
translation:12}
transform myRectangle in pasteboard coordinates from center anchor with matrix
myTransformationMatrix
set myNewRectangle to duplicate myRectangle
--Move the duplicated rectangle to the location of the original
--rectangle by inverting, then applying the transformation matrix.
set myTransformationMatrix to invert matrix myTransformationMatrix
transform myNewRectangle in pasteboard coordinates from center anchor with matrix
myTransformationMatrix
```

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

```
set myTransformationMatrixA to make transformation matrix with properties
{counterclockwise rotation angle:30}
set myTransformationMatrixB to make transformation matrix with properties {horizontal
translation:72, vertical translation:72}
set myRectangle to rectangle -1 of page 1 of document 1
set myNewRectangle to duplicate myRectangle
--Rotate the duplicated rectangle.
transform myNewRectangle in pasteboard coordinates from center anchor with matrix
myTransformationMatrixA
set myNewRectangle to duplicate myRectangle
--Move the duplicate (unrotated) rectangle.
transform myNewRectangle in pasteboard coordinates from center anchor with matrix
myTransformationMatrixB
--Merge the two transformation matrices.
set myTransformationMatrix to catenate matrix myTransformationMatrixA with matrix
myTransformationMatrixB
set myNewRectangle to duplicate myRectangle
--The duplicated rectangle will be both moved and rotated.
transform myNewRectangle in pasteboard coordinates from center anchor with matrix
myTransformationMatrix
```

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

```
set myRectangle to rectangle -1 of page 1 of document 1
--Note that transform values of always returns a list containing a
--single transformation matrix.
set myTransformArray to transform values of myRectangle in pasteboard coordinates
set myTransformationMatrix to item 1 of myTransformArray
set myRotationAngle to counterclockwise rotation angle of myTransformationMatrix
set myShearAngle to clockwise shear angle of myTransformationMatrix
set myXScale to horizontal scale factor of myTransformationMatrix
set myYScale to vertical scale factor of myTransformationMatrix
set myXTranslate to horizontal translation of myTransformationMatrix
set myYTranslate to vertical translation of myTransformationMatrix
set myString to "Rotation Angle: " & myRotationAngle & return
set myString to myString & "Shear Angle: " & myShearAngle & return
set myString to myString & "Horizontal Scale Factor: " & myXScale & return
set myString to myString & "Vertical Scale Factor: " & myYScale & return
set myString to myString & "Horizontal Translation: " & myXTranslate & return
set myString to myString & "Vertical Translation: " & myYTranslate & return & return
set myString to myString & "Note that the Horizontal Translation and" & return
set myString to myString & "Vertical Translation values are the location" & return
set myString to myString & "of the center anchor in pasteboard coordinates."
display dialog (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 `pasteboard coordinates` 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:

- **`pasteboard coordinates`** is the coordinate space of the entire InDesign document. This coordinate space extends behind all spreads in a document. It does not correspond to InDesign's rulers or zero point, nor does it have anything to do with the pasteboard area you can see around pages in the InDesign user interface. Transformations applied to objects have no effect on this coordinate space (e.g., the angle of the horizontal and vertical axes do not change).
- **`parent coordinates`** 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.
- **`inner coordinates`** is the coordinate space in which the object itself was created.
- **`spread coordinates`** 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 to rectangle 1 of group 1 of page 1 of document 1
set myString to "The page contains a group which has been" & return
set myString to myString & "rotated 45 degrees (counterclockwise)." & return
set myString to myString & "The rectangle inside the group was" & return
set myString to myString & "rotated 45 degrees counterclockwise before" & return
set myString to myString & "it was added to the group." & return & return
set myString to myString & "Watch as we apply a series of scaling" & return
set myString to myString & "operations in different coordinate spaces."
display dialog myString
set myTransformationMatrix to make transformation matrix with properties {horizontal
scale factor:2}
--Transform the rectangle using inner coordinates.
transform myRectangle in inner coordinates from center anchor with matrix
myTransformationMatrix
--Select the rectangle and display an alert.
select myRectangle
display dialog "Transformed using inner coordinates."
--Undo the transformation.
tell document 1 to undo
--Transform using parent coordinates.
transform myRectangle in parent coordinates from center anchor with matrix
myTransformationMatrix
select myRectangle
display dialog "Transformed using parent coordinates."
tell document 1 to undo
--Transform using pasteboard coordinates.
transform myRectangle in pasteboard coordinates from center anchor with matrix
myTransformationMatrix
select myRectangle
display dialog "Transformed using pasteboard coordinates."
tell document 1 to 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.

    ```
    center anchor
    ```

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

    ```
    {{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.

    ```
    {{72, 144}, center anchor}
    ```

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

    ```
    {72, 72}
    ```

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

    ```
    {{72, 72}, parent coordinates}
    ```

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

    ```
    {{72, 72}}
    ```

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

```
set myRectangle to rectangle 1 of document 1
set myString to "Watch as we rotate the rectangle using different anchor points," &
return
set myString to myString & "bounds types, and coordinate spaces." & return & return
set myString to myString & "You might have to drag the alert aside to" & return
set myString to myString & "see the effect of the transformation."
set myNewRectangle to duplicate myRectangle
set myTransformationMatrix to make transformation matrix with properties
{counterclockwise rotation angle:30}
--Rotate around the duplicated rectangle's center point.
transform myNewRectangle in pasteboard coordinates from center anchor with matrix
myTransformationMatrix
--Select the rectangle and display an alert.
select myNewRectangle
display dialog "Rotated around center anchor."
--Undo the transformation.
tell document 1 to undo
--Rotate the rectangle around the ruler location [-100, -100]. Note that the anchor
point specified here specifes the page
--containing the point--*not* that transformation point itself. The transformation gets
the ruler coordinate [-100, -100] based
```

```
--on that page. Setting the considerRulerUnits parameter to true makes certain that the
transformation uses the current
--ruler units.
transform myNewRectangle in pasteboard coordinates from {{-100, -100}, top left anchor}
with matrix myTransformationMatrix with considering ruler units
--Move the page guides to reflect the transformation point.
tell guide 1 of page 1 of document 1 to set location to -100
tell guide 2 of page 1 of document 1 to set location to -100
--Select the rectangle and display an alert.
select myNewRectangle
display dialog "Rotated around -100x, -100y."
--Undo the transformation and the guide moves.
tell document 1 to undo
tell document 1 to undo
tell document 1 to 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 myRectangle to rectangle 1 of group 1 of page 1 of document 1
--Get ruler coordinate {72, 72} in pasteboard coordinates.
set myPageLocation to resolve myRectangle location {{72, 72}, top right anchor} in
pasteboard coordinates with considering ruler units
--resolve returns a list containing a single item.
set myPageLocation to item 1 of myPageLocation
display dialog "Pasteboard Coordinates:" & return & return & "X: " & item 1 of
myPageLocation & return & "Y: " & item 2 of myPageLocation
--Get ruler coordinate {72, 72} in parent coordinates.
set myPageLocation to resolve myRectangle location {{72, 72}, top right anchor} in
parent coordinates with considering ruler units
--resolve returns a list containing a single item.
set myPageLocation to item 1 of myPageLocation
display dialog "Parent Coordinates:" & return & return & "X: " & item 1 of
myPageLocation & return & "Y: " & item 2 of myPageLocation
```

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. This is particularly useful for AppleScript, which lacks the basic trigonometric functions (sine, cosine) required for most transformations. The ChangeCoordinates sample script shows how to draw a series of regular polygons using this approach:

```
--General purpose routine for drawing regular polygons from their center point.
on myDrawPolygon(myParent, myCenterPoint, myNumberOfPoints, myRadius, myStarPolygon,
myStarInset)
    local myPathPoints, myTransformedPoint
    tell application "Adobe InDesign CS5"
        set myPathPoints to {}
        set myPoint to {0, 0}
        if myStarPolygon is true then
            set myNumberOfPoints to myNumberOfPoints * 2
        end if
        set myInnerRadius to myRadius * myStarInset
        set myAngle to 360 / myNumberOfPoints
        set myRotateMatrix to make transformation matrix with properties
        {counterclockwise rotation angle:myAngle}
        set myOuterTranslateMatrix to make transformation matrix with properties
        {horizontal translation:myRadius}
        set myInnerTranslateMatrix to make transformation matrix with properties
        {horizontal translation:myInnerRadius}
        repeat with myPointCounter from 0 to myNumberOfPoints - 1
            --Translate the point to the inner/outer radius.
            if myStarInset = 1 or my myIsEven(myPointCounter) is true then
                set myTransformedPoint to change coordinates
                myOuterTranslateMatrix point myPoint
            else
                set myTransformedPoint to change coordinates
                myInnerTranslateMatrix point myPoint
            end if
            --Rotate the point.
            set myTransformedPoint to change coordinates
            myRotateMatrix point myTransformedPoint
            copy myTransformedPoint to the end of myPathPoints
            set myRotateMatrix to rotate matrix myRotateMatrix by angle myAngle
        end repeat
        --Create a new polygon.
        tell myParent
            set myPolygon to make polygon
        end tell
        --Set the entire path of the polygon to the array we've created.
        set entire path of path 1 of myPolygon to myPathPoints
        --If the center point is somewhere other than [0,0],
        --translate the polygon to the center point.
        if item 1 of myCenterPoint is not equal to 0 or item 2 of
    myCenterPoint is not equal to 0 then
            set myTransformationMatrix to make transformation matrix with properties
            {horizontal translation:item 1 of myCenterPoint,
            vertical translation:item 2 of myCenterPoint}
            transform myPolygon in pasteboard coordinates from {myCenterPoint,
```

```
            center anchor} with matrix myTransformationMatrix with considering
        ruler units
        end if
    end tell
end myDrawPolygon
--This function returns true if myNumber is even, false if it is not.
on myIsEven(myNumber)
    set myResult to myNumber mod 2
    if myResult = 0 then
        set myResult to true
    else
        set myResult to false
    end if
    return myResult
end myIsEven
```

You also can use the `change coordinates` 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:

- `transform again`
- `transform again individually`
- `transform sequence again`
- `transform sequence again individually`

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

```
set myTransformationMatrix to make transformation matrix with properties
{counterclockwise rotation angle:45}
transform myRectangleA in pasteboard coordinates from center anchor with matrix
myTransformationMatrix
set myRectangleB to duplicate myRectangleA
transform myRectangleB in pasteboard coordinates from {{0, 0}, top left anchor} with
matrix myTransformationMatrix with considering ruler units
set myRectangleC to duplicate myRectangleB
set myResult to transform again myRectangleC
set myRectangleD to duplicate myRectangleC
set myResult to transform again myRectangleD
set myRectangleE to duplicate myRectangleD
set myResult to transform again myRectangleE
set myRectangleF to duplicate myRectangleE
set myResult to transform again myRectangleF
set myRectangleG to duplicate myRectangleF
set myResult to transform again myRectangleG
set myRectangleH to duplicate myRectangleG
set myResult to transform again myRectangleH
transform myRectangleB in pasteboard coordinates from center anchor with matrix
myTransformationMatrix
set myResult to transform again myRectangleD
set myResult to transform again myRectangleF
set myResult to transform again myRectangleH
```

Resize and Reframe

In addition to scaling page items using the transform command, you can also change the size of the shape using two other commands: resize and reframe. These commands 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 command. For the complete script, see Resize.

```
--Given a reference to a rectangle "myRectangle"...
set myDuplicate to duplicate myRectangle
resize myDuplicate in inner coordinates from center anchor by multiplying current
dimensions by values{2, 2}
```

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

```
--Given a reference to a rectangle "myRectangle"...
set myBounds to geometric bounds of myRectangle
set myX1 to item 2 of myBounds - 72
set myY1 to item 1 of myBounds - 72
set myX2 to item 4 of myBounds + 72
set myY2 to item 3 of myBounds + 72
set myDuplicate to duplicate myRectangle
myDuplicate = myRectangle.duplicate();
reframe myDuplicate in inner coordinates opposing corners {{myY1, myX1},{myY2, myX2}}
```

6 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 that you have already read *Adobe InDesign CS5 Scripting Tutorial* and know how to create, install, and run a script. We also assume that you have some knowledge of working with text in InDesign and understand basic typesetting terms.

Entering and Importing Text

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

Creating a text frame

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

```
--Given a document "myDocument"...
set myDocument to document 1
set myPage to page 1 of myDocument
tell myPage
    set myTextFrame to make text frame
    --Set the bounds of the text frame.
    set geometric bounds of myTextFrame to {72, 72, 288, 288}
    --Enter text in the text frame.
    set contents of myTextFrame to "This is some example text."
    --Note that you could also use a properties record
    --to set the bounds and contents of the text in a
    --single line:
    --set myTextFrame to make text frame with properties{geometric bounds:{72, 72, 288,
288}, contents:"This is some example text."}
end tell
```

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

```
--Given a document "myDocument"
set myPage to page 1 of myDocument
tell myPage
    set myTextFrame to make text frame
    --Set the bounds of the text frame.
    set geometric bounds of myTextFrame to my myGetBounds(myDocument, myPage)
end tell
```

The following script fragment shows the myGetBounds handler.

```
on myGetBounds(myDocument, myPage)
    tell application "Adobe InDesign CS5"
        tell document preferences of myDocument
            set myPageWidth to page width
            set myPageHeight to page height
        end tell
        tell margin preferences of myPage
            if side of myPage is left hand then
                set myX2 to left
                set myX1 to right
            else
                set myX1 to left
                set myX2 to right
            end if
            set myY1 to top
            set myY2 to bottom
        end tell
        set myX2 to myPageWidth - myX2
        set myY2 to myPageHeight - myY2
        return {myY1, myX1, myY2, myX2}
    end tell
end myGetBounds
```

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):

```
--Given a document "myDocument" with a text frame on page 1
set myTextFrame to text frame 1 of page 1 of myDocument
--Add text to the end of the text frame. To do this,
--We'll use the last insertion point in the story.
set myNewText to "This is a new paragraph of example text."
tell parent story of myTextFrame
    set contents of insertion point -1 to return & myNewText
end tell
```

Stories and text frames

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

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

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

```
--Given a document "myDocument" with a text frame on page 1...
set myTextFrame to text frame 1 of myDocument
--Now add text beyond the end of the text frame.
set myString to return & "This is some overset text"
tell insertion point -1 of myTextFrame to set contents to myString
set myString to contents of text 1 of myTextFrame
display dialog ("The last paragraph in this dialog should be \"This is some overset
text\". Is it?" & return & myString)
set myString to contents of parent story of myTextFrame
display dialog ("The last paragraph in this alert should be \"This is some overset
text\". Is it?" & return & myString)
```

For more on understanding the relationships between text objects in an InDesign document, see "Understanding Text Objects" on page 88.

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):

```
--Given a document "myDocument" with a text frame on page 1...
set myTextFrame to text frame 1 of page 1 of myDocument
--Replace the third word with the phrase "a little bit of".
tell word 3 of parent story of myTextFrame
    set contents to "a little bit of"
end tell
```

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

```
--Given a document "myDocument" with a text frame on page 1...
set myTextFrame to text frame 1 of page 1 of myDocument
--Replace the text in the second paragraph without
--replacing the return character at the end of the paragraph
--(character -2 is the character before the return).
tell parent story of myTextFrame
    set myText to object reference of text from character 1 to character -2 of paragraph
2
    set contents of myText to "This text replaces the text in paragraph 2."
end tell
    end tell
```

In thepreceding script , we excluded the return character because deleting the return might change the paragraph style applied to the paragraph. To do this, we supplied a character range—from the first character of the paragraph to the last character before the return character that ends the paragraph.

Inserting special characters

Because the Script Editor supports Unicode, you can simply enter Unicode characters in text strings that 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 78" or in the SpecialCharacters tutorial script.)

```
--Given a document "myDocument" containing a story...
set myStory to story 1 of myDocument
--Entering special characters directly:
set contents of myStory to "Registered trademark: ?" & return & "Copyright:  ©" & return
& "Trademark: Ÿ" & return
--Entering special characters by their Unicode glyph ID value:
set contents of insertion point -1 of myStory to "Not equal to: <2260>" & return
set contents of insertion point -1 of myStory to "Square root: <221A>" & return
set contents of insertion point -1 of myStory to "Square root: <00B6>" & return
--Entering special characters by their enumerations:
set contents of insertion point -1 of myStory to "Automatic page number marker: "
set contents of insertion point -1 of myStory to auto page number
set contents of insertion point -1 of myStory to return & "Section symbol: "
set contents of insertion point -1 of myStory to section symbol
set contents of insertion point -1 of myStory to return & "En dash: "
set contents of insertion point -1 of myStory to En dash
set contents of insertion point -1 of myStory to return
```

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):

```
--Given a document "myDocument"
set myPage to page 1 of myDocument
--Get the top and left margins to use as a place point.
tell margin preferences of myPage
    set myX to left
    set myY to right
end tell
--Autoflow a text file on the current page.
--Parameters for the place command of a page:
--file as file or string
--[place point as list {x, y}
--[destination layer as layer object or string]
--[showing options as Boolean (default is false)]
--[autoflowing as Boolean (default is false)]
--You'll have to fill in a valid  file path on your system.
tell myPage
    --Note that if the PlacePoint parameter is inside a column, only the vertical (y)
    --coordinate will be honored--the text frame will expand horizontally to fit the
column.
    set myStory to place alias "Macintosh HD:scripting:test.txt" place point {myX, myY}
autoflowing yes without showing options
end tell
```

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 78," or see the PlaceTextFileInFrame tutorial script.)

```
--Given a document "myDocument" with a text frame on page 1...
set myTextFrame to text frame 1 of page 1 of myDocument
tell insertion point -1 of myTextFrame
    --You'll need to fill in a valid file path for your system.
    place "yukino:test.txt" without showing options
end tell
```

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 78," or see the InsertTextFile tutorial script.)

```
--Given a document "myDocument" with a text frame on page 1...
--Place a text file at the end of the text.
set myTextFrame to text frame 1 of page 1 of myDocument
tell insertion point -1 of parent story of myTextFrame
    --You'll need to fill in a valid file path for your system.
    place "yukino:test.txt" without showing options
end tell
```

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.

```
tell text import preferences
    --Options for character set:
    set character set to UTF8
    set convert spaces into tabs to true
    set spaces into tabs count to 3
    --The dictionary property can take many values, such as French, Italian.
    set dictionary to "English: USA"
    --platform options:
    --macintosh
    --pc
    set platform to macintosh
    set strip returns between lines to true
    set strip returns between paragraphs to true
    set use typographers quotes to true
end tell
```

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

```
tell tagged text import preferences
    set remove text formatting to false
    --Options for style conflict are:
    --publication definition
    --tag file definition
    set style conflict to publication definition
    set use typographers quotes to true
end tell
```

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

```
tell word RTF import preferences
    --convert page breaks property can be:
    --column break
    --none
    --page break
    set convert page breaks to none
    --convert tables to property can be:
    --unformatted tabbed text
    --unformatted table
    set convert tables to to Unformatted Table
    set import endnotes to true
    set import footnotes to true
    set import index to true
    set import TOC to true
    set import unused styles to false
    set preserve graphics to false
    set preserve local overrides to false
    set preserve track changes to false
    set remove formatting to false
    --resolve character style clash and resolve paragraph style clash properties can be:
    --resolve clash auto rename
    --resolve clash use existing
    --resolve clash use new
    set resolve character style clash to resolve clash use existing
    set resolve paragraph style clash to resolve clash use existing
    set use typographers quotes to true
end tell
```

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

```
tell application "Adobe InDesign CS5"
    tell excel import preferences
        --alignment Style property can be:
        --center align
        --left align
        --right align
        --spreadsheet
        set alignment style to spreadsheet
        set decimal places to 4
        set preserve graphics to false
        --Enter the range you want to import as "start cell:end cell".
        set range name to "A1:B16"
        set sheet index to 1
        set sheet name to "pathpoints"
        set show hidden cells to false
        --table formatting property can be:
        --excel formatted table
        --excel unformatted tabbed text
        --excel unformatted table
        set table formatting to excel formatted table
        set use typographers quotes to true
        set view name to ""
    end tell
end tell
```

Exporting Text and Setting Text-Export Preferences

The following script shows how to export text from an InDesign document. Note that you must use text or story objects to export into text file formats; you cannot export all text in a document in one operation. (We omitted the `myGetBounds` function from this listing; you can find it in "Creating a text frame" on page 78," or see the ExportTextFile tutorial script.)

```
tell application "Adobe InDesign CS5"
    set myDocument to active document
    set myPage to page 1 of myDocument
    tell myPage
        set myTextFrame to make text frame with properties {geometric bounds:
        my myGetBounds(myDocument, myPage), contents:placeholder text}
    end tell
    --Text export method parameters:
    --format as enumeration
    --to alias as string
    --showing options boolean
    --version comments string
    --force save boolean
    --Format parameter can be:
    --InCopy CS Document
    --InCopy Document
    --rtf
    --tagged text
    --text type
    --Export the story as text. You'll have to fill in a valid file path on your system.
    tell parent story of myTextFrame
        export to "yukino:test.txt" format text type
    end tell
end tell
```

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 78," or see the ExportTextRange tutorial script.)

```
--Given a document with a text frame on page 1...
set myTextFrame to text frame 1 of page 1 of active document
set myStory to parent story of myTextFrame
set myStartCharacter to index of character 1 of paragraph 1 of myStory
set myEndCharacter to the index of last character of paragraph 1 of myStory
set myText to object reference of text from character myStartCharacter to character
myEndCharacter of myStory
--Export the text range. You'll have to fill in a valid file path on your system.
tell myText to export to "Macintosh HD:scripting:test.txt" format text type
```

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):

```
tell text export preferences
    --Options for character set:
    --UTF8
    --UTF16
    --platform
    set character set to UTF8
    --platform options:
    --macintosh
    --pc
    set platform to macintosh
end tell
```

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

```
tell tagged text export preferences
    --Options for character set:
    --ansi
    --ascii
    --gb18030
    --ksc5601
    --shiftJIS
    --unicode
    set character set to unicode
    --tag form options:
    --abbreviated
    --verbose
    set tag form to verbose
end tell
```

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

```
tell application "Adobe InDesign CS5"
    if (count documents) > 0 then
        set myDocument to active document
        if (count stories of myDocument) > 0 then
            my myExportAllText(name of myDocument)
        end if
    end if
end tell
```

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

```
on myExportAllText(myDocumentName)
    tell application "Adobe InDesign CS5"
        --File name for the exported text. Fill in a valid file path on your system.
        set myFileName to "Adobe:test.txt"
        --If you want to add a separator line between stories,
        --set myAddSeparator to true.
        set myAddSeparator to true
        set myNewDocument to make document
        set myDocument to document myDocumentName
        tell page 1 of myNewDocument
            set myTextFrame to make text frame with properties
            {geometric bounds:my myGetBounds(myNewDocument, page 1 of myNewDocument)}
            set myNewStory to parent story of myTextFrame
            repeat with myCounter from 1 to (count stories in myDocument)
                set myStory to story myCounter of myDocument
                --Duplicate the text of the story to the end of the temporary story.
                tell text 1 of myStory
                    duplicate to after insertion point -1 of story 1 of myNewDocument
                end tell
                --If the imported text did not end with a return, enter a return
                --to keep the stories from running together.
                if myCounter is not equal to (count stories of myDocument) then
                    if contents of character -1 of myNewStory is not return then
                        set contents of insertion point -1 of myNewStory to return
                        if myAddSeparator is true then
                            set contents of insertion point -1 of myNewStory to
                            "----------------------------------------" & return
                        end if
                    end if
                end if
            end repeat
            tell myNewStory
                export to myFileName format text type
            end tell
            close myNewDocument saving no
        end tell
    end tell
end myExportAllText
```

Do not assume that you are limited to exporting text using existing export filters. Because AppleScript 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 78," or see the ExportHTML tutorial script.)

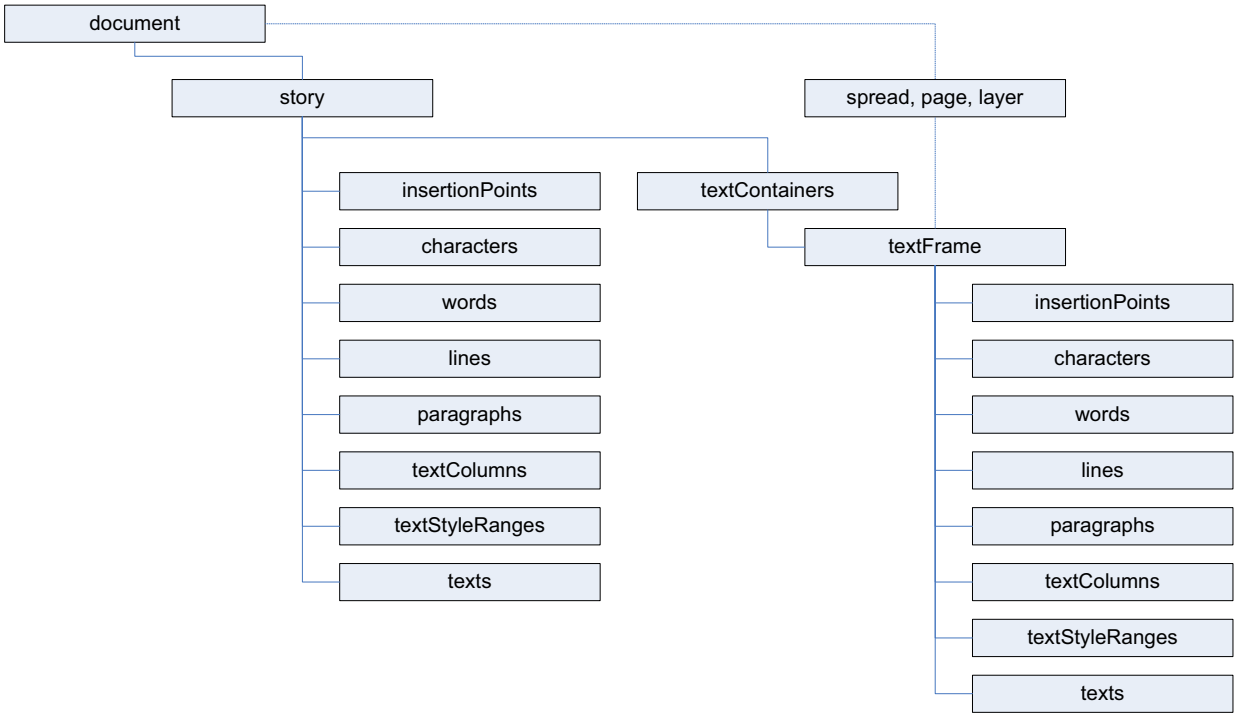
```
tell application "Adobe InDesign CS5"
    --Use the myStyleToTagMapping array to set up your paragraph style to tag mapping.
    set myStyleToTagMapping to {}
    --For each style to tag mapping, add a new item to the array.
    copy {"body_text", "p"} to end of myStyleToTagMapping
    copy {"heading1", "h1"} to end of myStyleToTagMapping
    copy {"heading2", "h3"} to end of myStyleToTagMapping
    copy {"heading3", "h3"} to end of myStyleToTagMapping
    --End of style to tag mapping.
    if (count documents) is not equal to 0 then
        set myDocument to document 1
        if (count stories of myDocument) is not equal to 0 then
            --Open a new text file.
            set myTextFile to choose file name with prompt "Save HTML As"
            --Iterate through the stories.
```

```
repeat with myCounter from 1 to (count stories of myDocument)
    set myStory to story myCounter of myDocument
    repeat with myParagraphCounter from 1 to (count paragraphs of myStory)
        set myParagraph to object reference of paragraph myParagraphCounter
        of myStory
        if (count tables of myParagraph) is 0 then
            --If the paragraph is a simple paragraph--no tables,
            --no local formatting--then simply export the text of the
            --pararaph with the appropriate tag.
            if (count text style ranges of myParagraph) is 1 then
                set myTag to my myFindTag(name of applied paragraph style
                of myParagraph, myStyleToTagMapping)
                --If the tag comes back empty, map it to the
                --basic paragraph tag.
                if myTag = "" then
                    set myTag to "p"
                end if
                set myStartTag to "<" & myTag & ">"
                set myEndTag to "</" & myTag & ">"
                --If the paragraph is not the last paragraph in the story,
                --omit the return character.
                if the contents of character -1 of myParagraph is return then
                    set myText to object reference of text from character 1
                    to character -2 of myParagraph
                    set myString to contents of myText
                else
                    set myString to contents of myParagraph
                end if
                --Write the text of the paragraph to the text file.
                if myParagraphCounter = 1 then
                    set myAppendData to false
                else
                    set myAppendData to true
                end if
                my myWriteToFile(myStartTag & myString & myEndTag & return,
                myTextFile, myAppendData)
            else
                --Handle text style range export by iterating through
                --the text style ranges in the paragraph.
                set myTextStyleRanges to text style ranges of myParagraph
                repeat with myRangeCounter from 1 to (count text style ranges
                of myParagraph)
                    set myTextStyleRange to object reference of text style
                    range myRangeCounter of myParagraph
                    if character -1 of myTextStyleRange is return then
                        set myStartCharacter to index of character 1 of
                        myTextStyleRange
                        set myEndCharacter to index of character -2 of
                        myTextStyleRange
                        set myText to object reference of text from character
                        myStartCharacter to character myEndCharacter of myStory
                        set myString to contents of myText
                    else
                        set myString to contents of myTextStyleRange
                    end if
                    if font style of myTextStyleRange is "Bold" then
                        set myString to "<b>" & myString & "</b>"
                    else if font style of myTextStyleRange is "Italic" then
                        set myString to "<i>" & myString & "</i>"
                    end if
```

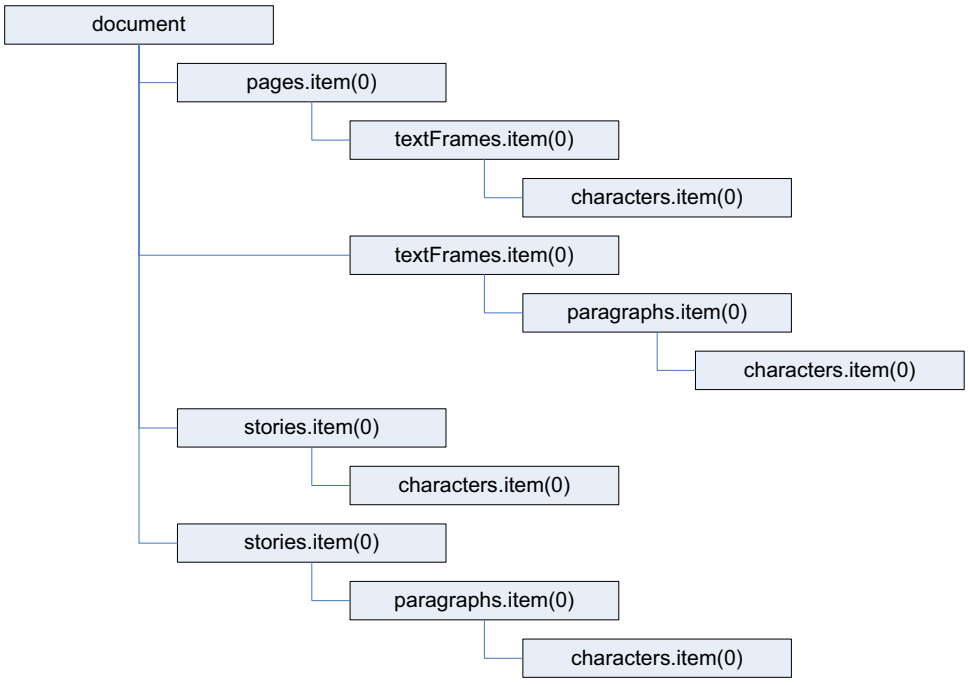
```
                        my myWriteToFile(myString, myTextFile, true)
                    end repeat
                    my myWriteToFile(return, myTextFile, true)
                end if
            else
                --Handle table export (assumes that there is only
                --one table per paragraph,
                --and that the table is in the paragraph by itself).
                set myTable to table 1 of myParagraph
                my myWriteToFile("<table border = 1>", myTextFile, true)
                repeat with myRowCounter from 1 to (count rows of myTable)
                    my myWriteToFile("<tr>", myTextFile, true)
                    repeat with myColumnCounter from 1 to
                    (count columns of myTable)
                        if myRowCounter = 1 then
                            set myString to "<th>"
                            set myString to myString & text 1 of cell myColumnCounter
                            of row myRowCounter of myTable
                            set myString to myString & "</th>"
                        else
                            set myString to "<td>"
                            set myString to myString & text 1 of cell myColumnCounter
                            of row myRowCounter of myTable
                            set myString to myString & "</td>"
                        end if
                        my myWriteToFile(myString, myTextFile, true)
                    end repeat
                    my myWriteToFile("</tr>" & return, myTextFile, true)
                end repeat
                my myWriteToFile("</table>" & return, myTextFile, true)
            end if
        end repeat
    end repeat
  end if
 end if
end tell
```

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 `parent text frames` property.

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

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

Working with text selections

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

```
if (count documents) is not equal to 0 then
    --If the selection contains more than one item, the selection
    --is not text selected with the Type tool.
    set mySelection to selection
    if (count mySelection) is not equal to 0 then
        --Evaluate the selection based on its type.
        set myTextClasses to {insertion point, word, text style range,
        line, paragraph, text column, text, story}
        if class of item 1 of selection is in myTextClasses then
            --The object is a text object; display the text object type.
            --A practical script would do something with the selection,
            --or pass the selection on to a function.
            display dialog ("Selection is a text object.")
            --If the selection is inside a note, the parent of the selection
            --will be a note object.
            if class of parent of item 1 of selection is note then
                display dialog ("Selection is inside a note.")
            end if
        else if class of item 1 of selection is text frame then
            display dialog ("Selection is a text frame")
        else
            display dialog ("Selected item is not a text object.")
        end if
    else
        display dialog ("Please select some text and try again.")
    end if
else
    display dialog ("No documents are open.")
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 (whose arguments are identical to the `move` method). The following script fragment shows how it works (for the complete script, see MoveText):

```
--Given a document myDocument" with four text frams on page 1...
set myTextFrameA to text frame 4 of page 1 of myDocument
set myTextFrameB to text frame 3 of page 1 of myDocument
set myTextFrameC to text frame 2 of page 1 of myDocument
set myTextFrameD to text frame 1 of page 1 of myDocument
--Note that moving text removes it from its original location.
tell parent story of myTextFrameD
    --Move WordC between the words in TextFrameC.
    move paragraph 4 to before word -1 of paragraph 1 of parent story of myTextFrameC
    --Move WordB after the word in TextFrameB.
    move paragraph 3 to after insertion point -1 of myTextFrameB
    --Move WordA to before the word in TextFrameA.
    move paragraph 2 to before word 1 in myTextFrameA
end tell
```

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 78," or see the MoveTextBetweenDocuments tutorial script.)

```
--Given a document "mySourceDocument"...
set mySourceStory to story 1 of mySourceDocument
--Create a new document to move the text to.
set myTargetDocument to make document
set myPage to page 1 of myTargetDocument
tell myPage
    set myTextFrame to make text frame with properties
    {geometric bounds:my myGetBounds(myTargetDocument, myPage)}
end tell
set myTargetStory to story 1 of myTargetDocument
set contents of myTargetStory to "This is the target text. Insert the source text after
this paragraph." & return
duplicate paragraph 1 of mySourceStory to after insertion point -1 of myTargetStory
```

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 78," or see the CopyPasteText tutorial script.)

```
--Given an open document with a text frame on page 1
set myDocumentA to document 1
set myTextFrameA to text frame 1 of page 1 of myDocumentA
--Create another example document.
set myDocumentB to make document
set myPageB to page 1 of myDocumentB
tell myPageB
    set myTextFrameB to make text frame with properties {geometric bounds:my
myGetBounds(myDocumentB, myPageB)}
end tell
--Make document A the active document.
set active document to myDocumentA
--Select the text.
select text 1 of parent story of myTextFrameA
copy
--Make document B the active document.
set active document to myDocumentB
--Select the insertion point at which you want to paste the text.
select insertion point -1 of myTextFrameB
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):

```
tell application "Adobe InDesign CS5"
    set myDocument to document 1
    set myPage to page 1 of myDocument
    tell myPage
        set myTextFrameA to make text frame with properties
        {geometric bounds:{72, 72, 144, 288}}
        set contents of myTextFrameA to "This is a formatted string."
        set font style of text 1 of parent story of myTextFrameA to "Bold"
        set myTextFrameB to make text frame with properties
        {geometric bounds:{228, 72, 300, 288}}
        set contents of myTextFrameB to "This is the destination text frame.
        Text pasted here will retain its formatting."
        set font style of text 1 of myTextFrameB to "Italic"
    end tell
    --Copy from one frame to another using a simple copy.
    select text 1 of myTextFrameA
    copy
    select insertion point -1 of myTextFrameB
    paste
    --Create another text frame on the active page.
    tell myPage
        set myTextFrameC to make text frame with properties
        {geometric bounds:{312, 72, 444, 288}}
    end tell
    set contents of myTextFrameC to "Text copied here will take on
    the formatting of the existing text."
    set font style of text 1 of parent story of myTextFrameC to "Italic"
    --Copy the unformatted string from text frame A to the end of text frame C (note
    --that this doesn't really copy the text it replicates the text string from one
    --text frame in another text frame):
    set contents of insertion point -1 of myTextFrameC to contents of text 1 of parent
story of myTextFrameA
end tell
```

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 78," or see the TextIterationWrong tutorial script.)

```
--The following for loop will fail to format all of the paragraphs
--and then generate an error.
repeat with myParagraphCounter from 1 to (count paragraphs of myStory)
    if contents of word 1 of paragraph myParagraphCounter of myStory is "Delete" then
        tell paragraph myParagraphCounter of myStory to delete
    else
        set point size of paragraph myParagraphCounter of myStory to 24
    end if
end repeat
```

In the preceding example, some of the paragraphs are left unformatted. How does this happen? The loop in the script iterates through the paragraphs from the first paragraph in the story to the last. As it does so, it deletes paragraphs that begin with the word "Delete." When the script deletes the second paragraph, the third paragraph moves up to take its place. When the loop counter reaches 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 78," or see the TextIterationRight tutorial script.)

```
--By iterating backwards we can avoid the error.
repeat with myParagraphCounter from (count paragraphs of myStory) to 1 by -1
    if contents of word 1 of paragraph myParagraphCounter of myStory is "Delete" then
        tell paragraph myParagraphCounter of myStory to delete
    else
        set point size of paragraph myParagraphCounter of myStory to 24
    end if
end repeat
```

Working with Text Frames

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

Linking text frames

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

```
--Given a document "myDocument" with two unlinked text frames on page 1...
set myTextFrameA to text frame 2 of page 1 of document 1
set myTextFrameB to text frame 1 of page 1 of document 1
--Link TextFrameA to TextFrameB using the next text frame property.
set next text frame of myTextFrameA to myTextFrameB
--Add a page.
tell myDocument
    set myNewPage to make page
end tell
--Create another text frame on the new page.
tell myNewPage
    set myTextFrameC to make text frame with properties
    {geometric bounds:{72, 72, 144, 144}}
    --Link TextFrameC to TextFrameB using the previousTextFrame property.
    set previous text frame of myTextFrameC to myTextFrameB
    --Fill the text frames with placeholder text.
    set contents of myTextFrameA to placeholder text
end tell
```

Unlinking text frames

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

```
--Unlink the two text frames.
set next text frame of myTextFrameA to 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):

```
set myObjectList to {}
--Script does nothing if no documents are open or if no objects are selected.
tell application "Adobe InDesign CS5"
    if (count documents) is not equal to 0 then
        set mySelection to selection
        if (count mySelection) is not equal to 0 then
            --Process the objects in the selection to create a list of
            --qualifying objects (text frames).
            repeat with myCounter from 1 to (count mySelection)
                if class of item myCounter of mySelection is text frame then
                    set myObjectList to myObjectList & item myCounter of mySelection
                else if class of item myCounter of mySelection is in {text,
                insertion point, character, word, line, text style range, paragraph,
                text column} then
                    set myObject to item 1 of parent text frames of item myCounter
                    of mySelection
                    set myObjectList to myObjectList & myObject
                end if
            end repeat
            --If the object list is not empty, pass it on to the handler
            --that does the real work.
            if (count myObjectList) is not equal to 0 then
                my myBreakFrames(myObjectList)
            end if
        end if
    end if
end tell
```

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

```
on myBreakFrames(myObjectList)
    repeat with myCounter from 1 to (count myObjectList)
        my myBreakFrame(item myCounter of myObjectList)
    end repeat
end myBreakFrames
on myBreakFrame(myTextFrame)
    tell application "Adobe InDesign CS5"
        if next text frame of myTextFrame is not equal to nothing and previous text frame
of myTextFrame is not equal to nothing then
            set myNewFrame to duplicate myTextFrame
            if contents of myTextFrame is not equal to "" then
                tell text 1 of myTextFrame to delete
            end if
            delete myTextFrame
        end if
    end tell
end myBreakFrame
```

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):

```
on mySplitStory(myStory)
    tell application "Adobe InDesign CS5"
        tell document 1
            local myTextContainers
            set myTextContainers to text containers in myStory
            if (count myTextContainers) is greater than 1 then
                repeat with myCounter from (count myTextContainers) to 1 by -1
                    set myTextFrame to item myCounter of myTextContainers
                    tell myTextFrame
                        duplicate
                        if text 1 of myTextFrame is not equal to "" then
                            tell text 1 of myTextFrame
                                delete
                            end tell
                        end if
                        delete
                    end tell
                end repeat
            end if
        end tell
    end tell
end mySplitStory
```

Creating an anchored frame

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

```
--Given a document "myDocument" with a text frame on page 1...
set myPage to page 1 of myDocument
set myTextFrame to text frame 1 of myPage
tell insertion point 1 of paragraph 1 of myTextFrame
    set myInlineFrame to make text frame
end tell
--Recompose the text to make sure that getting the
--geometric bounds of the inline graphic will work.
tell text 1 of myTextFrame to recompose
--Get the geometric bounds of the inline frame.
set myBounds to geometric bounds of myInlineFrame
--Set the width and height of the inline frame. In this example, we'll
--make the frame 24 points tall by 72 points wide.
set e1 to item 1 of myBounds
set e2 to item 2 of myBounds
set e3 to (item 1 of myBounds) + 24
set e4 to (item 2 of myBounds) + 72
set geometric bounds of myInlineFrame to {e1, e2, e3, e4}
set contents of myInlineFrame to "This is an inline frame."
tell insertion point 1 of paragraph 2 of myTextFrame
    set myAnchoredFrame to make text frame
end tell
--Recompose the text to make sure that getting the
--geometric bounds of the inline graphic will work.
tell text 1 of myTextFrame to recompose
--Get the geometric bounds of the inline frame.
set myBounds to geometric bounds of myAnchoredFrame
--Set the width and height of the inline frame. In this example, we'll
--make the frame 24 points tall by 72 points wide.
set e1 to item 1 of myBounds
```

```
set e2 to item 2 of myBounds
set e3 to (item 1 of myBounds) + 24
set e4 to (item 2 of myBounds) + 72
set geometric bounds of myAnchoredFrame to {e1, e2, e3, e4}
set contents of myAnchoredFrame to "This is an anchored frame."
tell anchored object settings of myAnchoredFrame
    set anchored position to anchored
    set anchor point to top left anchor
    set horizontal reference point to anchor location
    set horizontal alignment to left align
    set anchor xoffset to 72
    set vertical reference point to line baseline
    set anchor yoffset to 24
    set anchor space above to 24
end tell
```

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.)

```
set horizontal measurement units of view preferences to points
set vertical measurement units of view preferences to points
--To set the text formatting defaults for a document, replace "app"
--in the following lines with a reference to a document.
tell text defaults
    set alignToBaseline to true
    --Because the font might not be available, it's usually best
    --to apply the font within a try...catch structure. Fill in the
    --name of a font on your system.
    try
        set appliedFont to font "Minion Pro"
    end try
    --Because the font style might not be available, it's usually best
    --to apply the font style within a try...catch structure.
    try
        set font style to "Regular"
    end try
    --Because the language might not be available, it's usually best
    --to apply the language within a try...catch structure.
    try
        set applied language to "English: USA"
    end try
    set autoLeading to 100
    set balsanceRaggedLines to false
    set baselineShift to 0
    set capitalization to normal
    set composer to "Adobe Paragraph Composer"
    set desiredGlyphScaling to 100
    set desiredLetterSpacing to 0
```

```
set desiredWordSpacing to 100
set drop cap characters to 0
if drop cap characters is not equal to 0 then
    dropCapLines to 3
    --Assumes that the application has a default character style named "myDropCap"
    set drop cap style to character style "myDropCap"
end if
set fill color to "Black"
set fill tint to 100
set first line indent to 14
set grid align first line only to false
set horizontal scale to 100
set hyphenate after first to 3
set hyphenate before last to 4
set hyphenate capitalized words to false
set hyphenate ladder limit to 1
set hyphenate words longer than to 5
set hyphenation to true
set hyphenation zone to 36
set hyphen weight to 9
set justification to left align
set keep all lines together to false
set keep lines together to true
set keep first lines to 2
set keep last lines to 2
set keep with next to 0
set kerning method to "Optical"
set leading to 14
set left indent to 0
set ligatures to true
set maximum glyph scaling to 100
set maximum letter spacing to 0
set maximum word spacing to 160
set minimum glyph scaling to 100
set minimum letter spacing to 0
set minimum word spacing to 80
set no break to false
set OTF contextual alternate to true
set OTF discretionary ligature to false
set OTF figure style to proportional oldstyle
set OTF fraction to true
set OTF historical to false
set OTF ordinal to false
set OTF slashed zero to false
set OTF swash to false
set OTF titling to false
set overprint fill to false
set overprint stroke to false
set pointSize to 11
set position to normal
set right indent to 0
set rule above to false
if rule above = true then
    set rule above color to color "Black"
    set rule above gap color to swatch "None"
    set rule above gap overprint to false
    set rule above gap tint to 100
    set rule above left indent to 0
    set rule above line weight to 0.25
    set rule above offset to 14
```

```
            set rule above overprint to false
            set rule above right indent to 0
            set rule above tint to 100
            set rule above type to stroke style "Solid"
            set rule above width to column width
        end if
        set rule below to false
        if rule below = true then
            set rule below color to color "Black"
            set rule below gap color to swatch "None"
            set rule below gap overprint to false
            set rule below gap tint to 100
            set rule below left indent to 0
            set rule below line weight to 0.25
            set rule below offset to 14
            set rule below overprint to false
            set rule below right indent to 0
            set rule below tint to 100
            set rule below type to stroke style "Solid"
            set rule below width to column width
        end if
        set single word justification to left align
        set skew to 0
        set space after to 0
        set space before to 0
        set start paragraph to anywhere
        set strike thru to false
        if strike thru = true then
            set strike through color to color "Black"
            set strike through gap color to swatch "None"
            set strike through gap overprint to false
            set strike through gap tint to 100
            set strike through offset to 3
            set strike through overprint to false
            set strike through tint to 100
            set strike through type to stroke style "Solid"
            set strike through weight to 0.25
        end if
        set stroke color to "None"
        set stroke tint to 100
        set stroke weight to 0
        set tracking to 0
        set underline to false
        if underline = true then
            set underline color to color "Black"
            set underline gap color to swatch "None"
            set underline gap overprint to false
            set underline gap tint to 100
            set underline offset to 3
            set underline overprint to false
            set underline tint to 100
            set underline type to stroke style "Solid"
            set underline weight to 0.25
        end if
        set vertical scale to 100
    end tell
```

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 myApplicationFonts to the name of every font
set myDocument to active document
tell myDocument
    set myPage to page 1
    tell myPage
        set myTextFrame to make text frame with properties {geometric bounds:my
myGetBounds(myDocument, myPage)}
    end tell
    set myStory to parent story of myTextFrame
    set myDocumentFonts to name of every font
end tell
set myString to "Document Fonts:" & return
repeat with myCounter from 1 to (count myDocumentFonts)
    set myString to myString & (item myCounter) of myDocumentFonts & return
end repeat
set myString to myString & return & "Application Fonts:" & return
repeat with myCounter from 1 to (count myApplicationFonts)
    set myString to myString & (item myCounter) of myApplicationFonts & return
end repeat
set contents of myStory to myString
```

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

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

Applying a font

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

```
--Given a font name "myFontName" and a text object "myText"...
set applied font of myText to myFontName
```

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

```
tell myText
set applied font to "Adobe Caslon Pro"
set font style to "Semibold Italic"
end tell
```

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:

```
--Given a document "myDocument" containing a story...
set myStory to story 1 of myDocument
tell character 1 of myStory
    set align to baseline to false
    set applied character style to character style "[None]" of myDocument
    set applied font to "Minion ProRegular"
    set applied language to "English: USA"
    set applied numbering list to "[Default]"
    set applied paragraph style to paragraph style "[No Paragraph Style]" of myDocument
    set auto leading to 120
    set balance ragged lines to no balancing
    set baseline shift to 0
    set bullets alignment to left align
    set bullets and numbering list type to no list
    set bullets character style to character style "[None]" of myDocument
    set bullets text after to "^t"
    set capitalization to normal
    set composer to "Adobe Paragraph Composer"
    set desired glyph scaling to 100
    set desired letter spacing to 0
    set desired word spacing to 100
    set drop cap characters to 0
    set drop cap lines to 0
    set drop cap style to character style "[None]" of myDocument
    set dropcap detail to 0
    set fill color to color "Black" of myDocument
    set fill tint to -1
    set first line indent to 0
    set font style to "Regular"
    set gradient fill angle to 0
    set gradient fill length to -1
    set gradient fill start to [0, 0]
    set gradient stroke angle to 0
    set gradient stroke length to -1
    set gradient stroke start to [0, 0]
    set grid align first line only to false
    set horizontal scale to 100
    set hyphen weight to 5
    set hyphenate across columns to true
    set hyphenate after first to 2
    set hyphenate before last to 2
    set hyphenate capitalized words to true
    set hyphenate ladder limit to 3
    set hyphenate last word to true
    set hyphenate words longer than to 5
    set hyphenation to true
    set hyphenation zone to 3
    set ignore edge alignment to false
    set justification to left align
    set keep all lines together to false
    set keep first lines to 2
    set keep last lines to 2
    set keep lines together to false
    set keep rule above in frame to false
    set keep with next to 0
    set kerning method to "Optical"
    set last line indent to 0
    set leading to 12
    set left indent to 0
    set ligatures to true
```

```
set maximum glyph scaling to 100
set maximum letter spacing to 0
set maximum word spacing to 133
set minimum glyph scaling to 100
set minimum letter spacing to 0
set minimum word spacing to 80
set no break to false
set numbering alignment to left align
set numbering apply restart policy to true
set numbering character style to character style "[None]" of myDocument
set numbering continue to true
set numbering expression to "^#.^t"
set numbering format to "1, 2, 3, 4..."
set numbering level to 1
set numbering start at to 1
set OTF contextual alternate to true
set OTF discretionary ligature to false
set OTF figure style to proportional lining
set OTF fraction to false
set OTF historical to false
set OTF locale to true
set OTF mark to true
set OTF ordinal to false
set OTF slashed zero to false
set OTF stylistic sets to 0
set OTF swash to false
set OTF titling to false
set overprint fill to false
set overprint stroke to false
set point size to 72
set position to normal
set positional form to none
set right indent to 0
set rule above to false
set rule above color to "Text Color"
set rule above gap color to swatch "None" of myDocument
set rule above gap overprint to false
set rule above gap tint to 100
set rule above left indent to 0
set rule above line weight to 0.25
set rule above offset to 14
set rule above overprint to false
set rule above right indent to 0
set rule above tint to 100
set rule above type to stroke style "Solid" of myDocument
set rule above width to column width
set rule below to false
set rule below color to "Text Color"
set rule below gap color to swatch "None" of myDocument
set rule below gap overprint to false
set rule below gap tint to 100
set rule below left indent to 0
set rule below line weight to 0.25
set rule below offset to 14
set rule below overprint to false
set rule below right indent to 0
set rule below tint to 100
set rule below type to stroke style "Solid" of myDocument
set rule below width to column width
set single word justification to left align
```

```
    set skew to 0
    set space after to 0
    set space before to 0
    set start paragraph to anywhere
    set strike thru to false
    set strike through color to color "Black" of myDocument
    set strike through gap color to swatch "None" of myDocument
    set strike through gap overprint to false
    set strike through gap tint to 100
    set strike through offset to 3
    set strike through overprint to false
    set strike through tint to 100
    set strike through type to stroke style "Solid" of myDocument
    set strike through weight to 0.25
    set stroke color to swatch "None" of myDocument
    set stroke tint to -1
    set stroke weight to 1
    set tracking to 0
    set tracking to 0
    set underline to false
    set underline color to color "Black" of myDocument
    set underline gap color to swatch "None" of myDocument
    set underline gap overprint to false
    set underline gap tint to 100
    set underline offset to 3
    set underline overprint to false
    set underline tint to 100
    set underline type to stroke style "Solid" of myDocument
    set underline weight to 0.25
    set vertical scale to 100
end tell
```

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 myStory to story 1 of myDocument
set myColorA to color "DGC1_664a" of myDocument
set myColorB to color "DGC1_664b" of myDocument
tell paragraph 1 of myStory
    set point size to 72
    set justification to center align
    --Apply a color to the fill of the text.
    set fill color to myColorA
    set stroke color to myColorB
end tell
tell paragraph 2 of myStory
    set stroke weight to 3
    set point size to 144
    set justification to center align
    set fill color to myColorB
    set stroke color to myColorA
    set stroke weight to 3
end tell
```

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):

```
--Given a document "myDocument" containing a story and the color "Red"...
tell myDocument
    set myColor to color "Red"
    set myStory to story 1 of myDocument
    set contents of myStory to "Normal text. Text with a character style applied to it.
    More normal text."
    --Create a character style named "myCharacterStyle" if
    --no style by that name already exists.
    try
        set myCharacterStyle to character style "myCharacterStyle"
    on error
        --The style did not exist, so create it.
        set myCharacterStyle to make character style with properties
        {name:"myCharacterStyle"}
    end try
    --At this point, the variable myCharacterStyle contains a reference to a character
    --style object, which you can now use to specify formatting.
    set fill color of myCharacterStyle to myColor
    --Create a paragraph style named "myParagraphStyle" if
    --no style by that name already exists.
    try
        set myParagraphStyle to paragraph style "myParagraphStyle"
    on error
        --The paragraph style did not exist, so create it.
        set myParagraphStyle to make paragraph style with properties
        {name:"myParagraphStyle"}
    end try
    --At this point, the variable myParagraphStyle contains a reference to a paragraph
    --style object, which you can now use to specify formatting.
    --(Note that the story object does not have the apply paragraph style method.)
    tell text 1 of myStory
        apply paragraph style using myParagraphStyle
        tell text from character 13 to character 54
            apply character style using myCharacterStyle
        end tell
    end tell
end tell
```

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):

```
--Given a document "myDocument"...
--Get the last paragraph style.
set myParagraphStyle to paragraph style -1 of myDocument
--Get the last character style
set myCharacterStyle to character style -1 of myDocument
--At this point, the variable myParagraphStyle contains a reference to a paragraph
--style object. Next, add a nested style to the paragraph style.
tell myParagraphStyle
    set myNestedStyle to make nested style with properties {applied character
style:myCharacterStyle, delimiter:".", inclusive:true, repetition:1}
end tell
set myPage to page 1 of myDocument
set myTextFrame to text frame 1 of myPage
--Apply the paragraph style to the story so that we can see the
--effect of the nested style we created.
--(Note that the story object does not have the apply paragraph style method.)
set myStory to parent story of myTextFrame
tell text 1 of myStory
    apply paragraph style using myParagraphStyle
end tell
```

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):

```
--Given a document "myDocment" with paragraph styles named
--"myParagraphStyleA" and "myParagraphStyleB", remove the
--paragraph style "myParagraphStyleA" and replace with
--"myParagraphStyleB."
tell myDocument
    delete paragraph style "myParagraphStyleA" replacing with paragraph style
"myParagraphStyleB"
end tell
```

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):

```
--You'll have to fill in a valid file path for your system.
set myFilePath to "Macintosh HD:scripting:styles.indd"
--Create a new document.
set myDocument to make document
tell myDocument
    --Import the styles from the saved document.
    --importStyles parameters:
    --Format options for text styles are:
    --    paragraph styles format
    --    character styles format
    --    text styles format
    --From as file or string
    --Global Strategy options are:
    --    do not load the style
    --    load all with overwrite
    --    load all with rename
    import styles format text styles format from myFilePath global strategy load all
with overwrite
end tell
```

Finding and Changing Text

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

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

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

About find/change preferences

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

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

```
--find/change text preferences
tell application "Adobe InDesign CS5"
set find text preferences to nothing
set change text preferences to nothing
end tell
```

```
--find/change grep preferences
tell application "Adobe InDesign CS5"
set find grep preferences to nothing
set change grep preferences to nothing
end tell
```

```
--find/change glyph preferences
tell application "Adobe InDesign CS5"
set find glyph preferences to nothing
set change glpyh preferences to nothing
end tell
```

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

Finding and changing text

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

```
--Clear the find/change preferences.
set find text preferences to nothing
set change text preferences to nothing
--Search the document for the string "Text".
set find what of find text preferences to "text"
--Set the find options.
set case sensitive of find change text options to false
set include footnotes of find change text options to false
set include hidden layers of find change text options to false
set include locked layers for find of find change text options to false
set include locked stories for find of find change text options to false
set include master pages of find change text options to false
set whole word of find change text options to false
tell active document
    set myFoundItems to find text
    display dialog ("Found " & (count myFoundItems) & " instances of the search
string.")
end tell
```

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

```
--Clear the find/change preferences.
set find text preferences to nothing
set change text preferences to nothing
--Search the document for the string "copy".
set find what of find text preferences to "copy"
set change to of change text preferences to "text"
--Set the find options.
set case sensitive of find change text options to false
set include footnotes of find change text options to false
set include hidden layers of find change text options to false
set include locked layers for find of find change text options to false
set include locked stories for find of find change text options to false
set include master pages of find change text options to false
set whole word of find change text options to false
tell active document
    set myFoundItems to change text
    display dialog ("Found " & (count myFoundItems) & " instances of the search
string.")
end tell
```

Finding and changing text formatting

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

```
--Clear the find/change preferences.
set find text preferences to nothing
set change text preferences to nothing
--Set the find options.
set case sensitive of find change text options to false
set include footnotes of find change text options to false
set include hidden layers of find change text options to false
set include locked layers for find of find change text options to false
set include locked stories for find of find change text options to false
set include master pages of find change text options to false
set whole word of find change text options to false
set point size of find text preferences to 24
--The following line will only work if your default font has a font style named "Bold"
--if not, change the text to a font style used by your default font.
set point size of change text preferences to 48
--Search the document. In this example, we'll use the
--InDesign search metacharacter "^9" to find any digit.
tell document 1
    set myFoundItems to change text
end tell
display dialog ("Changed " & (count myFoundItems) & " instances of the search string.")
--Clear the find/change preferences after the search.
set find text preferences to nothing
set change text preferences to nothing
```

Using grep

InDesign supports regular expression find/change through the `findGrep` and `changeGrep` methods. Regular-expression find/change also can find text with a specified format or replace the formatting of the text with formatting specified in the properties of the `changeGrepPreferences` object. The following

script fragment shows how to use these methods and the related preferences objects (for the complete script, see FindGrep):

```
--Clear the find/change preferences.
set find grep preferences to nothing
set change grep preferences to nothing
--Set the find options.
set include footnotes of find change grep options to false
set include hidden layers of find change grep options to false
set include locked layers for find of find change grep options to false
set include locked stories for find of find change grep options to false
set include master pages of find change grep options to false
--Regular expression for finding an email address.
set find what of find grep preferences to "(?i)[A-Z0-9]*?@[A-Z0-9]*?[.]..."
--Apply the change to 24-point text only.
set point size of find grep preferences to 24
set underline of change grep preferences to true
tell myDocument
    change grep
end tell
--Clear the find/change preferences after the search.
set find grep preferences to nothing
set change grep preferences to nothing
```

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

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

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

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

```
set myDocument to document 1
set myStory to story 1 of myDocument
myReadPMTags(myStory)
```

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

```
on myReadPMTags(myStory)
    local myFoundTags, myFoundItems, myFoundTag, myString, myStyleName, myStyle
    tell application "Adobe InDesign CS5"
        set myDocument to parent of myStory
        --Reset the find grep preferences to ensure that
        --previous settings do not affect the search.
        set find grep preferences to nothing
        set change grep preferences to nothing
        --Set the find options.
        set include footnotes of find change grep options to false
        set include hidden layers of find change grep options to false
        set include locked layers for find of find change grep options to false
        set include locked stories for find of find change grep options to false
        set include master pages of find change grep options to false
        --Find the tags.
        set find what of find grep preferences to "(?i)^<\\s*\\w+\\s*>"
        tell myStory
            set myFoundItems to find grep
        end tell
        if (count myFoundItems) is not equal to 0 then
            set myFoundTags to {}
            repeat with myCounter from 1 to (count myFoundItems)
                set myFoundTag to contents of item myCounter of myFoundItems
                if myFoundTags does not contain myFoundTag then
                    copy myFoundTag to end of myFoundTags
                end if
            end repeat
            --At this point, we have a list of tags to search for.
            repeat with myCounter from 1 to (count myFoundTags)
                set myString to item myCounter of myFoundTags
                --Find the tag using find what.
                set find what of find text preferences to myString
                --Extract the style name from the tag.
                set myStyleName to text 2 through ((count characters of myString) - 1)
                of myString
                tell myDocument
                    --Create the style if it does not already exist.
                    try
                        set myStyle to paragraph style myStyleName
                    on error
                        set myStyle to make paragraph style with properties
                        {name:myStyleName}
                    end try
                end tell
                --Apply the style to each instance of the tag.
                set applied paragraph style of change text preferences to myStyle
                tell myStory
                    change text
                end tell
                --Reset the change text preferences.
```

```
            set change text preferences to nothing
            --Set the change to property to an empty string.
            set change to of change text preferences to ""
            --Search to remove the tags.
            tell myStory
                change text
            end tell
            --Reset the find/change preferences again.
            set change text preferences to nothing
        end repeat
    end if
    --Reset the findGrepPreferences.
    set find grep preferences to nothing
end tell
end myReadPMTags
```

Using glyph search

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

```
--Clear glyph search preferences.
set find glyph preferences to nothing
set change glyph preferences to nothing
set myDocument to document 1
--You must provide a font that is used in the document for the
--applied font property of the find glyph preferences object.
set applied font of find glyph preferences to applied font of character 1 of story 1 of
myDocument
--Provide the glyph ID, not the glyph Unicode value.
set glyph ID of find glyph preferences to 374
--The applied font of the change glyph preferences object can be
--any font available to the application.
set applied font of change glyph preferences to "ITC Zapf DingbatsMedium"
set glyph ID of change glyph preferences to 85
tell myDocument
    change glyph
end tell
--Clear glyph search preferences.
set find glyph preferences to nothing
set change glyph preferences to nothing
```

Working with Tables

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

```
--Given a document "myDocument" containing a story...
set myStory to story 1 of myDocument
tell myStory
    set myStartCharacter to index of character 1 of paragraph 7
    set myEndCharacter to index of character -2 of paragraph 7
    set myText to object reference of text from character
    myStartCharacter to character myEndCharacter
    --The convertToTable method takes three parameters:
    --[column separator as string]
    --[row separator as string]
    --[number of columns as integer] (only used if the column separator
    --and row separator values are the same)
    --In the last paragraph in the story, columns are separated by commas
    --and rows are separated by semicolons, so we provide those characters
    --to the method as parameters.
    tell myText
        set myTable to convert to table column separator "," row separator ";"
    end tell
    set myStartCharacter to index of character 1 of paragraph 2
    set myEndCharacter to index of character -2 of paragraph 5
    set myText to object reference of text from character myStartCharacter
    to character myEndCharacter
    --In the second through the fifth paragraphs, colums are separated by
    --tabs and rows are separated by returns. These are the default delimiter
    --parameters, so we don't need to provide them to the method.
    tell myText
        set myTable to convert to table column separator tab row separator return
    end tell
    --You can also explicitly add a table--you don't have to convert text to a table.
    tell insertion point -1
        set myTable to make table
        set column count of myTable to 3
        set body row count of myTable to 3
    end tell
end tell
```

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

```
--Given a document "myDocument" containing a story...
tell story 1 of myDocument
    tell table 1
        --Merge all of the cells in the first column.
        merge cell 1 of column 1 with cell -1 of column 1
        --Convert column 2 into 2 cells (rather than 4).
        merge cell 3 of column 2 with cell -1 of column 2
        merge cell 1 of column 2 with cell 2 of column 2
        --Merge the last two cells in row 1.
        merge cell -2 of row 1 with cell -1 of row 1
        --Merge the last two cells in row 3.
        merge cell -2 of row 3 with cell -1 of row 3
    end tell
end tell
```

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

```
tell table 1 of story 1 of document 1
    split cell 1 using horizontal
    split column 1 using vertical
    split cell 1 using vertical
    split row -1 using horizontal
    split cell -1 using vertical
    --Fill the cells with row:cell labels.
    repeat with myRowCounter from 1 to (count rows)
        set myRow to row myRowCounter
        repeat with myCellCounter from 1 to (count cells of myRow)
            set myString to "Row: " & myRowCounter & " Cell: " & myCellCounter
            set contents of text 1 of cell myCellCounter of row myRowCounter to myString
        end repeat
    end repeat
end tell
```

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

```
--Given a document containing a story that contains a table...
tell table 1 of story 1 of document 1
    --Convert the first row to a header row.
    set row type of row 1 to header row
    --Convert the last row to a footer row.
    set row type of row -1 to footer row
end tell
```

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

```
set myTable to table 1 of story 1 of myDocument
tell myTable
    --Convert the first row to a header row.
    set row type of row 1 to header row
    --Use a reference to a swatch, rather than to a color.
    set fill color of row 1 to swatch "DGC1_446b" of myDocument
    set fill tint of row 1 to 40
    set fill color of row 2 to swatch "DGC1_446a" of myDocument
    set fill tint of row 2 to 40
    set fill color of row 3 to swatch "DGC1_446a" of myDocument
    set fill tint of row 3 to 20
    set fill color of row 4 to swatch "DGC1_446a" of myDocument
    set fill tint of row 4 to 40
    tell every cell in myTable
        set top edge stroke color to swatch "DGC1_446b" of myDocument
        set top edge stroke weight to 1
        set bottom edge stroke color to swatch "DGC1_446b" of myDocument
        set bottom edge stroke weight to 1
        --When you set a cell stroke to a swatch, make certain that
        --you also set the stroke weight.
        set left edge stroke color to swatch "None" of myDocument
        set left edge stroke weight to 0
        set right edge stroke color to swatch "None" of myDocument
        set right edge stroke weight to 0
    end tell
end tell
```

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

```
--Given a document "myDocument" containing a story that
--contains a table...
set myTable to table 1 of story 1 of myDocument
tell myTable
    --Apply alternating fills to the table.
    set alternating fills to alternating rows
    set start row fill color to swatch "DGC1_446a" of myDocument
    set start row fill tint to 60
    set end row fill color to swatch "DGC1_446b" of myDocument
    set end row fill tint to 50
end tell
```

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.)

```
--Check to see if any documents are open.
if (count documents) is not equal to 0 then
    --If the selection contains more than one item, the selection
    --is not text selected with the Type tool.
    set mySelection to selection
    if (count mySelection) is not equal to 0 then
        --Evaluate the selection based on its type.
        set myTextClasses to {insertion point, word, text style range,
        line, paragraph, text column, text, story}
        if class of item 1 of selection is in myTextClasses then
            --The object is a text object; display the text object type.
            --A practical script would do something with the selection,
            --or pass the selection on to a function.
            if class of parent of item 1 of mySelection is cell then
                display dialog ("The selection is inside a table cell")
            else
                display dialog ("The selection is not in a table")
            end if
        else if class of item 1 of selection is cell then
            display dialog ("The selection is a table cell")
        else if class of item 1 of selection is row then
            display dialog ("The selection is a table row")
        else if class of item 1 of selection is column then
            display dialog ("The selection is a table column")
        else if class of item 1 of selection is table then
            display dialog ("The selection is a table.")
        else
            display dialog ("The selection is not in a table")
        end if
    else
        display dialog ("Please select some text and try again.")
    end if
else
    display dialog ("Nothing is selected. Please select some text and try again.")
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):

```
--Given a document "myDocument" with a rectangle on page 1...
set myRectangle to rectangle 1 of page 1 of myDocument
tell myRectangle
    set myTextPath to make text path with properties {contents:"This is path text."}
end tell
```

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 93).

Autocorrect

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

```
tell auto correct preferences
    set autocorrect to true
    set auto correct capitalization errors to true
    --Add a word pair to the autocorrect list. Each auto correct table
    --is linked to a specific language.
end tell
set myAutoCorrectTable to auto correct table "English: USA"
--To safely add a word pair to the auto correct table, get the current
--word pair list, then add the new word pair to that array, and then
--set the autocorrect word pair list to the array.
set myWordPairList to {}
set myWordPairList to myWordPairList & auto correct word pair list of
myAutoCorrectTable
--Add a new word pair to the array.
set myWordPairList to myWordPairList & {{"paragarph", "paragraph"}}
--Update the word pair list.
set auto correct word pair list of auto correct table "English: USA" to myWordPairList
--To clear all autocorrect word pairs in the current dictionary:
--myAutoCorrectTable.autoCorrectWordPairList to {{}}
```

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 to document 1
tell footnote options of myDocument
    set separator text to tab
    set marker positioning to superscript marker
end tell
set myStory to story 1 of myDocument
--Add four footnotes at random locations in the story.
local myStoryLength, myRandomNumber
set myStoryLength to (count words of myStory)
repeat with myCounter from 1 to 5
    set myRandomNumber to my myGetRandom(1, myStoryLength)
    tell insertion point -1 of word myRandomNumber of myStory
        set myFootnote to make footnote
    end tell
    --Note: when you create a footnote, it contains text--the footnote marker
    --and the separator text (if any). If you try to set the text of the footnote
    --by setting the footnote contents, you will delete the marker. Instead, append
    --the footnote text, as shown below.
    tell insertion point -1 of myFootnote
        set contents to "This is a footnote."
    end tell
end repeat
```

Span Columns

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

```
set myDocument to active document
set myPage to page 1 of myDocument
set myTextFrame to item 1 of text frames of myPage
tell myTextFrame
    set text column count of text frame preferences to 3
    set myStory to parent story
    tell item 1 of paragraphs of myStory
        --split column
        set span column type to split columns
        set span split column count to 2
        set split column outside gutter to 0
        set split column inside gutter to 1
    end tell
    set mySpanIndex to (count of paragraphs of myStory) div 2 + 1
    tell item mySpanIndex of paragraphs of myStory
        --span columns
        set span column type to span columns
        set span split column count to all
    end tell
end tell
```

Setting Text Preferences

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

```
tell text preferences
    set abut text to text wrap to true
    --baseline shift key increment can range from .001 to 200 points.
    set baseline shift key increment to 1
    set highlight custom spacing to false
    set highlight hj violations to true
    set highlight keeps to true
    set highlight substituted fonts to true
    set highlight substituted glyphs to true
    set justify text wraps to true
    --kerning key increment value is 1/1000 of an em.
    set kerning key increment to 10
    --leading key increment value can range from .001 to 200 points.
    set leading key increment to 1
    set link text files when importing to false
    set show invisibles to true
    set small cap to 60
    set subscript position to 30
    set subscript size to 60
    set superscript position to 30
    set superscript size to 60
    set typographers quotes to false
    set use optical size to false
    set use paragraph leading to false
    set z order text wrap to false
end tell
--Text editing preferences are application-wide.
tell text editing preferences
    set allow drag and drop text in story to true
    set drag and drop text in layout to true
    set smart cut and paste to true
    set triple click selects line to false
end tell
```

7 User Interfaces

AppleScript 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.

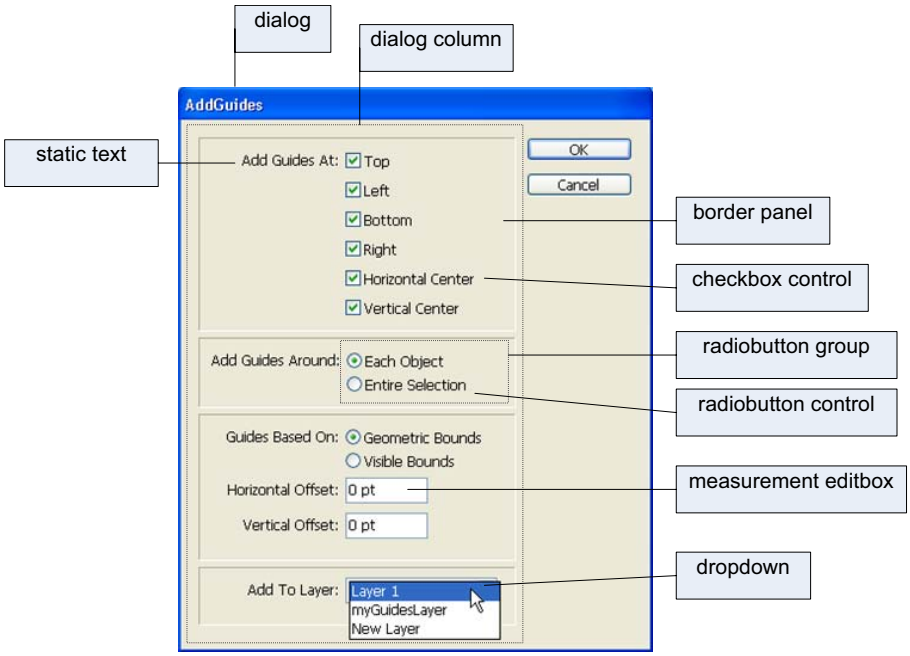
This chapter shows how to work with InDesign dialog scripting. The sample scripts in this chapter are presented in order of complexity, starting with very simple scripts and building toward more complex operations.

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 CS5 JavaScript Tools Guide*.

We assume you already read *Adobe InDesign CS5 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 `dialog column` object. `dialog columns` give you a way to control the positioning of controls within a dialog box. Inside `dialog columns`, you can further subdivide the dialog box into other `dialog columns` or `border panels` (both of which can, if necessary, contain more `dialog columns` and `border panels`).

Like any other InDesign scripting object, each part of a dialog box has its own properties. A `checkbox control`, for example, has a property for its text (`static label`) and another property for its state (`checked state`). The `dropdown` control has a property (`string list`) 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 myDialog to make dialog with properties {name:"Simple Dialog"}
tell myDialog
    tell (make dialog column)
        make static text with properties {static label:"This is a very
        simple dialog box."}
    end tell
end tell
--Show the dialog box.
set myResult to show myDialog
--If the user clicked OK, display one message;
--if they clicked Cancel, display a different message.
if myResult is true then
    display dialog ("You clicked the OK button")
else
    display dialog ("You clicked the Cancel button")
end if
--Remove the dialog box from memory.
destroy myDialog
```

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 CS5 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:

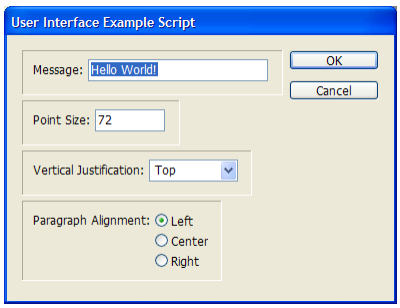
```
set myDialog to make dialog
tell myDialog
    set name to "Simple User Interface Example Script"
    set myDialogColumn to make dialog column
    tell myDialogColumn
        --Create a text entry field.
        set myTextEditField to make text editbox with properties
        {edit contents:"Hello World!", min width:180}
        --Create a number (real) entry field
        set myPointSizeField to make measurement editbox with properties
        {edit value:72, edit units:points}
    end tell
end tell
set myResult to show myDialog
if myResult is true then
    --Get the settings from the dialog box.
    --Get the point size from the point size field.
    set myPointSize to edit value of myPointSizeField
    --Get the example text from the text edit field.
    set myString to edit contents of myTextEditField
    --Remove the dialog box from memory.
    destroy myDialog
    my myMakeDocument(myPointSize, myString)
else
    destroy myDialog
end if
```

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

```
on myMakeDocument(myPointSize, myString)
    tell application "Adobe InDesign CS5"
        set myDocument to make document
        tell view preferences of myDocument
        end tell
        tell page 1 of myDocument
            --Create a text frame.
            set myTextFrame to make text frame
            set geometric bounds of myTextFrame to my myGetBounds
            (myDocument, page 1 of myDocument)
            --Apply the settings from the dialog box to the text frame.
            set contents of myTextFrame to myString
            --Set the point size of the text in the text frame.
            set point size of text 1 of myTextFrame to myPointSize
        end tell
    end tell
end myMakeDocument
```

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 to make dialog
--This example dialog box uses border panels and dialog columns to
--separate and organize the user interface items in the dialog.
    tell myDialog
        set name to "User Interface Example Script"
        set myDialogColumn to make dialog column
        tell myDialogColumn
            set myBorderPanel to make border panel
            tell myBorderPanel
                set myDialogColumn to make dialog column
                tell myDialogColumn
                    make static text with properties {static label:"Message:"}
                end tell
                set myDialogColumn to make dialog column
                tell myDialogColumn
                    set myTextEditField to make text editbox with properties
                    {edit contents:"Hello World!", min width:180}
                end tell
            end tell
            set myBorderPanel to make border panel
            tell myBorderPanel
                set myDialogColumn to make dialog column
```

```
            tell myDialogColumn
                make static text with properties {static label:"Point Size:"}
            end tell
            set myDialogColumn to make dialog column
            tell myDialogColumn
                set myPointSizeField to make measurement editbox with
                properties {edit value:72, edit units:points}
            end tell
        end tell
        set myBorderPanel to make border panel
        tell myBorderPanel
            set myDialogColumn to make dialog column
            tell myDialogColumn
                make static text with properties
                {static label:"Vertical Justification:"}
            end tell
            set myDialogColumn to make dialog column
            tell myDialogColumn
                set myVerticalJustificationMenu to make dropdown with properties
                {string list:{"Top", "Center", "Bottom"}, selected index:0}
            end tell
        end tell
        set myBorderPanel to make border panel
        tell myBorderPanel
            make static text with properties {static label:"Paragraph Alignment:"}
            set myParagraphAlignmentGroup to make radiobutton group
            tell myParagraphAlignmentGroup
                set myLeftRadioButton to make radiobutton control with
                properties {static label:"Left", checked state:true}
                set myCenterRadioButton to make radiobutton control with
                properties {static label:"Center"}
                set myRightRadioButton to make radiobutton control with
                properties {static label:"Right"}
            end tell
        end tell
    end tell
end tell
set myResult to show myDialog
if myResult is true then
    --Get the settings from the dialog box.
    --Get the point size from the point size field.
    set myPointSize to edit value of myPointSizeField
    --Get the example text from the text edit field.
    set myString to edit contents of myTextEditField
    --Get the vertical justification setting from the pop-up menu.
    if selected index of myVerticalJustificationMenu is 0 then
        set myVerticalJustification to top align
    else if selected index of myVerticalJustificationMenu is 1 then
        set myVerticalJustification to center align
    else
        set myVerticalJustification to bottom align
    end if
    --Get the paragraph alignment setting from the radiobutton group.
    get properties of myParagraphAlignmentGroup
```

```
        if selected button of myParagraphAlignmentGroup is 0 then
            set myParagraphAlignment to left align
        else if selected button of myParagraphAlignmentGroup is 1 then
            set myParagraphAlignment to center align
        else
            set myParagraphAlignment to right align
        end if
        --Remove the dialog box from memory.
        destroy myDialog
        my myMakeDocument(myPointSize, myString,
        myParagraphAlignment, myVerticalJustification)
    else
        destroy myDialog
    end if
end tell
```

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

```
on myMakeDocument(myPointSize, myString, myParagraphAlignment,
myVerticalJustification)
    tell application "Adobe InDesign CS5"
        set myDocument to make document
        tell view preferences of myDocument
        end tell
        set myPage to page 1 of myDocument
        tell myPage
            set myTextFrame to make text frame
            set geometric bounds of myTextFrame to my myGetBounds(myDocument, myPage)
            --Apply the settings from the dialog box to the text frame.
            set contents of myTextFrame to myString
            --Apply the vertical justification setting.
            set vertical justification of text frame preferences of
            myTextFrame to myVerticalJustification
            --Apply the paragraph alignment ("justification").
            --"text 1 of myTextFrame" is all of the text in the text frame.
            set justification of text 1 of myTextFrame to myParagraphAlignment
            --Set the point size of the text in the text frame.
            set point size of text 1 of myTextFrame to myPointSize
        end tell
    end tell
end myMakeDocument
```

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 AppleScript users. InDesign scripts can execute scripts written in other scripting languages using the do script 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 an AppleScript (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 an AppleScript (for the complete script, see CallProgressBar):

```
set myDocument to make document
--Add pages to the active document.
--If you don't do this, the progress bar will
--go by too quickly.
--Note that the JavaScripts must use the "session"
--engine for this to work.
set myJavaScript to "#targetengine \"session\"" & return
set myJavaScript to myJavaScript & "myCreateProgressPanel(100, 400);" & return
set myJavaScript to myJavaScript & "myProgressPanel.show();" & return
do script myJavaScript language javascript
repeat with myCounter from 1 to 40
    set myJavaScript to "#targetengine \"session\"" & return
    set myJavaScript to myJavaScript & "myProgressPanel.myProgressBar.value = "
    set myJavaScript to myJavaScript & myCounter & "/myIncrement;" & return
    do script myJavaScript language javascript
    tell myDocument to make page
    if myCounter = 100 then
        set myJavaScript to "#targetengine \"session\"" & return
        set myJavaScript to myJavaScript &
        "myProgressPanel.myProgressBar.value = 0;" & return
        set myJavaScript to myJavaScript & "myProgressPanel.hide();" & return
        do script myJavaScript language javascript
        close myDocument saving no
    end if
end repeat
```

8 Events

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

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

We assume that you have already read *Adobe InDesign CS5 Scripting Tutorial* and know how to create, install, and run a script.

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

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

Understanding the Event Scripting Model

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

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

You can view the available events using the AppleScript Dictionary Viewer. In the AppleScript Dictionary viewer in your AppleScript editor, look at the event class in the Basics Suite.

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 `event listener` registered for that event, the `event listener` is triggered by the event. An event can be handled by more than one object as it propagates.

There are two types of event propagation:

- **None —** Only the `event listeners` registered to the event target are triggered by the event. The `beforeDisplay` event is an example of an event that does not propagate.
- **Bubbling** — The event starts propagation at its `target` and triggers any qualifying `event listeners` registered to the `target`. The event then proceeds upward through the scripting object model, triggering any qualifying `event listeners` 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 prevent default command .
CurrentTarget	The current scripting object processing the event. See target in this table.
DefaultPrevented	If true, the default behavior of the event on the current target was prevented, thereby canceling the action. See target in this table.
EventPhase	The current stage of the event propagation process.
EventType	The type of the event, as a string (for example, "beforeNew").
PropagationStopped	If true, the event has stopped propagating beyond the current target (see target in this table). To stop event propagation, use the stop propagation command .
Target	The object from which the event originates. For example, the target of a beforeImport event is a document; of a beforeNew event, the application.
TimeStamp	The time and date when the event occurred.

Working with Event Listeners

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

```
--Registers an event listener on the afterNew event.
tell application "Adobe InDesign CS5"
    make event listener with properties {event type:"afterNew",
    handler:my myDisplayEventType}
end tell
```

The preceding script fragment refers to the following handler:

```
on myDisplayEventType()
    tell application "Adobe InDesign CS5"
        --"evt" is the event passed to this script by the event listener.
        set myEvent to evt
        display dialog ("This event is the "& event type of myEvent & "event.")
    end tell
end myMessage
```

When you use an event handler defined in the script, rather than an event handler defined in a separate script file, the script will work correctly when run from the InDesign Scripts panel, but will not work when you run it from a script editing application (such as the Apple Script Editor).

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

```
tell application "Adobe InDesign CS5"
    remove event listener event type "afterNew" handler myDisplayEventType
end tell
```

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

event listeners do not persist beyond the current InDesign session. To make an event listener available in every InDesign session, add the script to the startup scripts folder. (For more on installing scripts, see "Installing Scripts" in *Adobe InDesign CS5 Scripting Tutorial*.) When you add an event listener script to a document, it is not saved with the document or exported to IDML.

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

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

```
--Shows that an event can trigger multiple event listeners.
tell application "Adobe InDesign CS5"
    set myDocument to make document
    --You'll have to fill in a valid file path for your system
    make event listener with properties {event type:"beforeImport",
    handler:my myEventInfo}
    tell myDocument
        make event listener with properties {event type:"beforeImport",
        handler:my myEventInfo}
    end tell
end tell
```

The handler referred to in the preceding script fragment contains the following:

```
on myEventInfo(myEvent)
    tell application "Adobe InDesign CS5"
        set myString to "Current Target: " & name of current target of myEvent
        display dialog (myString)
    end tell
end myEventInfo
```

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

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

```
tell application "Adobe InDesign CS5"
set myEventNames to {"beforeNew", "afterNew", "beforeQuit", "afterQuit", "beforeOpen",
"afterOpen", "beforeClose", "afterClose", "beforeSave", "afterSave", "beforeSaveAs",
"afterSaveAs", "beforeSaveACopy", "afterSaveACopy", "beforeRevert", "afterRevert",
"beforePrint", "afterPrint", "beforeExport", "afterExport", "beforeImport",
"afterImport", "beforePlace", "afterPlace"}
    repeat with myEventName in myEventNames
        make event listener with properties {event type:myEventName,
        handler:"yukino:IDEventHandlers:GetEventInfo.applescript"}
    end repeat
end tell
```

The preceding script refers to the following script. The file reference in the preceding script must match the location of this script on your disk. For the complete script, see `GetEventInfo.applescript`.

```
main(evt)
on main(myEvent)
    tell application "Adobe InDesign CS5"
        set myString to "Handling Event: " & event type of myEvent & return
        set myString to myString & "Target: " & name of target of myEvent & return
        set myString to myString & "Current: " & name of current target of
        myEvent & return
        set myString to myString & "Phase: " & my myGetPhaseName(event phase o
        f myEvent) & return
        set myString to myString & "Bubbles: " & bubbles of myEvent & return
        set myString to myString & "Cancelable: " & cancelable of
        myEvent & return
        set myString to myString & "Stopped: " & propagation stopped of
        myEvent & return
        set myString to myString & "Canceled: " & default prevented of
        myEvent & return
        set myString to myString & "Time: " & time stamp of myEvent & return
        display dialog (myString)
    end tell
end main
--Function returns a string corresponding to the event phase.
on myGetPhaseName(myEventPhase)
    tell application "Adobe InDesign CS5"
        if myEventPhase is at target then
            set myString to "At Target"
        else if myEventPhase is bubbling phase then
            set myString to "Bubbling"
        else if myEventPhase is done then
            set myString to "Done"
        else if myEventPhase is not dispatching then
            set myString to "Not Dispatching"
        else
            set myString to "Unknown Phase"
        end if
        return myString
    end tell
end myGetPhaseName
```

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

```
-tell application "Adobe InDesign CS5"
    tell event listeners to delete
end tell
```

Sample afterNew Event Listener

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

```
--Registers an event listener on the afterNew event.
tell application "Adobe InDesign CS5"
    make event listener with properties {event type:"afterNew",
handler:"yukino:IDEventHandlers:AfterNewHandler.applescript"}
end tell
```

The preceding script refers to the following script. The file reference in the preceding script must match the location of this script on your disk. For the complete script, see `AfterNewHandler.applescript`.

```
--AfterNewHandler.applescript
--An InDesign CS5 AppleScript
--
--This script is called by the AfterNew.applescript. It
--Sets up a basic document layout and adds XMP information
--to the document.
main(evt)
on main(myEvent)
    tell application "Adobe InDesign CS5"
        if user name = "" then
            set user name to "Adobe"
        end if
        set myUserName to user name
        --set myDocument to parent of myEvent
        set myDocument to document 1
        tell view preferences of myDocument
            set horizontal measurement units to points
            set vertical measurement units to points
            set ruler origin to page origin
        end tell
        --MySlugOffset is the distance from the bottom of the page
        --to the top of the slug.
        set mySlugOffset to 12
        --MySlugHeight is the height of the text frame
        --containing the job information.
        set mySlugHeight to 72
        tell document preferences of myDocument
            set documentSlugUniformSize to false
            set slug bottom offset to mySlugOffset + mySlugHeight
            set slug top offset to 0
            set slug inside or left offset to 0
            set slug right or outside offset to 0
        end tell
        repeat with myCounter from 1 to (count master spreads of myDocument)
            set myMasterSpread to master spread myCounter of myDocument
            repeat with myMasterPageCounter from 1 to (count pages of
            myMasterSpread)
                set myPage to page myMasterPageCounter of myMasterSpread
                set mySlugBounds to my myGetSlugBounds(myDocument, myPage,
                mySlugOffset, mySlugHeight)
                tell myPage
                    set mySlugFrame to make text frame with properties {geometric
```

```
                bounds:mySlugBounds, contents:"Created: " &
                time stamp of myEvent &
                return & "by: " & myUserName}
            end tell
        end repeat
    end repeat
    tell metadata preferences of myDocument
        set author to "Adobe Systems"
        set description to "This is an example document
        containing XMP metadata. Created: " & time stamp of myEvent
    end tell
  end tell
end main
on myGetSlugBounds(myDocument, myPage, mySlugOffset, mySlugHeight)
    tell application "Adobe InDesign CS5"
        tell myDocument
            set myPageWidth to page width of document preferences
            set myPageHeight to page height of document preferences
        end tell
        set myLeft to left of margin preferences of myPage
        set myRight to right of margin preferences of myPage
        set myX1 to myLeft
        set myY1 to myPageHeight + mySlugOffset
        set myX2 to myPageWidth - myRight
        set myY2 to myY1 + mySlugHeight
        return {myY1, myX1, myY2, myX2}
    end tell
end myGetSlugBounds
```

Sample beforePrint Event Listener

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

```
--Adds an event listener that performs a preflight check on
--a document before printing. If the preflight check fails,
--the script gives the user the opportunity to cancel the print job.
tell application "Adobe InDesign CS5"
    make event listener with properties {event type:"beforePrint",
handler:"yukino:IDEventHandlers:BeforePrintHandler.applescript"}
end tell
```

The preceding script refers to the following script. The file reference in the preceding script must match the location of this script on your disk. For the complete script, see `BeforePrintHandler.applescript`.

```
--BeforePrintHandler.applescript
--An InDesign CS5 AppleScript
--
--Peforms a preflight check on a document. Called by the
--BeforePrint.applescript event listener example.
--"evt" is the event passed to this script by the event listener.
main(evt)
on main(myEvent)
    tell application "Adobe InDesign CS5"
        --The parent of the event is the document.
        set myDocument to parent of myEvent
        if my myPreflight(myDocument) is false then
            tell myEvent
                stop propagation
                prevent default
            end tell
            display dialog ("Document did not pass preflight check.
            Please fix the problems and try again.")
        else
            display dialog ("Document passed preflight check. Ready to print.")
        end if
    end tell
end main
on myPreflight(myDocument)
    set myPreflightCheck to true
    set myFontCheck to my myCheckFonts(myDocument)
    set myGraphicsCheck to my myCheckGraphics(myDocument)
    display dialog ("Fonts: " & myFontCheck & return & "Links:" & myGraphicsCheck)
    if myFontCheck = false or myGraphicsCheck = false then
        set myPreflightCheck to false
        return myPreflightCheck
    end if
end myPreflight
on myCheckFonts(myDocument)
    tell application "Adobe InDesign CS5"
        set myFontCheck to true
        repeat with myCounter from 1 to (count fonts of myDocument)
            set myFont to font myCounter of myDocument
            if font status of myFont is not installed then
                set myFontCheck to false
                exit repeat
            end if
        end repeat
        return myFontCheck
    end tell
end myCheckFonts
on myCheckGraphics(myDocument)
    tell application "Adobe InDesign CS5"
        set myGraphicsCheck to true
        repeat with myCounter from 1 to (count graphics of myDocument)
            set myGraphic to graphic myCounter of myDocument
            set myLink to item link of myGraphic
            if link status of myLink is not normal then
                set myGraphicsCheck to false
                exit repeat
            end if
        end repeat
        return myGraphicsCheck
    end tell
end myCheckGraphics
```

Sample Selection Event Listeners

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

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

```
set myDocument to make document
tell myDocument
    set myEventListener to make event listener with properties {event
type:"afterSelectionChanged",
    handler:my myDisplaySelectionType}
end tell
```

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

```
on myDisplaySelectionType(en)
    tell application "Adobe InDesign CS5"
        if (count documents) is greater than 0 then
            tell document 1
                set mySelection to selection
                if (count mySelection) is greater than 0 then
                    set myString to "Selection Contents:" & return
                    repeat with myCounter from 1 to (count mySelection)
                        set myString to myString & class of item myCounter of
                        mySelection & return
                    end repeat
                    display dialog myString
                end if
            end tell
        end if
    end tell
end myDisplaySelectionType
```

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

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

```
set myDocument to make document
tell myDocument
    set myEventListener to make event listener with properties {event
type:"afterSelectionAttributeChanged", handler:my myCheckForRegistration}\
end tell
```

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

```
on myCheckForRegistration()
    tell application "Adobe InDesign CS5"
        if (count documents) is greater than 0 then
            tell document 1
                set mySelection to selection
                if (count mySelection) is greater than 0 then
                    set myRegistrationSwatchUsed to false
                    repeat with myCounter from 1 to (count mySelection)
                        set myFillColor to fill color of item myCounter of mySelection
                        set myStrokeColor to stroke color of item myCounter of mySelection
                        if name of myFillColor is "Registration" or
                        name of myStrokeColor is "Registration" then
                            set myRegistrationSwatchUsed to true
                        end if
                    end repeat
                    if myRegistrationSwatchUsed is true then
                        display dialog "The Registration swatch is applied to some of the"
& return &
                        "objects in the selection. Did you really intend to apply this
swatch?"
                    end if
                end if
            end tell
        end if
    end tell
end myCheckForRegistration
```

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

Sample onIdle Event Listener

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

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

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

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

```
set myIdleTaskName to "my_idle_task"
set myIdleTask to make idle task with properties {name:myIdleTaskName, sleep:10000}
tell myIdleTask
    --You need to fill in your own file path.
    set fileName to "Macintosh HD:scripting:OnIdleEventHandler.applescript"
    set onIdleEventListener to make event listener with properties {event
type:"onIdle", handler:fileName}
    display alert "Created idle task " & name & "; added event listener on " & event type
of onIdleEventListener
end tell
```

The event handler is a script file, OnIdleEventHandler.applescript.

```
--"evt" is the event passed to this script by the event listener.
onIdleEventHandler(evt)
on onIdleEventHandler(myIdleEvent)
    tell application "Adobe InDesign CS5"
        if (count of documents) = 0 then
            set myDoc to make document
            display alert "Created document " & name of myDoc & " in idle task."
            return
        end if

        tell item 1 of pages of active document
            if (count of text frames) = 0 then
                make text frame with properties {geometric bounds:["72pt", "72pt",
"288pt", "288pt"], contents:"Text frame created in idle task"}
                display alert "Created a text frame in idle task."
                return
            end if
        end tell

        --Delete idle task by setting its sleep time to zero.
        set sleep of parent of myIdleEvent to 0
        display alert "Nothing to do. Delete idle task."
    end tell
end onIdleEventHandler
```

To remove the idle task created by preceding script, run the following script (for the complete script, see RemoveIdleTask):

```
set taskCount to count of idle tasks
if taskCount is 0 then
    display alert "There is no idle task."
else
    set myIdleTaskName to "my_idle_task"
    repeat with i from taskCount to 1 by -1
        set myIdleTask to item i of idle tasks
        if name of myIdleTask is myIdleTaskName then
            delete myIdleTask
        end if
    end repeat
    display alert "Idle task " & myIdleTaskName & " removed."
end if
```

To remove all idle tasks, run the following script (for the complete script, see RemoveAllIdleTasks):

```
set taskCount to count of idle tasks
if taskCount is 0 then
    display alert "There is no idle task."
else
    repeat with i from taskCount to 1 by -1
        set myIdleTask to item i of idle tasks
        delete myIdleTask
    end repeat
    display alert "" & taskCount & " idle task(s) removed."
end if
```

To list existing idle tasks, run the following script (for the complete script, see ListIdleTasks):

```
set taskCount to count of idle tasks
if taskCount is 0 then
    display alert "There is no idle task."
else
    set str to ""
    repeat with i from 1 to taskCount
        set myIdleTask to item i of idle tasks
        tell myIdleTask
            set str to str & "idle task " & id & ": " & name & return
        end tell
    end repeat
    display alert str
end if
```

9 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 CS5 Scripting Tutorial* and know how to create, install, and run a script.

Understanding the Menu Model

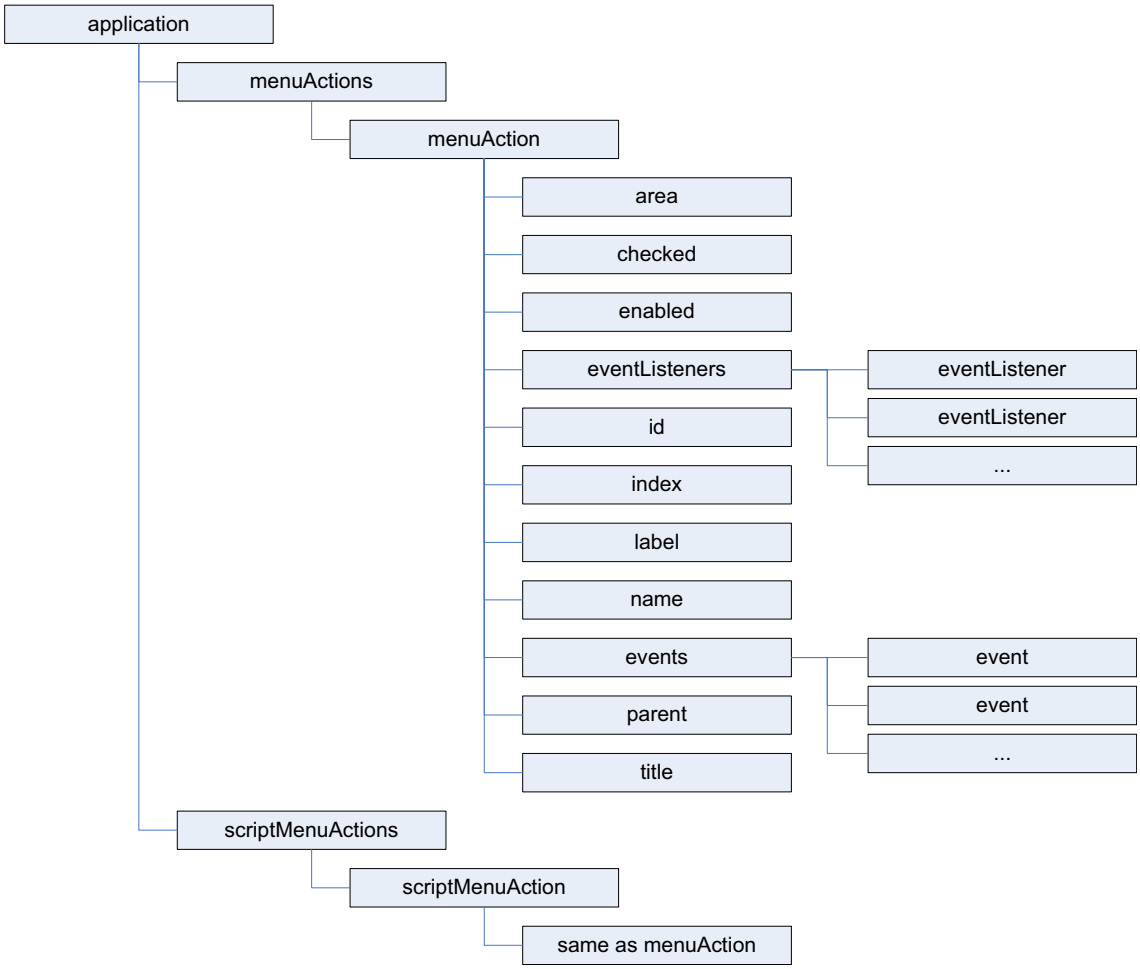
The InDesign menu-scripting model is made up of a series of objects that correspond to the menus you see in the application's user interface, including menus associated with panels as well as those displayed on the main menu bar. A `menu` object contains the following objects:

- `menu items` — The menu options shown on a menu. This does not include submenus.
- `menu separators` — Lines used to separate menu options on a menu.
- `submenus` — Menu options that contain further menu choices.
- `menu elements` — All `menu items`, `menu separators` and `submenus` shown on a menu.
- `event listeners` — These respond to user (or script) actions related to a menu.
- `events` — The `events` triggered by a menu.

Every `menu item` is connected to a `menu action` through the `associated menu action` property. The properties of the `menu action` define what happens when the menu item is chosen. In addition to the `menu actions` defined by the user interface, InDesign scripters can create their own, `script menu actions`, which associate a script with a menu selection.

A `menu action` or `script menu action` can be connected to zero, one, or more `menu items`.

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 myTextFile to choose file name("Save menu action names as:")
if myTextFile is not equal to "" then
    tell application "Adobe InDesign CS5"
        set myString to ""
        set myMenuActionNames to name of every menu action
        repeat with myMenuActionName in myMenuActionNames
            set myString to myString & myMenuActionName & return
        end repeat
        my myWriteToFile(myString, myTextFile, false)
    end tell
end if
on myWriteToFile(myString, myFileName, myAppendData)
    set myTextFile to open for access myFileName with write permission
    if myAppendData is false then
        set eof of myTextFile to 0
    end if
    write myString to myTextFile starting at eof
    close access myTextFile
end myWriteToFile
```

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.

```
--Open a new text file.
set myTextFile to choose file name ("Save Menu Action Names As")
--If the user clicked the Cancel button, the result is null.
if (myTextFile is not equal to "") then
    tell application "Adobe InDesign CS5"
        --Open the file with write access.
        my myWriteToFile("Adobe InDesign CS5 Menu Names" & return,
        myTextFile, false)
        repeat with myCounter from 1 to (count menus)
            set myMenu to item myCounter of menus
            set myString to "----------" & return & name of myMenu & return &
            "----------" & return
            set myString to my myProcessMenu(myMenu, myString)
            my myWriteToFile(myString, myTextFile, true)
        end repeat
        display dialog ("done!")
    end tell
end if
on myProcessMenu(myMenu, myString)
    tell application "Adobe InDesign CS5"
        set myIndent to my myGetIndent(myMenu)
        repeat with myCounter from 1 to (count menu elements of myMenu)
            set myMenuElement to menu element myCounter of myMenu
            set myClass to class of myMenuElement
            if myClass is not equal to menu separator then
                set myMenuElementName to name of myMenuElement
                set myString to myString & myIndent & myMenuElementName & return
                if class of myMenuElement is submenu then
                    if myMenuElementName is not "Font" then
                        set myString to my myProcessMenu(myMenuElement, myString)
                    end if
                end if
            end if
        end repeat
        return myString
    end tell
end myProcessMenu
on myGetIndent(myObject)
    tell application "Adobe InDesign CS5"
        set myString to ""
        repeat until class of myObject is menu
            set myString to myString & tab
            set myObject to parent of myObject
        end repeat
        return myString
    end tell
end myGetIndent
on myWriteToFile(myString, myFileName, myAppendData)
    set myTextFile to open for access myFileName with write permission
    if myAppendData is false then
        set eof of myTextFile to 0
    end if
    write myString to myTextFile starting at eof
    close access myTextFile
end myWriteToFile
```

Localization and menu names

in InDesign scripting, menu items, menus, menu actions,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):

```
tell application "Adobe InDesign CS5"
    set myMenuAction to menu action "Convert to Note"
    set myKeyStrings to find key strings for title of myMenuAction
    if class of myKeyStrings is list then
        repeat with myKeyString in myKeyStrings
            set myString to myKeyString & return
        end repeat
    else
        set myString to myKeyStrings
    end if
    display dialog(myString)
end tell
```

NOTE: It is much better to get the locale-independent name of a menu action than of a menu, menu item, or submenu, because the title of a menu action is more likely to be a single string. Many of the other menu objects return multiple strings when you use the find key strings 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):

```
tell application "Adobe InDesign CS5"
    set myString to translate key string for "$ID/NotesMenu_ConvertToNote"
    display dialog(myString)
end tell
```

Running a Menu Action from a Script

Any of InDesign's built-in menu actions can be run from a script. The menu action does not need to be attached to a menu item; however, in every other way, running a menu item 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 menu action from a script also displays a dialog box.

The following script shows how to run a menu action from a script (for the complete script, see InvokeMenuAction):

```
tell application "Adobe InDesign CS5"
    --Get a reference to a menu action.
    set myMenuAction to menu action "$ID/NotesMenu_ConvertToNote"
    --Run the menu action. The example action will fail if you do not
    --have text selected.
    invoke myMenuAction
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    set myMainMenu to menu "Main"
    set myTypeMenu to submenu "Type" of myMainMenu
    set myFontMenu to submenu "Font" of myTypeMenu
    set myKozukaMenu to submenu "Kozuka Mincho Pro " of myFontMenu
    tell myMainMenu
        set mySpecialFontMenu to make submenu with properties
        {title:"Kozuka Mincho Pro"}
    end tell
    repeat with myMenuItem in menu items of myKozukaMenu
        set myAssociatedMenuAction to associated menu action of myMenuItem
        tell mySpecialFontMenu
            make menu item with properties
            {associated menu action:myAssociatedMenuAction}
        end tell
    end repeat
end tell
```

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

```
tell application "Adobe InDesign CS3"
    set myMainMenu to menu "Main"
    try
        set myKozukaMenu to submenu "Kozuka Mincho Pro" of myMainMenu
        tell myKozukaMenu to delete
    end try
end tell
```

Menus and Events

Menus and submenus generate events as they are chosen in the user interface, and `menu actions` and `script menu actions` generate events as they are used. Scripts can install `event listeners` 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.
`menu action`	`afterInvoke`	Runs the attached script when the associated `menu item` is selected, but after the `onInvoke` event.
	`beforeInvoke`	Runs the attached script when the associated `menu item` is selected, but before the `onInvoke` event.

Object	Event	Description
script menu action	afterInvoke	Runs the attached script when the associated menu item is selected, but after the onInvoke event.
	beforeInvoke	Runs the attached script when the associated menu item is selected, but before the onInvoke event.
	beforeDisplay	Runs the attached script before an internal request for the enabled/checked status of the script menu actionscript menu action.
	onInvoke	Runs the attached script when the script menu action is invoked.
submenu	beforeDisplay	Runs the attached script before the contents of the submenu are shown.

For more about events and event listeners, see Chapter 8, "Events."

To change the items displayed in a menu, add an event listener for the beforeDisplay event. When the menu is selected, the event listener 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 script menu action to create a new menu action whose behavior is implemented through the script registered to run when the onInvoke event is triggered.

The following script shows how to create a script menu action 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.

```
tell application "Adobe InDesign CS5"
    --Create the script menu action "Display Message"
    --if it does not already exist.
    try
        set myScriptMenuAction to script menu action "Display Message"
    on error
        set myScriptMenuAction to make script menu action
        with properties {title:"Display Message"}
    end try
    tell myScriptMenuAction
        --If the script menu action already existed,
        --remove the existing event listeners.
        if (count event listeners) > 0 then
            tell every event listener to delete
        end if
        set myEventListener to make event listener with properties
        {event type:"onInvoke", handler:"yukino:message.applescript"}
    end tell
    tell menu "$ID/Main"
        set mySampleScriptMenu to make submenu with properties
        {title:"Script Menu Action"}
        tell mySampleScriptMenu
            set mySampleScriptMenuItem to make menu item with properties
            {associated menu action:myScriptMenuAction}
        end tell
    end tell
end tell
```

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

```
    tell application "Adobe InDesign CS5"
        display dialog ("You selected an example script menu action.")
    end tell
```

To remove the `menu`, `submenu`, `menu item`, and `script menu action` created by the above script, run the following script fragment (from the RemoveScriptMenuAction tutorial script):

```
tell application "Adobe InDesign CS5"
    try
        set myScriptMenuAction to script menu action "Display Message"
        tell myScriptMenuAction
            delete
        end tell
        tell submenu "Script Menu Action" of menu "$ID/Main" to delete
    end try
end tell
```

You also can remove all `script menu action`, as shown in the following script fragment (from the RemoveAllScriptMenuActions tutorial script). This script also removes the menu listings of the `script menu action`, but it does not delete any menus or submenus you might have created.

```
tell application "Adobe InDesign CS5"
    delete every script menu action
end tell
```

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

```
set myTextFile to choose file name {"Save Script Menu Action Names As"}
--If the user clicked the Cancel button, the result is null.
if myTextFile is not equal to "" then
    tell application "Adobe InDesign CS5"
        set myString to ""
        set myScriptMenuActionNames to name of every script menu action
        repeat with myScriptMenuActionName in myScriptMenuActionNames
            set myString to myString & myScriptMenuActionName & return
        end repeat
        my myWriteToFile(myString, myTextFile, false)
    end tell
end if
on myWriteToFile(myString, myFileName, myAppendData)
    set myTextFile to open for access myFileName with write permission
    if myAppendData is false then
        set eof of myTextFile to 0
    end if
    write myString to myTextFile starting at eof
    close access myTextFile
end myWriteToFile
```

`script menu actions` also can run scripts during their `beforeDisplay event`, in which case they are executed before an internal request for the state of the `script menu action` (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 `event listener` to the `beforeDisplay event` that checks the current selection. If there is no selection, the script in the `event listener` 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.)

```
tell application "Adobe InDesign CS5"
    --Create the script menu action "Display Message"
    --if it does not already exist.
    try
        set myScriptMenuAction to script menu action "Display Message"
    on error
        set myScriptMenuAction to make script menu action with properties
        {title:"Display Message"}
    end try
    tell myScriptMenuAction
        --If the script menu action already existed,
        --remove the existing event listeners.
        if (count event listeners) > 0 then
            tell every event listener to delete
        end if
        --Fill in a valid file path for your system.
```

```
        make event listener with properties {event type:"onInvoke",
        handler:"yukino:WhatIsSelected.applescript"}
    end tell
    tell menu "$ID/Main"
        set mySampleScriptMenu to make submenu with properties
        {title:"Script Menu Action"}
        tell mySampleScriptMenu
            set mySampleScriptMenuItem to make menu item with properties
            {associated menu action:myScriptMenuAction}
            --Fill in a valid file path for your system.
            make event listener with properties {event type:"beforeDisplay",
            handler:"yukino:BeforeDisplayHandler.applescript"}
        end tell
    end tell
end tell
```

The BeforeDisplayHander tutorial script file contains the following script:

```
tell application "Adobe InDesign CS5"
    try
        set mySampleScriptAction to script menu action "Display Message"
        set mySelection to selection
        if (count mySelection) > 0 then
            set enabled of mySampleScriptAction to true
        else
            set enabled of mySampleScriptAction to false
        end if
    on error
        alert("Script menu action did not exist.")
    end try
end tell
```

The WhatIsSelected tutorial script file contains the following script:

```
tell application "Adobe InDesign CS5"
    set mySelection to selection
    if (count mySelection) > 0 then
        set myString to class of item 1 of mySelection as string
        display dialog ("The first item in the selection is a " & myString & ".")
    end if
end tell
```

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 event listener` which checks for the existence of a `menu item` and removes it if it already exists. We do this to ensure the `menu item` 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 `event listener` then checks the selection to see if it contains a graphic; if so, it creates a new `script menu item`.

```
tell application "Adobe InDesign CS5"
    --The locale-independent name (aka "key string") for the
    --Layout context menu is "$ID/RtMouseLayout".
    set myLayoutMenu to menu "$ID/RtMouseLayout"
    --Note that the following script actions only create the script menu action
    --and set up event listeners--they do not actually add the menu item to the
    --Layout context menu. That job is taken care of by the event handler scripts
    --themselves. The Layout context menu will not display the menu item unless
    --a graphic is selected.
    tell myLayoutContextMenu
        set myBeforeDisplayEventListener to make event listener with
        properties{event type:"beforeDisplay",
        handler:"yukino:IDEventHandlers:LayoutMenuBeforeDisplay.applescript",
        captures:false}
    end tell
end tell
```

The `LayoutMenuBeforeDisplay.applescript` file referred to in the above example contains the following:

```
myBeforeDisplayHandler(evt)
on myBeforeDisplayHandler(myEvent)
    tell application "Adobe InDesign CS5"
        set myGraphicList to {PDF, EPS, image}
        set myParentList to {rectangle, oval, polygon}
        set myObjectList to {}
        set myLayoutContextMenu to menu "$ID/RtMouseLayout"
        if (count documents) > 0 then
            set mySelection to selection
            if (count mySelection) > 0 then
                repeat with myCounter from 1 to (count mySelection)
                    set myObject to item myCounter of mySelection
                    if class of myObject is in myGraphicList then
                        copy myObject to end of myObjectList
                    else if class of myObject is in myParentList then
                        if (count graphics of myObject) > 0 then
                            copy graphic 1 of myObject to end of myObjectList
                        end if
                    end if
                end repeat
                if (count myObjectList) is not equal to 0 then
                    --The selection contains a qualifying item or items.
                    --Add the menu item if it does not already exist.
                    if my myMenuItemExists(myLayoutContextMenu,
                    "Create Graphic Label") is false then
                        my myMakeLabelGraphicMenuItem(myLayoutContextMenu)
                    end if
                else
                    --Remove the menu item if it exists.
                    if my myMenuItemExists(myLayoutContextMenu,
                    "Create Graphic Label") is true then
                        tell myLayoutContextMenu
                            delete menu item "Create Graphic Label"
                        end tell
                    end if
                end if
            end if
        end if
    end tell
end myBeforeDisplayHandler
```

```
on myMakeLabelGraphicMenuItem(myLayoutContextMenu)
    tell application "Adobe InDesign CS5"
        if my myScriptMenuActionExists("Create Graphic Label") is false then
            set myLabelGraphicMenuAction to make script menu action with
            properties {name:"Create Graphic Label"}
            tell myLabelGraphicMenuAction
                set myLabelGraphicEventListener to make event listener
                with properties {event type:"onInvoke",handler:
                "yukino:IDEventHandlers:LayoutMenuOnInvoke.applescript",
                captures:false}
            end tell
        else
            set myLabelGraphicMenuAction to script menu action
            "Create Graphic Label"
        end if
        tell myLayoutContextMenu
            set myLabelGraphicMenuItem to make menu item with properties
            {associated menu action:myLabelGraphicMenuAction}
        end tell
    end tell
end myMakeLabelGraphicMenuItem
```

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

```
LabelGraphicEventHandler(evt)
on LabelGraphicEventHandler(myEvent)
    tell application "Adobe InDesign CS5"
        set myGraphicList to {PDF, EPS, image}
        set myParentList to {rectangle, oval, polygon}
        set myObjectList to {}
        set myLayoutContextMenu to menu "$ID/RtMouseLayout"
        if (count documents) > 0 then
            set mySelection to selection
            if (count mySelection) > 0 then
                repeat with myCounter from 1 to (count mySelection)
                    set myObject to item myCounter of mySelection
                    if class of myObject is in myGraphicList then
                        copy myObject to end of myObjectList
                    else if class of myObject is in myParentList then
                        if (count graphics of myObject) > 0 then
                            copy graphic 1 of myObject to end of myObjectList
                        end if
                    end if
                end repeat
                if (count myObjectList) is not equal to 0 then
                    --The selection contains a qualifying item or items.
                    my myDisplayDialog(myObjectList)
                end if
            end if
        end if
    end tell
end LabelGraphicEventHandler
--Displays a dialog box containing label options.
on myDisplayDialog(myObjectList)
    tell application "Adobe InDesign CS5"
        set myLabelWidth to 100
        set myStyleNames to my myGetParagraphStyleNames(document 1)
        set myLayerNames to my myGetLayerNames(document 1)
        set myDialog to make dialog with properties {name:"LabelGraphics"}
```

```
tell myDialog
    tell (make dialog column)
        --Label Type
        tell (make dialog row)
            tell (make dialog column)
                make static text with properties
                {static label:"Label Type:", min width:myLabelWidth}
            end tell
            tell (make dialog column)
                set myLabelTypeDropdown to make dropdown with properties
                {string list:{"File Name", "File Path", "XMP Author",
                "XMP Description"}, selected index:0}
            end tell
        end tell
        --Text frame height
        tell (make dialog row)
            tell (make dialog column)
                make static text with properties
                {static label:"Label Height:", min width:myLabelWidth}
            end tell
            tell (make dialog column)
                set myLabelHeightField to make measurement editbox
                with properties {edit value:24, edit units:points}
            end tell
        end tell
        --Text frame offset
        tell (make dialog row)
            tell (make dialog column)
                make static text with properties
                {static label:"Label Offset:", min width:myLabelWidth}
            end tell
            tell (make dialog column)
                set myLabelOffsetField to make measurement editbox with
                properties {edit value:0, edit units:points}
            end tell
        end tell
        --Paragraph style to apply
        tell (make dialog row)
            tell (make dialog column)
                make static text with properties
                {static label:"Label Style:", min width:myLabelWidth}
            end tell
            tell (make dialog column)
                set myLabelStyleDropdown to make dropdown with properties
                {string list:myStyleNames, selected index:0}
            end tell
        end tell
        --Layer
        tell (make dialog row)
            tell (make dialog column)
                make static text with properties
                {static label:"Layer:", min width:myLabelWidth}
            end tell
            tell (make dialog column)
                set myLayerDropdown to make dropdown with properties
                {string list:myLayerNames, selected index:0}
            end tell
        end tell
    end tell
end tell
```

```
        set myResult to show myDialog
        if myResult is true then
            set myLabelType to (selected index of myLabelTypeDropdown)
            set myLabelHeight to edit value of myLabelHeightField
            set myLabelOffset to edit value of myLabelOffsetField
            set myLabelStyleName to item ((selected index of
            myLabelStyleDropdown)+ 1) of myStyleNames
            set myLayerName to item ((selected index of myLayerDropdown) + 1)
            of myLayerNames
            destroy myDialog
            tell view preferences of document 1
                set myOldXUnits to horizontal measurement units
                set myOldYUnits to vertical measurement units
                set horizontal measurement units to points
                set vertical measurement units to points
            end tell
            repeat with myCounter from 1 to (count myObjectList)
                set myGraphic to item myCounter of myObjectList
                my myAddLabel(myGraphic, myLabelType, myLabelHeight,
                myLabelOffset, myLabelStyleName, myLayerName)
            end repeat
            tell view preferences of document 1
                set horizontal measurement units to myOldXUnits
                set vertical measurement units to myOldYUnits
            end tell
        else
            destroy myDialog
        end if
    end tell
end myDisplayDialog
on myAddLabel(myGraphic, myLabelType, myLabelHeight, myLabelOffset, myLabelStyleName,
myLayerName)
    tell application "Adobe InDesign CS5"
        set myDocument to document 1
        set myLabelStyle to paragraph style myLabelStyleName of myDocument
        try
            set myLabelLayer to layer myLayerName of myDocument
        on error
            tell myDocument
                set myLabelLayer to make layer with properties {name:myLayerName}
            end tell
        end try
        set myLink to item link of myGraphic
        if myLabelType is 0 then
            set myLabel to name of myLink
        else if myLabelType is 1 then
            set myLabel to file path of myLink
        else if myLabelType is 2 then
            try
                set myLabel to author of link xmp of myLink
            on error
                set myLabel to "No author available."
            end try
        else if myLabelType is 3 then
            try
                set myLabel to description of link xmp of myLink
            on error
                set myLabel to "No description available."
            end try
        end if
```

```
        set myFrame to parent of myGraphic
        set myBounds to geometric bounds of myFrame
        set myX1 to item 2 of myBounds
        set myY1 to (item 3 of myBounds) + myLabelOffset
        set myX2 to item 4 of myBounds
        set myY2 to myY1 + myLabelHeight
        tell parent of myFrame
            set myTextFrame to make text frame with properties
            {item layer:myLabelLayer, contents:myLabel,
            geometric bounds:{myY1, myX1, myY2, myX2}}
            set first baseline offset of text frame preferences of
            myTextFrame to leading offset
            tell myTextFrame
                set applied paragraph style of paragraph 1 to myLabelStyle
            end tell
        end tell
    end tell
end myAddLabel
```

10 Working with Preflight

Preflight is a way to verify that you have all required files, fonts, assets (e.g., placed images and PDF files), printer settings, trapping styles, etc., before you send a publication to an output device. For example, if you placed an image as a low-resolution proxy but do not have the high-resolution original image accessible on your hard disk (or workgroup server), that may result in an error during the printing process. Preflight checks for this sort of problem. It can be run in the background as you work.

This chapter demonstrates how to interact with the preflight system using scripting. For illustration purposes, we show how to configure preflight to raise an error if the page size is something other than letter size (8.5" x 11"). We briefly highlight how it is done in the user interface, then show how to achieve the same results through scripting.

We assume you already read *Adobe InDesign CS5 Scripting Tutorial* and know how to create, install, and run a script.

Exploring Preflight Profiles

InDesign's preflight feature is profile based, rule driven, and parameterized. There might be one or more preflight profiles. Initially, there is one profile, [Basic], which is read-only; you cannot modify or delete it.

A preflight profile contains many preflight rules. Each rule has a name and multiple data objects. Each data object has a name, data type, and data value. The data value can be changed. Each rule can be configured as follows:

- Disabled — The preflight rule is disabled.
- Return as error — The preflight rule returns error-level feedback.
- Return as warning — The preflight rule returns warning-level feedback.
- Return as informational — The preflight rule returns informational-level feedback.

To check the profile in InDesign, choose Preflight Panel > Define Profiles. You also can get profile information by scripting.

Listing preflight profiles

This script fragment shows how to list all preflight profiles. For the complete script, see ListPreflightProfiles.

```
tell application "Adobe InDesign CS5"
    set myProfiles to preflight profiles
    set myProfileCount to count of myProfiles
    set myStr to "Preflight profiles: "
    repeat with i from 1 to myProfileCount
        if i > 1 then
            set myStr to myStr & ", "
        end if
        set myStr to myStr & name of item i of myProfiles
    end repeat
    display alert myStr
end tell
```

Listing preflight rules

This script fragment shows how to list all preflight rules in a profile. For the complete script, see ListPreflightRules.

```
tell application "Adobe InDesign CS5"
    -- Assume the [Basic] profile exists.
    set myProfile to item 1 of preflight profiles
    set myRules to preflight profile rules of myProfile
    set ruleCount to count of myRules
    set str to "Preflight rules of " & name of myProfile & ": "
    repeat with i from 1 to ruleCount
        if i > 1 then
            set str to str & ", "
        end if
        set str to str & name of item i of myRules
    end repeat
    display alert str
end tell
```

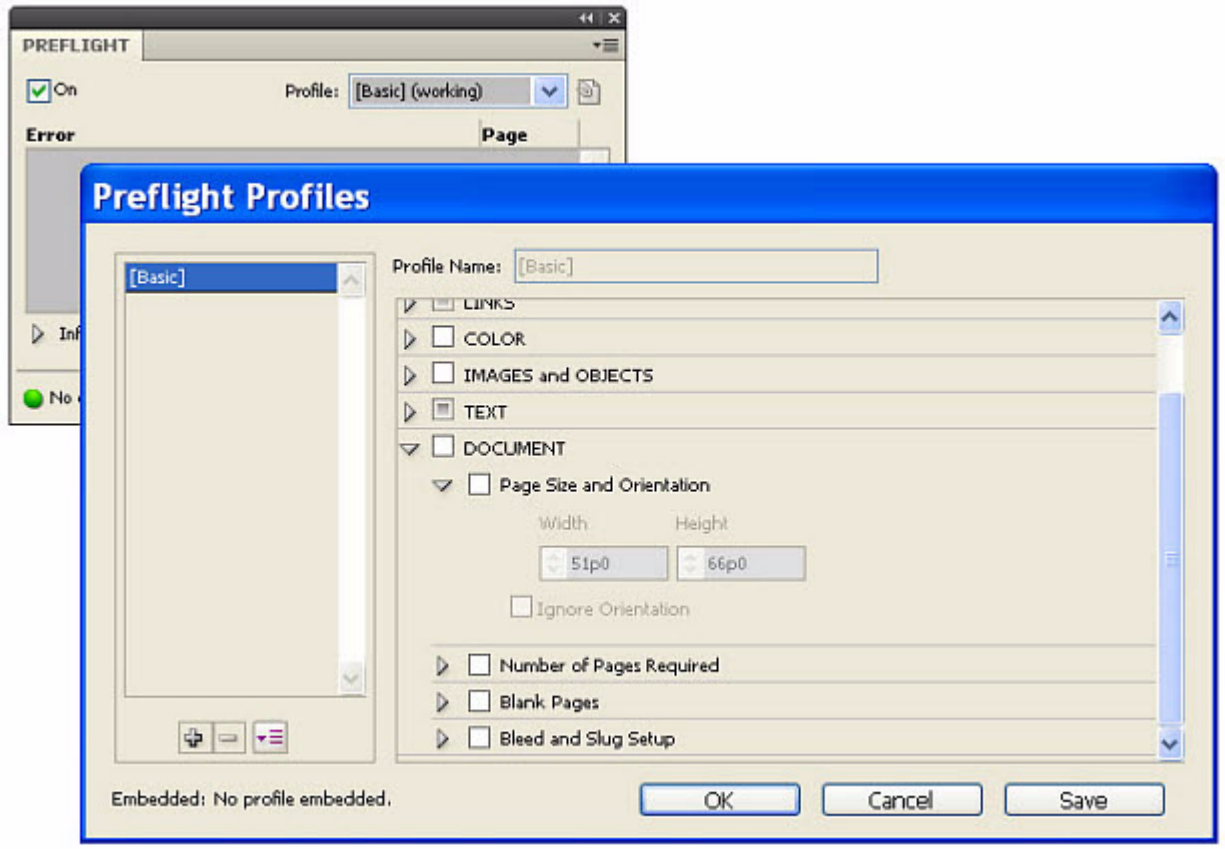
Listing preflight data objects

This script fragment shows how to list all preflight data objects in a profile rule. For the complete script, see ListPreflightDataObjects.

```
tell application "Adobe InDesign CS5"
    set myProfile to item 1 of preflight profiles
    set myRule to item 1 of preflight profile rules of myProfile
    set dataObjects to rule data objects of myRule
    set dataObjectCount to count of dataObjects
    set str to"Preflight rule data objects of " & name of myProfile &"." &name of myRule
&": "
    repeat with i from 1 to dataObjectCount
        if i > 1 then
            set str to str & "; "
        end if
        set myObject to item i of dataObjects
        set str to str & name of myObject & ", "
        set str to str & data type of myObject & ", "
        set str to str & data value of myObject
    end repeat
    display alert str
end tell
```

Importing a Preflight Profile

To import a preflight profile from the Preflight panel, choose Preflight Panel > Define Profiles, then choose Load Profile from the drop-down menu in the Preflight Profiles window.



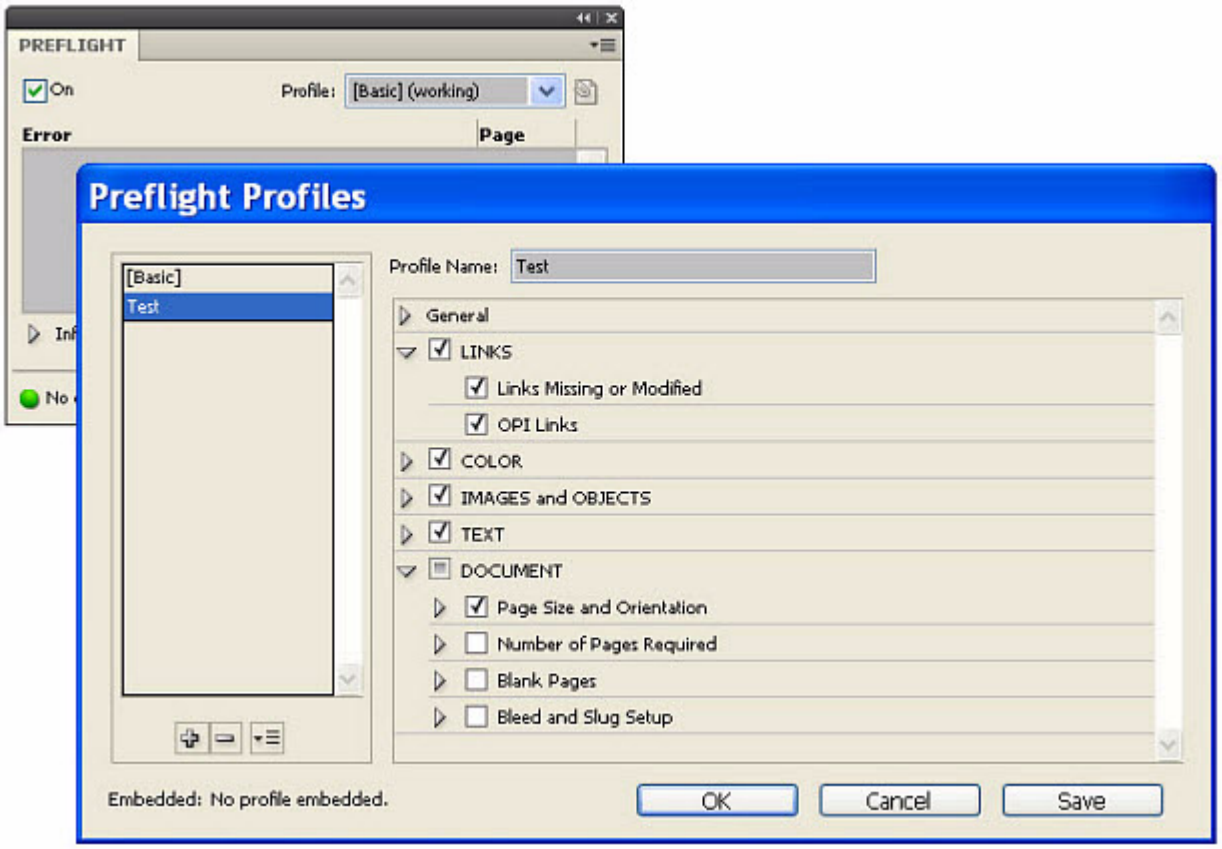
You also can load a profile with scripting. The following script fragment imports a profile called Test. For the complete script, see ImportPreflightProfile.

```
set myProfile to load preflight profile from "Macintosh HD:tmp:Test.idpp"
if myProfile is nothing then
    display alert "The profile did not load successfully"
else
    display alert "Preflight profile " & (name of myProfile) & " is loaded."
end if
```

It is easier to create profiles using the Preflight panel than with scripting. One workflow would be to create all profiles in the user interface, export them to files, and import them using scripting. This approach avoids the challenges involved with manually adding rules via scripting.

Creating a Preflight Profile

To create a preflight profile from the Preflight panel, choose Preflight Panel > Define Profiles, then choose the plus sign (+) to add a new preflight profile. Name the profile and fill in all data values for the available rules.



You also can create a profile with scripting. The following script fragment adds a single profile called Test. For the complete script, see CreatePreflightProfile.

```
--Add a new preflight profile.
set myProfile to make preflight profile
display alert "Preflight profile " & (name of myProfile) & " is created."
--Rename the profile
tell myProfile
    set oldName to name
    set name to "Test"
    set description to "Test description"
end tell
display alert "Profile " & oldName & " is renamed to " & (name of myProfile) & "."
```

Preflight-profile names must be unique. If the script above is executed more than once within the same InDesign instance, an error is raised, indicating that a profile with that name already exists. To avoid this, either access the existing profile using `app.preflightProfiles.itemByName()`, or check to see if a profile exists and remove it; see the following script fragment. For the complete script, see DeletePreflightProfile.

```
on removeProfile(profileName)
    tell application "Adobe InDesign CS5"
        set myProfiles to preflight profiles
        set profileCount to count of myProfiles
        repeat with i from 1 to profileCount
            if name of item i of myProfiles is equal to profileName then
                delete item i of myProfiles
            end if
        end repeat
    end tell
end removeProfile
```

Adding Rules

A preflight profile contains a mutually exclusive set of rules. To add a rule to a profile, follow these steps:

1. Add a rule to a profile by name.

   Rules are added by name. For information on rule names, see "Available Rules" on page 156. The following adds the ADBE_PageSizeOrientation rule to the profile.

   ```
   --Add a rule that requires a specific page size and orientation
   --(portrait or landscape).
   set RULE_NAME to "ADBE_PageSizeOrientation"
   set myRule to make preflight profile rule in myProfile with properties
   {name:RULE_NAME, id:RULE_NAME}
   ```

2. Set the rule's data values.

   Many, but not all, rules have data properties. For a complete specification of the rules available with InDesign, see "Available Rules" on page 156. The ADBE_PageSizeOrientation rule contains particular data properties that allow you to specify a page size. The following sets the acceptable page height and width, a tolerance (fudge factor), and an option for handling page orientation.

   ```
   tell myRule
       --Requires the page size to be 8.5in x 11in (Letter Size)
       --enters a value for tolerance
       make rule data object with properties {name:"tolerance", data type:real data
   type, data value:0.01}
       --Sets the width  to the point equivalent of 8.5 inches
       make rule data object with properties {name:"width", data type:real data type,
   data value:612}
       --Sets the width  to the point equivalent of 11 inches
       make rule data object with properties {name:"height", data type:real data type,
   data value:792}
       --true = ignore orientation is checked
       make rule data object with properties {name:"ignore_orientation", data
   type:boolean data type, data value:true}
   end tell
   ```

3. Set the rule's reporting state.

   This is done using the rule's `flag` property. There are several choices (disabled, information, warning, and error), controlled by the `PreflightRuleFlag` enumeration.

```
--set the rule to return an error
set flag of myRule to return as error
```

Processing a Profile

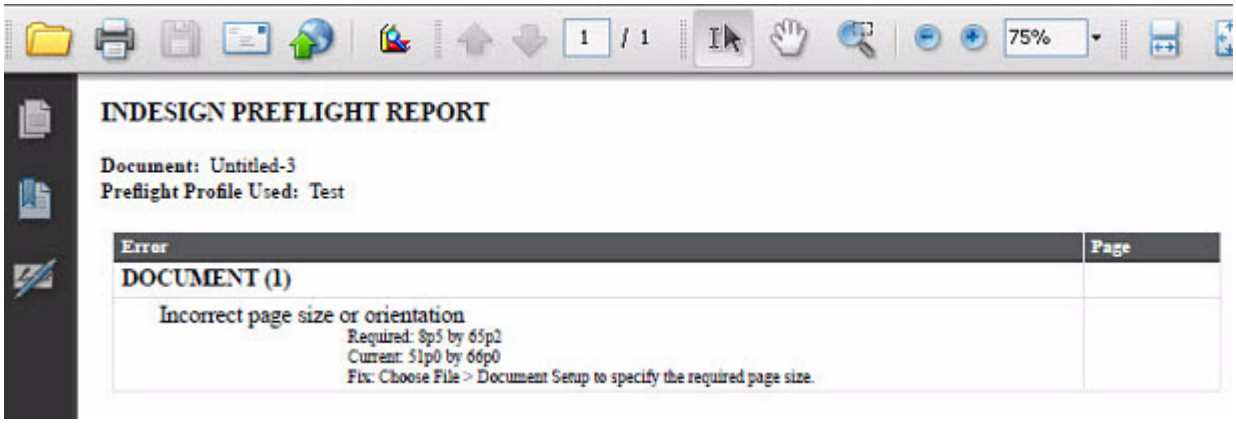
In the desktop version of InDesign, preflight errors are reported in the user interface. In scripting (especially for InDesign Server), the errors are generated on demand. The following script processes an InDesign document. (For the complete script, see ProcessPreflightProfile.) If there are errors, it writes the results to a new PDF file. For an example of the output, see the figure below the script.

```
--Assume there is an document.
set myDoc to document 1
--Use the second preflight profile
set myProfile to preflight profile 2

--Process the doc with the profile
set myProcess to make preflight process with properties {target object:myDoc, applied
profile:myProfile}
wait for process myProcess
set results to process results of myProcess

-- If errors were found
if results is not "None" then
    -- Export the file to PDF, and to open the file after export.
    save report myProcess to "Macintosh HD:tmp:PreflightResults.pdf" with auto open
end if

--Cleanup
delete myProcess
```



If you would rather produce a text file, simply name your output file with a `.txt` extension.

Alternately, you may prefer to iterate the errors yourself. The following demonstrates how to access the errors array. For the complete script, see ProcessPreflightProfileShowErrors.

```
-- If errors were found
if results is not "None" then
    --array containing detailed results
    tell aggregated results of myProcess
        set str to "Document: " & item 1 & ", Profile: " & item 2 & ", Results: ["
        set errorResults to item 3
    end tell
    --Show the errors in a message box.
    repeat with i from 1 to count of errorResults
        if i > 1 then
            str = str & ", "
        end if
        set str to str & item 2 of item i of errorResults
    end repeat
    set str to str & "]"
    display alert str
end if
```

Custom Rules

It is not possible to create custom rules through the Preflight panel or scripting; however, this can be done through a C++ plug-in. The InDesign Products SDK contains a sample, PreflightRule, that demonstrates how to add custom rules with a plug-in.

Available Rules

One of the hardest aspects of scripting rules is discovering rule names and properties. Due to the dynamic nature of rules (they really are just strings), specific rule names and properties do not appear in the Extend Script Tool Kit's Object Model Viewer. To discover this information, see "Exploring Preflight Profiles" on page 150. For your convenience, the `DumpPreflightRules.jsx` script is provided in the SDK to produce the following output as an HTML file (*SDK*\docs\references\PreflightRules.html). If you use a plug-in that adds custom rules, you can run the script to extract the new names and properties.

Rule name	Rule properties
ADBE_BlankPages	"ADBE_BlankPages" on page 157
ADBE_BleedSlug	"ADBE_BleedSlug" on page 157
ADBE_BleedTrimHazard	"ADBE_BleedTrimHazard" on page 158
ADBE_CMYPlates	no
ADBE_Colorspace	"ADBE_Colorspace" on page 159
ADBE_ConditionIndicators	no
ADBE_CrossReferences	"ADBE_CrossReferences" on page 159yes
ADBE_FontUsage	"ADBE_FontUsage" on page 159
ADBE_ImageColorManagement	"ADBE_ImageColorManagement" on page 160
ADBE_ImageResolution	"ADBE_ImageResolution" on page 160
ADBE_InteractiveContent	no

Rule name	Rule properties
ADBE_LayerVisibility	no
ADBE_MissingFonts	no
ADBE_MissingGlyph	no
ADBE_MissingModifiedGraphics	no
ADBE_OPI	no
ADBE_Overprint	no
ADBE_OversetText	no
ADBE_PageCount	"ADBE_PageCount" on page 160
ADBE_PageSizeOrientation	"ADBE_PageSizeOrientation" on page 161
ADBE_Registration	no
ADBE_ScaledGraphics	"ADBE_ScaledGraphics" on page 161
ADBE_ScaledType	"ADBE_ScaledType" on page 161
ADBE_SmallText	"ADBE_SmallText" on page 161
ADBE_SpellCheck	no
ADBE_SpotColorSetup	"ADBE_SpotColorSetup" on page 161
ADBE_StrokeRequirements	"ADBE_StrokeRequirements" on page 162
ADBE_TextOverrides	"ADBE_TextOverrides" on page 162
ADBE_TransparencyBlending	"ADBE_TransparencyBlending" on page 162
ADBE_TransparencyUsage	no
ADBE_WhiteOverprint	no

ADBE_BlankPages

Data Type	Name	Default value
Boolean	ignore_master	true
Boolean	ignore_nonprinting	true

ADBE_BleedSlug

Data Type	Name	Default value
Real	bleed_b	9
Real	bleed_b_aux	9

Data Type	Name	Default value
Integer	bleed_comparison_type	3
Boolean	bleed_enabled	true
Real	bleed_l	9
Real	bleed_l_aux	9
Real	bleed_r	9
Real	bleed_r_aux	9
Real	bleed_t	9
Real	bleed_t_aux	9
Real	slug_b	18
Real	slug_b_aux	18
Integer	slug_comparison_type	3
Boolean	slug_enabled	false
Real	slug_l	18
Real	slug_l_aux	18
Real	slug_r	18
Real	slug_r_aux	18
Real	slug_t	18
Real	slug_t_aux	18
Real	tolerance	0.01

ADBE_BleedTrimHazard

Data Type	Name	Default value
Boolean	binding_enabled	false
Real	binding_width	1
Real	live_b	18
Real	live_l	18
Real	live_r	18
Real	live_t	18
Real	tolerance	0.01

ADBE_Colorspace

Data Type	Name	Default value
Boolean	no_cmyk	false
Boolean	no_gray	false
Boolean	no_lab	false
Boolean	no_rgb	false
Boolean	no_spot	false

ADBE_CrossReferences

Data Type	Name	Default value
Boolean	xrefs_out_of_date	true
Boolean	xrefs_unresolved	true

ADBE_FontUsage

Data Type	Name	Default value
Boolean	no_ATC	false
Boolean	no_Bitmap	false
Boolean	no_CID	false
Boolean	no_MultipleMaster	false
Boolean	no_OpenTypeCFF	false
Boolean	no_OpenTypeCID	false
Boolean	no_OpenTypeTT	false
Boolean	no_TrueType	false
Boolean	no_Type1	false
Boolean	no_protected	true

ADBE_ImageColorManagement

Data Type	Name	Default value
Boolean	no_cmyk_profiles	true
Boolean	no_image_overrides	true
Boolean	overrides_exclude_uncal	true

ADBE_ImageResolution

Data Type	Name	Default value
Boolean	bw_max_enabled	false
Real	bw_max_res	2400
Boolean	bw_min_enabled	true
Real	bw_min_res	800
Boolean	color_max_enabled	false
Real	color_max_res	1200
Boolean	color_min_enabled	true
Real	color_min_res	250
Boolean	gray_max_enabled	false
Real	gray_max_res	1200
Boolean	gray_min_enabled	true
Real	gray_min_res	250
Real	tolerance	0.5

ADBE_PageCount

Data Type	Name	Default value
Integer	comparison_type	2
Integer	comparison_value	1
Integer	comparison_value_aux	1

ADBE_PageSizeOrientation

Data Type	Name	Default value
Real	height	792
Boolean	ignore_orientation	false
Real	tolerance	0.01
Real	width	612

ADBE_ScaledGraphics

Data Type	Name	Default value
Real	max_scale	100.5

ADBE_ScaledType

Data Type	Name	Default value
Boolean	ignore_justification	true
Real	max_scale	100.5

ADBE_SmallText

Data Type	Name	Default value
Real	minSize	4
Boolean	minSize_trap_safe_only	false

ADBE_SpotColorSetup

Data Type	Name	Default value
Boolean	lab_spots	true
Boolean	lab_spots_enabled	false
Integer	max_spots	1
Boolean	max_spots_enabled	true

ADBE_StrokeRequirements

Data Type	Name	Default value
Real	min_width	0.125
Boolean	min_width_trap_safe_only	false

ADBE_TextOverrides

Data Type	Name	Default value
Boolean	ignore_color_overrides	false
Boolean	ignore_font_overrides	false
Boolean	ignore_kerning_tracking_overrides	false
Boolean	ignore_language_overrides	false

ADBE_TransparencyBlending

Data Type	Name	Default value
Integer	space	3

11 Creating Dynamic Documents

InDesign can create documents for web and online use, also known as Rich Interactive Documents (RID). Dynamic documents contain sounds, animations, hyperlinks, and other interactive content. InDesign documents can be exported to SWF, XFL, or PDF. For SWF and XFL files, documents can include animations, buttons, multistate objects, movies, and sound clips. You can use the Preview panel in InDesign to test some types of dynamic content before exporting.

This chapter shows how to create dynamic documents using scripting. For more on exporting as PDF, SWF, and XFL, refer to the "Working with Documents" chapter.

Importing Movies and Sounds

InDesign can import movie and sound files that can then be viewed or listened to in exported PDF, SWF, or XFL documents. Movies and sounds in an InDesign document are very similar to graphics in that they exist inside container objects on an InDesign page. Unlike graphics, however, you cannot see (or hear) the content of the imported multimedia files on an InDesign page. For that, you'll need to either view the page in the Preview panel, or export the file, then open the file in a viewer capable of displaying the content (such as Acrobat Reader or a web browser).

Scripts can control the playback properties of a sound or movie in an exported dynamic document. You can also add a preview, or "poster," image to the page item containing the sound or movie.

The following script fragment shows how to import a movie and control the way that the movie is shown and played in an exported document (for the complete script, refer to PlaceMovie).

```
--Given a page "myPage"...
tell myPage
    set myFrame to make rectangle with properties {geometric bounds:{72, 72, 288, 288}}
end tell
--Import a movie file (you'll have to provide a valid file path on your system)
tell myFrame to place file "hazuki:movie.flv"
set myMovie to movie 1 of myFrame
--Set movie properties.
set embed in PDF of myMovie to true
set show controls of myMovie to true
get properties of myMovie
--Add a preview image. You'll have to provide a valid path on your system.
set poster file of myMovie to "hazuki:movie poster.jpg"
```

The following script fragment shows how to import a sound file and control the playback and display of the sound in an exported document (for the complete script, refer to PlaceSound).

```
--Given a page "myPage" in a document "myDocument..."
--Import a sound file (you'll have to provide a valid file path on your system)
tell myPage
    set mySound to place file "hazuki:sound.mp3" place point {72, 72}
end tell
set mySound to item 1 of mySound
tell mySound
    set embed in PDF to true
    set do not print poster to true
    set sound loop to true
    set stop on page turn to true
end tell
--Add a preview image. You'll have to provide a valid path on your system.
set poster file of mySound to "hazuki:sound poster.jpg"
```

Buttons can be used to control the playback of sounds and movies. For information on how to script buttons, see the next section.

Creating Buttons

Buttons are often used for navigation in dynamic documents. Buttons contain three states, known as "Normal," "Rollover," and "Click," which, in turn, can contain page items such as rectangles, ovals, text frames, or images. The button can display only one state at a time; the other states are displayed when triggered by mouse actions.

Behaviors control what the button does when you perform a specific mouse action. Behaviors correspond to the Actions shown in the Buttons panel in InDesign's user interface. Buttons can contain multiple behaviors.

The following script fragment shows how to create a simple button that displays the next page in an exported PDF or SWF (for the complete script, refer to SimpleButton). This button makes use of only the Normal state.

```
--Given a page "myPage" and a document containing the color "Red"...
--Make a button by converting a page item.
tell myPage
    set myRightArrow to make polygon with properties {fill color:"Red",
    name:"GoToNextPageButton"}
    set entire path of path 1 of myRightArrow to {{72, 72}, {144, 108}, {72, 144}}
    set myButton to make button with properties {geometric bounds:{72, 72, 144, 144}}
end tell
tell state 1 of myButton
    add items to state pageitems {myRightArrow}
end tell
tell myButton
    set myGoToNextPageBehavior to make goto next page behavior
    with properties {behavior event:mouse up}
end tell
```

The following script fragment shows how to create a somewhat more complicated button, containing page items that change the appearance of each of the three button states. For the complete script, refer to ButtonStates.

```
--Given a page "myPage" in a document "myDocument," containing the colors
--"Blue" and "Red"...
--Make a button "from scratch."
tell page 1 of myDocument
    set myButton to make button with properties {geometric bounds:{72, 72, 144, 144},
    name:"GoToNextPageButton"}
end tell
tell state 1 of myButton
    set myRightArrow to make polygon with properties {fill color:color "Red" of
    myDocument, stroke color:"None"}
    set entire path of path 1 of myRightArrow to {{72, 72}, {144, 108}, {72, 144}}
end tell
--Add the Rollover state.
tell myButton
    set myRolloverState to make state
end tell
tell myRolloverState
    set myRolloverArrow to make polygon with properties {fill color:color "Red"
    of myDocument, stroke color:"None"}
    set entire path of path 1 of myRolloverArrow to {{72, 72}, {144, 108}, {72, 144}}
    --Add a shadow to the polygon in the Rollover state.
end tell
tell drop shadow settings of fill transparency settings of myRolloverArrow
    set mode to drop
    set angle to 90
    set x offset to 0
    set y offset to 0
    set size to 6
end tell
tell myButton
    set myClickState to make state
end tell
tell myClickState
    set myClickArrow to make polygon with properties {fill color:color "Blue"
    of myDocument, stroke color:"None"}
    set entire path of path 1 of myClickArrow to {{72, 72}, {144, 108}, {72, 144}}
end tell
--Set the behavior for the button.
tell myButton
    set myGoToNextPageBehavior to make goto next page behavior
    with properties {behavior event:mouse up}
end tell
```

Buttons can be used to control the playback of movie and sound files. The following script fragment shows an example of using a set of buttons to control the playback of a moving file (for the complete script, refer to MovieControl).

```
--Given a page "myPage" in a document "myDocument,"
--containing the colors "Gray" and "Red"...
tell myPage
    set myFrame to make rectangle with properties {geometric bounds:{72, 72, 288, 288}}
    --Import a movie file (you'll have to provide a valid file path on your system)
    tell myFrame to place file "hazuki:movie.flv"
    --Create the movie "Start" button.
    set myPlayButton to make button with properties {geometric bounds:
    {294, 186, 354, 282}, name:"PlayMovieButton"}
    tell myPlayButton
        set myRightArrow to make polygon with properties {fill color:color "Gray"
        of myDocument, stroke color:"None"}
    end tell
    set entire path of path 1 of myRightArrow to {{186, 294}, {186, 354}, {282, 324}}
    --Add the Rollover state.
    tell myPlayButton
        set myRolloverState to make state
    end tell
    --Add a shadow to the polygon in the Rollover state.
    tell myRolloverState
        set myRolloverArrow to make polygon with properties {fill color:color "Gray"
        of myDocument, stroke color:"None"}
    end tell
    set entire path of path 1 of myRolloverArrow to {{186, 294}, {186, 354}, {282, 324}}
    tell drop shadow settings of fill transparency settings of myRolloverArrow
        set mode to drop
        set angle to 90
        set x offset to 0
        set y offset to 0
        set size to 6
    end tell
    tell myPlayButton
        set myClickState to make state
    end tell
    tell myClickState
        set myClickArrow to make polygon with properties {fill color:color "Red"
        of myDocument, stroke color:"None"}
    end tell
    set entire path of path 1 of myClickArrow to {{186, 294}, {186, 354}, {282, 324}}
    --Set the behavior for the button.
    tell myPlayButton
        set myMovieStartBehavior to make movie behavior with properties {movie item:
        movie 1 of myFrame, behavior event:mouse up, operation:play}
    end tell
    --Create the movie "Stop" button.
    set myStopButton to make button with properties {geometric bounds:
    {294, 78, 354, 174}, name:"StopMovieButton"}
    tell state 1 of myStopButton
        set myNormalRectangle to make rectangle with properties {geometric bounds:
        {294, 78, 354, 174}, fill color:color "Gray" of myDocument}
    end tell
    tell myStopButton
        set myRolloverState to make state
    end tell
    tell myRolloverState
        set myRolloverRectangle to make rectangle with properties {geometric bounds:
        {294, 78, 354, 174}, fill color:color "Gray" of myDocument}
    end tell
    tell drop shadow settings of fill transparency settings of myRolloverRectangle
        set mode to drop
```

```
        set angle to 90
        set x offset to 0
        set y offset to 0
        set size to 6
    end tell
    tell myStopButton
        set myClickState to make state
    end tell
    tell myClickState
        set myClickRectangle to make rectangle with properties {geometric bounds:
        {294, 78, 354, 174}, fill color:color "Red" of myDocument}
    end tell
    tell myStopButton
        set myMovieStopBehavior to make movie behavior with properties {movie item:
        movie 1 of myFrame, behavior event:mouse up, operation:stop}
    end tell
end tell
```

Buttons are also important in controlling the appearance of multistate objects, as we'll demonstrate in the next section.

Creating Multistate Objects

Multistate objects (or MSOs) are similar to buttons in that they contains states, and that only one state can be visible at a time. They are unlike buttons in that they can contain any number of states; buttons can contain three states, at most. Multistate objects rely on buttons to change the way they display their states.

The following script fragment shows how to create a simple multistate object and add a button to control the display of the states in the object (for the complete script, refer to MakeMultiStateObject).

```
--Given a document "myDocument" and a page "myPage" and
--four colors "myColorA," "myColorB," "myColorC," and "myColorD"...
tell myPage
    set myMSO to make multi state object with properties {name:"Spinner",
    geometric bounds:{72, 72, 144, 144}}
    --New multistate objects contain two states when they're created. Add two more.
    tell myMSO
        set name of state 1 to "Up"
        set name of state 2 to "Right"
        make state with properties {name:"Down"}
        make state with properties {name:"Left"}
    end tell
    --Add page items to the states.
    tell state 1 of myMSO
        set myPolygon to make polygon with properties {fill color:myColorA,
        stroke color:"None"}
        set entire path of path 1 of myPolygon to {{72, 144}, {144, 144}, {108, 72}}
    end tell
    tell state 2 of myMSO
```

```
        set myPolygon to make polygon with properties {fill color:myColorB,
        stroke color:"None"}
        set entire path of path 1 of myPolygon to {{72, 72}, {72, 144}, {144, 108}}
    end tell
    tell state 3 of myMSO
        set myPolygon to make polygon with properties {fill color:myColorC,
        stroke color:"None"}
        set entire path of path 1 of myPolygon to {{72, 72}, {108, 144}, {144, 72}}
    end tell
    tell state 4 of myMSO
        set myPolygon to make polygon with properties {fill color:myColorD,
        stroke color:"None"}
        set entire path of path 1 of myPolygon to {{144, 72}, {72, 108}, {144, 144}}
    end tell
end tell
```

Typically, you'll control the display of the states in a multistate object using a button. The following script fragment shows how to do this (for the complete script, refer to MultiStateObjectControl).

```
--Given a document "myDocument" and a page "myPage" and
--four colors "myColorA," "myColorB," "myColorC," and "myColorD"...
tell myPage
    set myMSO to make multi state object with properties {name:"Spinner",
    geometric bounds:{72, 72, 144, 144}}
    --New multistate objects contain two states when they're created. Add two more.
    tell myMSO
        set name of state 1 to "Up"
        set name of state 2 to "Right"
        make state with properties {name:"Down"}
        make state with properties {name:"Left"}
    end tell
    --Add page items to the states.
    tell state 1 of myMSO
        set myPolygon to make polygon with properties {fill color:myColorA,
        stroke color:"None"}
        set entire path of path 1 of myPolygon to {{72, 144}, {144, 144}, {108, 72}}
    end tell
    tell state 2 of myMSO
        set myPolygon to make polygon with properties {fill color:myColorB,
        stroke color:"None"}
        set entire path of path 1 of myPolygon to {{72, 72}, {72, 144}, {144, 108}}
    end tell
    tell state 3 of myMSO
        set myPolygon to make polygon with properties {fill color:myColorC,
        stroke color:"None"}
        set entire path of path 1 of myPolygon to {{72, 72}, {108, 144}, {144, 72}}
    end tell
    tell state 4 of myMSO
        set myPolygon to make polygon with properties {fill color:myColorD,
        stroke color:"None"}
        set entire path of path 1 of myPolygon to {{144, 72}, {72, 108}, {144, 144}}
    end tell
    set myButton to make button with properties {geometric bounds:{72, 72, 144, 144}}
    tell myButton
        set myRolloverState to make state
        set myClickState to make state
        set myNextStateBehavior to make goto next state behavior with
        properties {associated multi state object:myMSO, behavior event:mouse down,
        enable behavior:true, loops to next or previous:true}
    end tell
```

```
    tell myRolloverState
        set myRolloverRectangle to rectangle 1 of group 1
    end tell
    set stroke color of myRolloverRectangle to myColorD
    set stroke weight of myRolloverRectangle to 1
    set myStrokeTransparencySettings to stroke transparency settings of
    myRolloverRectangle
    set myDropShadowSettings to drop shadow settings of myStrokeTransparencySettings
    tell myDropShadowSettings
        set mode to drop
        set angle to 90
        set x offset to 0
        set y offset to 0
        set size to 6
    end tell
end tell
```

Working with Animation

Page items can be animated, adding motion to the dynamic documents you create using InDesign. You apply animation to objects using motion presets, define the movement of animated objects using motion paths, and control the duration of the animation using timing settings, timing lists, and timing groups.

The `animation settings` of an object control the animation that will be applied to the object. When animation settings have been applied to an object, InDesign sets the `has custom settings` property of the object to true; if the object is not to be animated, this property is false.

The point at which an animation begins to play, relative to the event that triggers the animation, is controlled by the objects and properties of the `timing settings` object attached to the page item or to one of its parent containers (usually the spread).

Basic animation

The following script fragment shows how to create a simple animation (for the complete script, refer to SimpleAnimation). The most basic forms of animation can be applied without using timing settings.

```
--Given a document "myDocument" and a page "myPage" and a color "myColorA"...
--Add a page items to animate.
tell myPage
    set myPolygon to make polygon with properties {fill color:myColorA,
    stroke color:"None"}
end tell
set entire path of path 1 of myPolygon to {{72, 72}, {72, 144}, {144, 108}}
--Create a motion path.
set myMotionPathPoints to {{{{108, 108}, {108, 108}, {108, 108}}, {{516, 108},
{516, 108}, {516, 108}}}, true}
--Set animation preferences for the polygon. We havent' set a dynamic trigger
--for the animation, so the polygon's animation will be triggered by
--on page load (the default).
tell animation settings of myPolygon
    set duration to 2
    set motion path points to myMotionPathPoints
end tell
```

TimingSettings

The `timing settings` objects of spreads, pages, and page items control the timing of the animation(s) applied to the object and to any objects contained by the object. `timing settings` contain:

- `timing lists`, which define the trigger event (page load, page click, and so on) that start the animation.
- `timing groups`, which associate a page item or series of page items with a specific timing and define the sequence in which animations are shown.

`timing groups` contain `timing targets`, which define the objects associated with a given `timing group`. `timing targets` also specify a delay value for the animation applied to the page item, relative to the start of the animation of the `timing group` (for the first item in the `timing group`), or from the start of the previous item in the `timing group` (for other items in the `timing group`).

The following script fragment shows how to control the timing of the animation of an object using the various timing objects (for the complete script, refer to TimingSettings). Note that the parameters used to create a `timing group` specify the properties of the first `timing target` in the `timing group`; subsequent `timing targets`, if any, can be added separately.

```
--Given a document "myDocument" and a page "myPage" and the color "myColorA",
--"myColorB", and "myColorC"...
--Add a page items to animate.
tell myPage
    set myPolygonA to make polygon with properties {fill color:myColorA,
    strokeColor:"None"}
    set entire path of path 1 of myPolygonA to {{72, 72}, {72, 144}, {144, 108}}
    set myPolygonB to make polygon with properties {fill color:myColorB,
    strokeColor:"None"}
    set entire path of path 1 of myPolygonB to {{72, 72}, {72, 144}, {144, 108}}
    set myPolygonC to make polygon with properties {fill color:myColorC,
    strokeColor:"None"}
    set entire path of path 1 of myPolygonC to {{72, 72}, {72, 144}, {144, 108}}
end tell
--Create a motion path.
set myMotionPathPoints to {{{{108, 108}, {108, 108}, {108, 108}}, {{516, 108},
{516, 108}, {516, 108}}}, true}
--Set animation preferences for the polygons.
tell animation settings of myPolygonA
    set duration to 2
    set motion path points to myMotionPathPoints
end tell
tell animation settings of myPolygonB
    set duration to 2
    set motion path points to myMotionPathPoints
end tell
tell animation settings of myPolygonC
    set duration to 2
    set motion path points to myMotionPathPoints
end tell
set myTimingSettings to timing settings of parent of myPage
```

```
--Remove the default timing list.
tell myTimingSettings
    delete timing list 1
    --Add a new timing list that triggers when the page is clicked.
    set myTimingList to make timing list with properties {trigger event:on page click}
end tell
--Add the polygons to a single timing group.
tell myTimingList
    set myTimingGroup to make timing group with properties {dynamic target:myPolygonA,
    delay seconds:0}
end tell
tell myTimingGroup
    make timing target with properties {dynamic target:myPolygonB, delay seconds:2}
    make timing target with properties {dynamic target:myPolygonC, delay seconds:2}
end tell
```

Note that attempting to add a page item whose `has custom settings` property (in the `animation settings` object of the page item) is false to a `timing target` generates an error.

The following script fragment shows how to control the sequence of animations applied to objects on a page (for the complete script, refer to MultipleTimingGroups). Note that the order in which `timing groups` are added to a `timing list` determines the order in which the animations play when the trigger event specified in the `timing list` occurs. Some trigger events, such as `on page load`, trigger the animations in the `timing list` (in sequence); others, such as `on page click`, trigger the animations one by one, in sequence, with each instance of the event. For example, a `timing list` containing five `timing groups`, each containing a single `timing target`, and having the trigger event `on page click` requires five mouse clicks to process all the animations.

```
--Given a document "myDocument" and a page "myPage" and the color "myColorA",
--"myColorB", and "myColorC"...
--Add a page items to animate.
tell myPage
    set myPolygonA to make polygon with properties {fill color:myColorA,
    strokeColor:"None"}
    set entire path of path 1 of myPolygonA to {{72, 72}, {72, 144}, {144, 108}}
    set myPolygonB to make polygon with properties {fill color:myColorB,
    strokeColor:"None"}
    set entire path of path 1 of myPolygonB to {{72, 72}, {72, 144}, {144, 108}}
    set myPolygonC to make polygon with properties {fill color:myColorC,
    strokeColor:"None"}
    set entire path of path 1 of myPolygonC to {{72, 72}, {72, 144}, {144, 108}}
    set myPolygonD to make polygon with properties {fill color:myColorA,
    strokeColor:"None"}
    set entire path of path 1 of myPolygonD to {{72, 144}, {72, 216}, {144, 180}}
    set myPolygonE to make polygon with properties {fill color:myColorB,
    strokeColor:"None"}
    set entire path of path 1 of myPolygonE to {{72, 144}, {72, 216}, {144, 180}}
    set myPolygonF to make polygon with properties {fill color:myColorC,
    strokeColor:"None"}
    set entire path of path 1 of myPolygonF to {{72, 144}, {72, 216}, {144, 180}}
end tell
--Create a motion path.
set myMotionPathPointsA to {{{{108, 108}, {108, 108}, {108, 108}}, {{516, 108},
{516, 108}, {516, 108}}}, true}
set myMotionPathPointsB to {{{{108, 180}, {108, 180}, {108, 180}}, {{516, 180},
{516, 180}, {516, 180}}}, true}
--Set animation preferences for the polygons.
--DynamicTriggerEvents.onPageLoad (the default).
set duration of animation settings of myPolygonA to 2
```

```
set motion path points of animation settings of myPolygonA to myMotionPathPointsA
set duration of animation settings of myPolygonB to 2
set motion path points of animation settings of myPolygonB to myMotionPathPointsA
set duration of animation settings of myPolygonC to 2
set motion path points of animation settings of myPolygonC to myMotionPathPointsA
set duration of animation settings of myPolygonD to 2
set motion path points of animation settings of myPolygonD to myMotionPathPointsB
set duration of animation settings of myPolygonE to 2
set motion path points of animation settings of myPolygonE to myMotionPathPointsB
set duration of animation settings of myPolygonF to 2
set motion path points of animation settings of myPolygonF to myMotionPathPointsB
set myTimingSettings to timing settings of parent of myPage
--Remove the default timing list.
tell myTimingSettings
    delete timing list 1
    --Add a new timing list that triggers when the page is clicked.
    set myTimingList to make timing list with properties {trigger event:on page click}
    --Add the polygons to a single timing group.
    tell myTimingList
        set myTimingGroupA to make timing group with properties
        {dynamic target:myPolygonA, delay seconds:0}
        tell myTimingGroupA
            make timing target with properties {dynamic target:myPolygonB,
            delay seconds:2}
            make timing target with properties {dynamic target:myPolygonC,
            delay seconds:2}
        end tell
        --myTimingGroupB will play on the second page click.
        set myTimingGroupB to make timing group with properties {dynamic
        target:myPolygonD, delay seconds:0}
        tell myTimingGroupB
            make timing target with properties {dynamic target:myPolygonE,
            delay seconds:2}
            make timing target with properties {dynamic target:myPolygonF,
            delay seconds:2}
        end tell
    end tell
end tell
```

A given `timing settings` object can contain multiple `timing list` objects, each of which responds to a different trigger event. The following script fragment shows a series of animations triggered by `on page load`, by `on page click` (for the complete script, refer to MultipleTimingLists).

```
set myTimingSettings to timing settings of parent of myPage
tell myTimingSettings
    --At this point, all of the polygons have already been added as timing targets
    --of the default timing list. Change the delay of myPolygonB and myPolygonC,
    --which are the targets of the second and third timing groups.
    set myTimingListA to timing list 1
    tell myTimingListA
        tell timing group 2
            set delay seconds of timing target 1 to 2
        end tell
        tell timing group 3
            set delay seconds of timing target 1 to 2
        end tell
        --Remove the last three timing groups in the timing list.
        --We have to do this, because we don't want these polygons to be
        --animated when the page loads.
        delete timing group -1
```

```
        delete timing group -1
        delete timing group -1
    end tell
    --Add a new timing list that triggers when the page is clicked.
    set myTimingListB to make timing list with properties {trigger event:on page click}
    tell myTimingListB
        set myTimingGroupB to make timing group with properties {dynamic
        target:myPolygonD, delay seconds:0}
        tell myTimingGroupB
            make timing target with properties {dynamic target:myPolygonE,
            delay seconds:2}
            make timing target with properties {dynamic target:myPolygonF,
            delay seconds:2}
        end tell
    end tell
end tell
```

In the previous examples, we've worked with the `timing settings` of the spread containing the page items we want to animate. When you want to animate a page item when a user clicks the item, you'll need to use the `timing settings` of the page item itself, as shown in the following script fragment (for the complete script, refer to PageItemTimingSettings).

```
--Given a page "myPage"...
tell myPage
    set myPolygonA to make polygon with properties {fill color:myColorA,
    stroke color:"None"}
    set entire path of path 1 of myPolygonA to {{72, 72}, {72, 144}, {144, 108}}
    --Create a motion path.
    set myMotionPathPointsA to {{{{108, 108}, {108, 108}, {108, 108}}, {{516, 108},
    {516, 108}, {516, 108}}}, true}
    --Set animation preferences for the polygon.
    tell animation settings of myPolygonA
        set duration to 2
        set motion path points to myMotionPathPointsA
    end tell
    --Remove the default timing list in the timing settings for the spread.
    set myTimingSettings to timing settings of parent of myPage
    tell myTimingSettings
        delete timing list 1
    end tell
    set myTimingSettings to timing settings of myPolygonA
    tell myTimingSettings
        set myTimingList to make timing list with properties {trigger event:on click}
    end tell
    tell myTimingList
        set myTimingGroup to make timing group with properties {dynamic
        target:myPolygonA, delay seconds:0}
    end tell
end tell
```

Animating transformations

Page items can change size, rotation or skewing angles, opacity, and visibility as their animation plays. The `animation settings` of the page item contain properties (such as `rotation array` or `hidden after`) that define the transformations that are applied during animation. The following script fragment shows how to make a page item rotate as it follows a motion path (for the complete script, refer to AnimateRotation).

```
--Given a document "myDocument" and a page "myPage" and the color "myColorA"...
--Add a page items to animate.
tell page 1
    set myPolygon to make polygon with properties {fill color:myColorA,
    stroke color:"None"}
end tell
--Create a motion path.
set entire path of path 1 of myPolygon to {{72, 72}, {72, 144}, {144, 108}}
set myMotionPathPoints to {{{{108, 108}, {108, 108}, {108, 108}}, {{516, 108},
{516, 108}, {516, 108}}}, true}
--Set animation preferences for the polygon.
tell animation settings of myPolygon
    set duration to 2
    set motion path points to myMotionPathPoints
    --Assuming 24 Frames Per Second (FPS)
    --23 = 1 second, 47 = 2 seconds, 71 = 3 seconds, 95 = 4 seconds,
    --119 = 5 seconds, 143 = 6 seconds
    --Since the duration of our animation is 2 seconds, the following line will
    --make the polygon rotate 360 degrees from the start to the end
    --of the animation.
    set rotation array to {{0, 0}, {47, 360}}
end tell
set myTimingSettings to timing settings of parent of myPage
tell myTimingSettings
    --Remove the default timing list.
    delete timing list 1
    --Add a new timing list that triggers when the page is clicked.
    set myTimingList to make timing list with properties {trigger event:on page click}
    tell myTimingList
        --Add the polygon to a single timing group.
        make timing group with properties {dynamic target:myPolygon, delay seconds:0}
    end tell
end tell
```

Scripting offers more control over animation than can be achieved with InDesign's user interface. A scripted animation can, for example, apply transformations at each key frame of a given motion path. For more on this topic, see "Key frames" later in this chapter.

Motion presets

In the preceding examples, we've constructed motion paths and specified animation settings as if we were creating animations from the basic level in InDesign's user interface. But InDesign can also use motion presets to define the animation of page items in a layout. A motion preset can apply a number of animation properties at once, as seen in the following script fragment (for the complete script, refer to MotionPreset). InDesign comes with a large number of motion presets, and you can add new presets using either the user interface or scripting.

```
--Given a page containing the oval "myOvalA"...
set myMotionPreset to Motion Preset "move-right-grow"
tell animation settings of myOvalA
    set duration to 2
    set plays loop to true
    set preset to myMotionPreset
end tell
```

Design options

Design options affect the way that an animated object appears, relative to the motion specified in the object's animation settings. The following script fragment shows how the design options for an animated shape can affect the playback of the animation (for the complete script, refer to DesignOptions).

```
--Given a page containing the ovals "myOvalA" and "myOvalB"...
set myMotionPreset to Motion Preset "move-right-grow"
tell animation settings of myOvalA
    set duration to 2
    set plays loop to true
    set preset to myMotionPreset
    set design option to from current appearance
end tell
tell animation settings of myOvalB
    set duration to 2
    set plays loop to true
    set preset to myMotionPreset
    set design option to to current appearance
end tell
```

Key frames

Key frames are points in the timeline of an animation. With InDesign scripting, you can add key frames at any time in the animation, which gives you the ability to apply changes to objects as they are animated. Key frames are part of the motion path applied to an animated page item, and are specified relative to the duration and speed of the animation. For example, for an animation with a duration of two seconds, playing at 24 frames per second, the last frame in the animation is frame 48.

The following script fragment shows how to add key frames to a motion path, and how to change the transformations applied to an animated page item at each key frame. For the complete script, refer to TransformAnimation.

```
--Given a page containing ovals "myOvalA," "myOvalB," and "myOvalC"...
--The motion path is constructed relative to the center of the object, and key frames
--are based on the duration of the animation divided by the number of frames per second
--(usually 24). The following array sets key frames at the start, midpoint, and end
--of a path.
set myMotionPath to {{0, {{0, 0}, {0, 0}, {0, 0}}}, {23, {{234, 0}, {234, 0},
{234, 0}}}, {47, {{468, 0}, {468, 0}, {468, 0}}}}
tell animation settings of myOvalA
    set duration to 2
    set motion path to myMotionPath
    --The transformation changes at each key frame.
    --scale x array in the form {{keyframe, scale_percentage}, {keyframe,
scalePercentage}, ...}
    set scale x array to {{0, 100}, {23, 200}, {47, 100}}
    --scale y array in the form {{keyframe, scale_percentage}, {keyframe,
scalePercentage}, ...}
    set scale y array to {{0, 100}, {23, 200}, {47, 100}}
    --opacity array in the form {{keyframe, opacity}, {keyframe, opacity},...}
    set opacity array to {{0, 100}, {23, 20}, {47, 100}}
    set plays loop to true
end tell
tell animation settings of myOvalB
    set duration to 2
```

```
    set motion path to myMotionPath
    set scale x array to {{0, 200}, {23, 300}, {47, 50}}
    set scale y array to {{0, 200}, {23, 300}, {47, 50}}
    set opacity array to {{0, 10}, {23, 80}, {47, 60}}
    set plays loop to true
end tell
tell animation settings of myOvalC
    set duration to 2
    set motion path to myMotionPath
    set scale x array to {{0, 50}, {23, 200}, {47, 400}}
    set scale y array to {{0, 50}, {23, 200}, {47, 400}}
    set opacity array to {{0, 100}, {23, 40}, {47, 80}}
    set plays loop to true
end tell
end tell
```

Adding Page Transitions

Page transitions are special effects that appear when you change pages in an exported dynamic document. Adding page transitions using scripting is easy, as shown in the following script fragment (for the complete script, refer to PageTransitions).

```
--Given a document "myDocument" containing at least two spreads...
repeat with myCounter from 1 to (count spreads of myDocument)
    set page transition type of spread myCounter of myDocument to page turn transition
    --This page transition option does not support the page transition direction options
    --or page transition duration properties. If you chose wipe transition
    --(for example), you would be able to set those options, as shown in the next
    --two lines:
    --set page transition direction options of spread myCounter of myDocument
    --to left to right
    --set page transition duration of spread myCounter of myDocument to medium
end repeat
--Export the document to SWF, and you'll see the page transitions.
```

12 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 CS5 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 13, "XML Rules."

Scripting XML Elements

This section shows how to set XML preferences and XML import preferences, import XML, create XML elements, and add XML attributes. The scripts in this section demonstrate techniques for working with the XML content itself; for scripts that apply formatting to XML elements, see "Adding XML Elements to a Layout" on page 185.

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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        tell XML view preferences
            set show attributes to true
            set show structure to true
            set show tagged frames to true
            set show tag markers to true
            set show text snippets to true
        end tell
    end tell
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        tell XML preferences
            set default cell tag color to blue
            set default cell tag name to "cell"
            set default image tag color to brick red
            set default image tag name to "image"
            set default story tag color to charcoal
            set default story tag name to "text"
            set default table tag color to cute teal
            set default table tag name to "table"
        end tell
    end tell
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        tell XML import preferences
            set allow transform to false
            set create link to XML to false
            set ignore unmatched incoming to true
            set ignore whitespace to true
            set import CALS tables to true
            set import style to merge import
            set import text into tables to false
            set import to selected to false
            set remove unmatched existing to false
            set repeat text elements to true
            --The following properties are only used when the
            --allow transform property is set to true.
            --set transform filename to "yukino:myTransform.xsl"
            --If you have defined parameters in your XSL file,
            --you can pass them to the file during the XML import
            --process. For each parameter, enter a list containign two
            --strings. The first string is the name of the parameter,
            --the second is the value of the parameter.
            --set transform parameters to {{"format", "1"}}
        end tell
    end tell
end tell
```

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):

```
tell myDocument
    import XML from "yukino:xml_test.xml"
end tell
```

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):

```
set myRootElement to XML element 1
tell myRootElement
    import XML from "yukino:xml_test.xml"
end tell
--Place the root XML element so that you can see the result.
tell story 1
    place XML using myRootElement
end tell
```

You also can set the `import to selected` property of the `xml import preferences` 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):

```
tell XML import preferences
    set import to selected to true
end tell
select myXMLElement
import XML from "yukino:xml_test.xml"
```

Creating an XML tag

XML tags are the names of the XML elements you want to create in a document. When you import XML, the element names in the XML file are added to the list of XML tags in the document. You also can create XML tags directly, as shown in the following script fragment (from the MakeXMLTags tutorial script):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        --You can create an XML tag without specifying a color for the tag.
        set myXMLTagA to make XML tag with properties {name:"XML_tag_A"}
        --You can define the highlight oclor o fthe XML tag.
        set myXMLTagB to make XML tag with properties
        {name:"XML_tag_B", color:gray}
        --...or you can proved an RGB array to set the color of the tag.
        set myXMLTagC to make XML tag with properties
        {name:"XML_tag_C", color:{0, 92, 128}}
    end tell
end tell
```

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):

```
tell myDocument
    load xml tags "yukino:test.xml"
end tell
```

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):

```
tell myDocument
    save xml tags "yukino:xml_tags.xml"
    version comments "Tag set created October 5, 2006")
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        set myXMLTag to make XML tag with properties {name:"myXMLTag"}
        set myRootElement to XML element 1
        tell myRootElement
            set myXMLElement to make XML element with properties
            {markup tag:myXMLTag}
        end tell
        set contents of myXMLElement to "This is an XML element containing text."
    end tell
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        set myXMLTag to make XML tag with properties {name:"myXMLTag"}
        set myRootElement to XML element 1
        tell myRootElement
            set myXMLElementA to make XML element with properties
            {markup tag:myXMLTag}
            set contents of myXMLElementA to "This is XML element A."
            set myXMLElementB to make XML element with properties
            {markup tag:myXMLTag}
            set contents of myXMLElementB to "This is XML element B."
        end tell
        move myXMLElementA to after myXMLElementB
        end tell
    end tell
end tell
```

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).

```
tell xml element 1 of XML element 1 of myDocument to 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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        set myXMLTag to make XML tag with properties {name:"myXMLTag"}
        set myRootElement to XML element 1
        tell myRootElement
            set myXMLElementA to make XML element with properties
            {markup tag:myXMLTag}
            set contents of myXMLElementA to "This is XML element A."
            set myXMLElementB to make XML element with properties
            {markup tag:myXMLTag}
            set contents of myXMLElementB to "This is XML element B."
        end tell
        duplicate myXMLElementA
    end tell
end tell
```

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 to XML element 1 of XML element 1 of myDocument
tell myXMLElement to 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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        set myXMLTag to make XML tag with properties {name:"myXMLElement"}
        set myRootXMLElement to XML element 1
        tell myRootXMLElement
            set myXMLElement to make XML element with properties
            {markup tag:myXMLTag}
            tell myXMLElement
                make XML comment with properties {value:"This is an XML comment."}
            end tell
        end tell
    end tell
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        set myRootXMLElement to XML element 1
        tell myRootXMLElement
            set myXMLInstruction to make XML instruction with properties
            {target:"xml-stylesheet type=\"text/css\"",
            data:"href=\"generic.css\""}
        end tell
    end tell
end tell
```

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").

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        set myXMLTag to make XML tag with properties {name:"myXMLElement"}
        set myRootXMLElement to XML element 1
        tell myRootXMLElement
            set myXMLElement to make XML element with properties
            {markup tag:myXMLTag}
            tell myXMLElement
                make XML attribute with properties
                {name:"example_attribute", value:"This is an XML attribute."}
            end tell
        end tell
    end tell
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        set myXMLTag to make XML tag with properties {name:"myXMLElement"}
        set myRootXMLElement to XML element 1
    end tell
    tell myRootXMLElement
        set myXMLElement to make XML element with properties
        {markup tag:myXMLTag, contents:"This is content in an XML element."}
    end tell
    tell myXMLElement
        convert to attribute
    end tell
end tell
```

You also can convert an XML attribute to an XML element, as shown in the following script fragment (from the ConvertAttributeToElement tutorial script):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        set myXMLTag to make XML tag with properties {name:"myXMLElement"}
        set myRootXMLElement to XML element 1
    end tell
    tell myRootXMLElement
        set myXMLElement to make XML element with properties
        {markup tag:myXMLTag, contents:"This is content in an XML element."}
    end tell
    tell myXMLElement
        convert to attribute
    end tell
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        set myXMLTag to make XML tag with properties {name:"myXMLElement"}
        set myRootXMLElement to XML element 1
    end tell
    tell myRootXMLElement
        set myXMLElementA to make XML element with properties
        {markup tag:myXMLTag, contents:"This is a paragraph in an XML story."}
        set myXMLElementB to make XML element with properties
        {markup tag:myXMLTag, contents:"This is another paragraph in an XML
        story."}
        set myXMLElementC to make XML element with properties
        {markup tag:myXMLTag, contents:"This is the third paragraph in an
        example XML story."}
        set myXMLElementD to make XML element with properties {markup
        tag:myXMLTag, contents:"This is the last paragraph in the XML story."}
    end tell
    set myXMLStory to xml story 1 of myDocument
    set the point size of text 1 of myXMLStory to 72
end tell
```

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):

```
tell myDocument
    export to "yukino:test.xml" format "XML"
end tell
```

In addition, you can use the `export from selected` property of the `xml export preferences` 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):

```
tell myDocument
    set export from selected of xml export preferences to true
    select xml element 2 of xml element 1
    export to "yukino:selectedXMLElement.xml" format "XML"
    set export from selected of xml export preferences to false
end tell
```

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 `place xml` 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):

```
tell text frame 1 of page 1 of myDocument
    place XML using xml element 1 of myDocument
end tell
```

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):

```
tell XML element 1 of XML element 1 of myDocument
    markup text frame 1 of page 1 of myDocument
end tell
```

Placing XML into page items

Another way to associate an XML element with a page item is to use the `place into frame` 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):

```
tell application "Adobe InDesign CS5"
    set myDocument to document 1
    tell view preferences of myDocument
        set horizontal measurement units to points
        set vertical measurement units to points
    end tell
    --place into frame has two parameters:
    --on: The page, spread, or master spread on which to create the frame
    --geometric bounds: The bounds of the new frame (in page coordinates).
    tell XML element 1 of myDocument
        place into frame on page 1 geometric bounds {72, 72, 288, 288}
    end tell
end tell
```

To associate an XML element with an inline page item (i.e., an anchored object), use the `place into copy` method, as shown in the following script fragment (from the PlaceIntoCopy tutorial script):

```
tell application "Adobe InDesign CS5"
    set myDocument to document 1
    set myPage to page 1 of myDocument
    set myTextFrame to text frame 1 of myPage
    tell XML element 1 of myDocument
        set myFrame to place into copy on myPage place point {288, 72} copy item
        myTextFrame
    end tell
end tell
```

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 `place into inline copy` method, as shown in the following script fragment (from the PlaceIntoInlineCopy tutorial script):

```
tell application "Adobe InDesign CS5"
    set myDocument to document 1
    set myPage to page 1 of myDocument
    tell myPage
        set myNewTextFrame to make text frame with properties
        {geometric bounds:{72, 72, 96, 144}}
    end tell
    set myXMLElement to XML element 3 of XML element 1 of myDocument
    set myFrame to place into inline copy myXMLElement copy item myNewTextFrame
    without retain existing frame
end tell
```

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 myDocument to document 1
set myXMLElement to xml element 2 of xml element 1 of myDocument
set myFrame to place into inline frame myXMLElement place point {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):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        set myXMLTag to make XML tag with properties {name:"myXMLElement"}
        set myRootXMLElement to XML element 1
        tell myRootXMLElement
            set myXMLElementA to make XML element with properties
            {markup tag:myXMLTag}
            set contents of myXMLElementA to "This is a paragraph in an XML
            story."
            set myXMLElementB to make XML element with properties
            {markup tag:myXMLTag}
            set contents of myXMLElementB to "This is a another paragraph in an
            XML story."
            set myXMLElementC to make XML element with properties {markup
            tag:myXMLTag}
            set contents of myXMLElementC to "This is the third paragraph in an
            XML story."
            set myXMLElementD to make XML element with properties {markup
            tag:myXMLTag}
            set contents of myXMLElementD to "This is the last paragraph in an XML
            story."
            tell myXMLElementA
                --By inserting the return character after the XML element, the
                --character becomes part of the content of the parent XML element,
                --and not part of the content of the XML element itself.
                insert text as content using return position after element
            end tell
            tell myXMLElementB
                insert text as content using "Static text: " position before
                element
                insert text as content using return position after element
            end tell
```

```
            tell myXMLElementC
                insert text as content using "Text at the start of an element: "
                position element start
                insert text as content using " Text at the end of an element. "
                position element end
            end tell
            tell myXMLElementD
                insert text as content using "Text before the element: " position
                before element
                insert text as content using " Text after the element. " position
                after element
            end tell
        end tell
    end tell
end tell
```

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 `map XML tags to styles` method of the document, InDesign applies the style to the text, as shown in the following script fragment (from the MapTagsToStyles tutorial script):

```
tell application "Adobe InDesign CS5"
    set myDocument to document 1
    tell myDocument
        --Create a tag to style mapping.
        make XML import map with properties {markup tag:"heading_1",
        mapped style:"heading 1"}
        make XML import map with properties {markup tag:"heading_2",
        mapped style:"heading 2"}
        make XML import map with properties {markup tag:"para_1",
        mapped style:"para 1"}
        make XML import map with properties {markup tag:"body_text",
        mapped style:"body text"}
        --Map the XML tags to the defined styles.
        map XML tags to styles
        --Place the story so that you can see the result of the change.
        set myPage to page 1
        tell page 1
            set myTextFrame to make text frame with properties
            {geometric bounds:my myGetBounds(myDocument, myPage)}
            set myStory to parent story of myTextFrame
        end tell
        tell myStory
            place XML using XML element 1 of myDocument
        end tell
    end tell
end tell
```

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 `XML export map` objects to create the links between XML tags and styles, then use the `map styles to XML tags` method to create the corresponding XML elements, as shown in the following script fragment (from the MapStylesToTags tutorial script):

```
tell application "Adobe InDesign CS5"
    set myDocument to document 1
    tell myDocument
        --Create a tag to style mapping.
        make XML export map with properties {markup tag:"heading_1",
        mapped style:"heading 1"}
        make XML export map with properties {markup tag:"heading_2",
        mapped style:"heading 2"}
        make XML export map with properties {markup tag:"para_1",
        mapped style:"para 1"}
        make XML export map with properties {markup tag:"body_text",
        mapped style:"body text"}
        --Apply the style to tag mapping.
        map styles to XML tags
    end tell
end tell
```

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):

```
tell application "Adobe InDesign CS5"
    set myDocument to document 1
    tell myDocument
        --Create a tag to style mapping.
        repeat with myParagraphStyle in paragraph styles
            set myParagraphStyleName to name of myParagraphStyle
            set myXMLTagName to my myReplace(myParagraphStyleName, " ", "_")
            set myXMLTagName to my myReplace(myXMLTagName, "[", "")
            set myXMLTagName to my myReplace(myXMLTagName, "]", "")
            set myMarkupTag to make XML tag with properties {name:myXMLTagName}
            make XML export map with properties {markup tag:myMarkupTag,
            mapped style:myParagraphStyle}
        end repeat
        map styles to XML tags
    end tell
end tell
```

Marking up graphics

The following script fragment shows how to associate an XML element with a graphic (for the complete script, see MarkingUpGraphics):

```
tell application "Adobe InDesign CS5"
    set myDocument to document 1
    tell myDocument
        set myXMLTag to make xml tag with properties{name:"graphic"}
        tell page 1
            set myGraphic to place "yukino:test.tif"
            --Associate the graphic with a new XML element as you create the
            --element
            tell XML element 1
                set myXMLElement to make XML element with properties
                {markup tag:myXMLTag, content:myGraphic}
            end tell
        end tell
end tell
```

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: `apply paragraph style`, `apply character style`, and `apply object style`. (For the complete script, see ApplyStylesToXMLElements.)

```
main()
on main()
    mySnippet()
end main
on mySnippet()
    tell application "Adobe InDesign CS5"
        set myDocument to make document
        tell myDocument
            set horizontal measurement units of view preferences to points
            set vertical measurement units of view preferences to points
            --Create a series of XML tags.
            set myHeading1XMLTag to make XML tag with properties
            {name:"heading_1"}
            set myHeading2XMLTag to make XML tag with properties
            {name:"heading_2"}
            set myPara1XMLTag to make XML tag with properties {name:"para_1"}
            set myBodyTextXMLTag to make XML tag with properties
            {name:"body_text"}
            --Create a series of paragraph styles.
            set myHeading1Style to make paragraph style with properties
            {name:"heading 1", point size:24}
            set myHeading2Style to make paragraph style with properties
            {name:"heading 2", point size:14, space before:12}
            set myPara1Style to make paragraph style with properties
            {name:"para 1", point size:12, first line indent:0}
            set myBodyTextStyle to make paragraph style with properties
            {name:"body text", point size:12, first line indent:24}
            --Create a character style.
            set myCharacterStyle to make character style with properties
            {name:"Emphasis", font style:"Italic"}
            --Add XML elements.
            set myRootXMLElement to XML element 1
            tell myRootXMLElement
                set myXMLElementA to make XML element with properties {markup
                tag:myHeading1XMLTag, contents:"Heading 1"}
                tell myXMLElementA
```

```
                    insert text as content using return position after element
                    apply paragraph style using myHeading1Style clearing
                    overrides yes
                end tell
                set myXMLElementB to make XML element with properties {markup
                tag:myPara1XMLTag, contents:"This is the first paragraph in the
                article."}
                tell myXMLElementB
                    insert text as content using return position after element
                    apply paragraph style using myPara1Style clearing overrides yes
                end tell
                set myXMLElementC to make XML element with properties {markup
                tag:myBodyTextXMLTag, contents:"This is the second paragraph in
                the article."}
                tell myXMLElementC
                    insert text as content using return position after element
                    apply paragraph style using myBodyTextStyle clearing
                    overrides yes
                end tell
                set myXMLElementD to make XML element with properties
                {markup tag:myHeading2XMLTag, contents:"Heading 2"}
                tell myXMLElementD
                    insert text as content using return position after element
                    apply paragraph style using myHeading2Style clearing
                    overrides yes
                end tell
                set myXMLElementE to make XML element with properties {markup
                tag:myPara1XMLTag, contents:"This is the first paragraph following
                the subhead."}
                tell myXMLElementE
                    insert text as content using return position after element
                    apply paragraph style using myPara1Style clearing overrides yes
                end tell
                set myXMLElementF to make XML element with properties {markup
                tag:myBodyTextXMLTag, contents:"Note:"}
                tell myXMLElementF
                    insert text as content using " " position after element
                    apply character style using myCharacterStyle
                end tell
                set myXMLElementG to make XML element with properties {markup
                tag:myBodyTextXMLTag, contents:"This is the second paragraph
                following the subhead."}
                tell myXMLElementG
                    insert text as content using return position after element
                    apply paragraph style using myBodyTextStyle clearing
                    overrides no
                end tell
            end tell
        end tell
        set myPage to page 1
        tell page 1
            set myTextFrame to make text frame with properties{geometric bounds:
            my myGetBounds(myDocument, myPage)}
        end tell
        set myStory to parent story of myTextFrame
        tell myStory
            place XML using myRootXMLElement
        end tell
    end tell
end mySnippet
```

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 `convert element to table` 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.

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        --Create a series of XML tags.
        set myRowTag to make XML tag with properties {name:"Row"}
        set myCellTag to make XML tag with properties {name:"Cell"}
        set myTableTag to make XML tag with properties {name:"Table"}
        --Add XML elements.
        set myRootXMLElement to XML element 1
        tell myRootXMLElement
            set myTableXMLElement to make XML element with properties {markup
            tag:myTableTag}
            tell myTableXMLElement
                repeat with myRowCounter from 1 to 6
                    set myXMLRow to make XML element with properties {markup
                    tag:myRowTag}
                    tell myXMLRow
                        set myString to "Row " & myRowCounter
                        repeat with myCellCounter from 1 to 4
                            make XML element with properties {markup tag:myCellTag,
                            contents:myString & ":Cell " & myCellCounter}
                        end repeat
                    end tell
                end repeat
                convert element to table row tag myRowTag cell tag myCellTag
            end tell
        end tell
        set myPage to page 1
        tell page 1
            set myTextFrame to make text frame with properties{geometric bounds:
            my myGetBounds(myDocument, myPage)}
        end tell
        set myStory to parent story of myTextFrame
        tell myStory
            place XML using XML element 1 of myDocument
        end tell
    end tell
end tell
```

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 `apply table style` and `apply cell style` methods, as shown in the following script fragment (from the ApplyTableStyles tutorial script):

```
tell application "Adobe InDesign CS5"
    set myDocument to make document
    tell myDocument
        --Create a series of XML tags.
        set myRowTag to make XML tag with properties {name:"Row"}
        set myCellTag to make XML tag with properties {name:"Cell"}
        set myTableTag to make XML tag with properties {name:"Table"}
        --Add XML elements.
        set myRootXMLElement to XML element 1
        tell myRootXMLElement
            set myTableXMLElement to make XML element with properties {markup
            tag:myTableTag}
            tell myTableXMLElement
                repeat with myRowCounter from 1 to 6
                    set myXMLRow to make XML element with properties {markup
                    tag:myRowTag}
                    tell myXMLRow
                        set myString to "Row " & myRowCounter
                        repeat with myCellCounter from 1 to 4
                            make XML element with properties {markup tag:myCellTag,
                            contents:myString & ":Cell " & myCellCounter}
                        end repeat
                    end tell
                end repeat
                convert element to table row tag myRowTag cell tag myCellTag
            end tell
        end tell
        set myPage to page 1
        tell page 1
            set myTextFrame to make text frame with properties{geometric bounds:
            my myGetBounds(myDocument, myPage)}
        end tell
        set myStory to parent story of myTextFrame
        tell myStory
            place XML using XML element 1 of myDocument
        end tell
    end tell
end tell
```

13 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 8, "Events.")

This chapter gives an overview of the structure and operation of XML rules, and shows how to do the following:

- Define an XML rule.
- Apply XML rules.
- Find XML elements using XML rules.
- Format XML data using XML rules.
- Create page items based on XML rules.
- Restructure data using XML rules.
- Use the XML-rules processor.

We assume you already read *Adobe InDesign CS5 Scripting Tutorial* and know how to create and run a script. We also assume you have some knowledge of XML and have read Chapter 12, "XML."

Overview

InDesign's XML rules feature has three parts:

- **XML rules processor (a scripting object)** — Locates XML elements in an XML structure using XPath and applies the appropriate XML rule(s). It is important to note that a script can contain multiple XML rule processor objects, and each rule-processor object is associated with a given XML rule set.
- **Glue code** — A set of routines provided by Adobe to make the process of writing XML rules and interacting with the XML rules-processor easier.
- **XML rules** — The XML actions you add to a script. XML rules are written in scripting code. A rule combines an XPath-based condition and a function to apply when the condition is met. The "apply" function can perform any set of operations that can be defined in InDesign scripting, including changing the XML structure; applying formatting; and creating new pages, page items, or documents.

A script can define any number of rules and apply them to the entire XML structure of an InDesign document or any subset of elements within the XML structure. When an XML rule is triggered by an XML rule processor, the rule can apply changes to the matching XML element or any other object in the document.

You can think of the XML rules feature as being something like XSLT. Just as XSLT uses XPath to locate XML elements in an XML structure, then transforms the XML elements in some way, XML rules use XPath to

locate and act on XML elements inside InDesign. Just as an XSLT template uses an XML parser outside InDesign to apply transformations to XML data, InDesign's XML Rules Processor uses XML rules to apply transformations to XML data inside InDesign.

Why use XML rules?

In prior releases of InDesign, you could not use XPath to navigate the XML structure in your InDesign files. Instead, you needed to write recursive script functions to iterate through the XML structure, examining each element in turn. This was difficult and slow.

XML rules makes it easy to find XML elements in the structure, by using XPath and relying on InDesign's XML-rules processors to find XML elements. An XML-rule processor handles the work of iterating through the XML elements in your document, and it can do so much faster than a script.

XML-rules programming model

An XML rule contains three things:

1. A name (as a string).
2. An `XPath` statement (as a string).
3. An `apply` function.

The XPath statement defines the location in the XML structure; when the XML rules processor finds a matching element, it executes the apply function defined in the rule.

Here is a sample XML rule:

```
to RuleName()
    script RuleName
        property name:"RuleNameAsString"
        property xpath: "ValidXPathSpecifier"
        on apply(element, ruleSet, ruleProcessor)
            --Do something here.
            --Return true to stop further processing of the XML element.
            return false
        end apply
    end script
end RuleName
```

In the above example, `RuleNameAsString` is the name of the rule and matches the `RuleName`; `ValidXPathSpecifier` is an XPath expression. Later in this chapter, we present a series of functioning XML-rule examples.

NOTE: XML rules support a limited subset of XPath 1.0. See "XPath limitations" on page 200."

XML-rule sets

An XML-rule set is an array of one or more XML rules to be applied by an XML-rules processor. The rules are applied in the order in which they appear in the array. Here is a sample XML-rule set:

```
set myRuleSet to {my SortByName(), my AddStaticText(), my LayoutElements(), my
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.as` 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

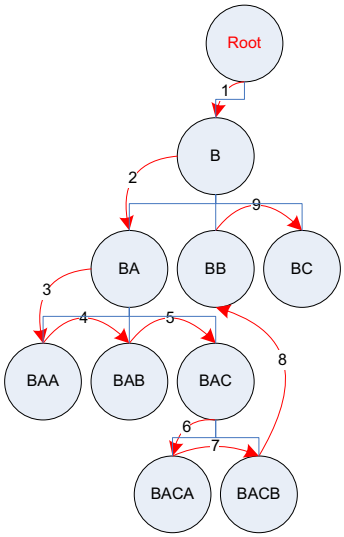
The XML-rules processor iterates through the XML structure of a document by processing each XML element in the order in which it appears in the XML hierarchy of the document. The XML-rules processor uses a forward-only traversal of the XML structure, and it visits each XML element in the structure twice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

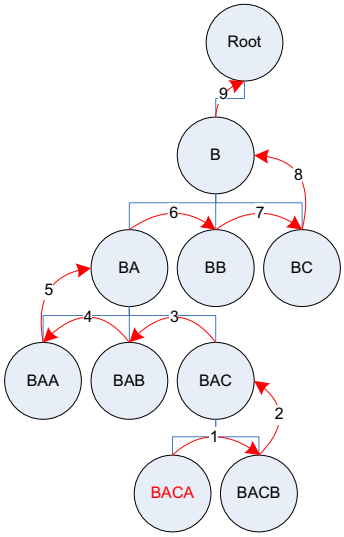
After an XML rule is applied to an XML element, the XML-rules processor continues searching for rules to apply to the descendents of that XML element. An XML rule can alter this behavior by using the `__skipChildren` or `__processChildren` function, or by changing the operation of other rules.

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:

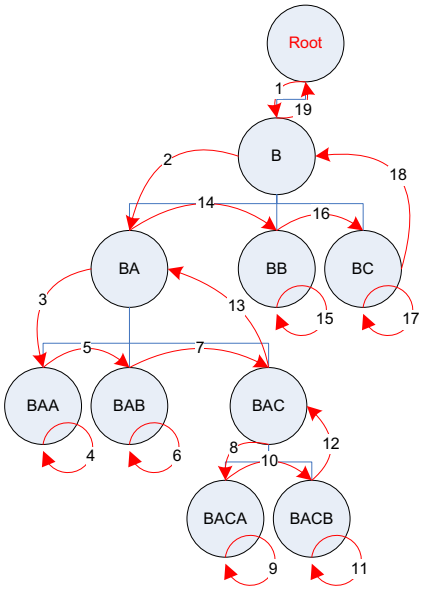


Iteration with `__processChildren` (assuming a rule that matches every XML element in the structure) is shown in the following figure:



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.)

```
to NormalRule()
    script NormalRule
        property name : "NormalRule"
        property xpath : "//XMLElement"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell insertion point -1 of story 1 of document 1
                        set contents to value of XML attribute 1 of myXMLElement &
                        return
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return false
        end apply
    end script
end NormalRule
to ProcessChildrenRule()
    script ProcessChildrenRule
        property name : "ProcessChildrenRule"
        property xpath : "//XMLElement"
        on apply(myXMLElement, myRuleProcessor)
            tell myGlueCode
                __processChildren(myRuleProcessor)
            end tell
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell insertion point -1 of story 1 of document 1
                        set contents to value of XML attribute 1 of myXMLElement &
                        return
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return false
        end apply
    end script
end ProcessChildrenRule
```



Changing structure during iteration

When an XML-rules processor finds a matching XML element and applies an XML rule, the rule can change the XML structure of the document. This can conflict with the process of applying other rules, if the affected XML elements in the structure are part of the current path of the XML-rules processor. To prevent errors that might cause the XML-rules processor to become invalid, the following limitations are placed on XML structure changes you might make within an XML rule:

- **Deleting an ancestor XML element** — To delete an ancestor XML element of the matched XML element, create a separate rule that matches and processes the ancestor XML element.
- **Inserting a parent XML element** — To add an ancestor XML element to the matched XML element, do so after processing the current XML element. The ancestor XML element you add is not processed by the XML-rules processor during this rule iteration (as it appears "above" the current element in the hierarchy).
- **Deleting the current XML element** — You cannot delete or move the matched XML element until any child XML elements contained by the element are processed. To make this sort of change, use the `__skipChildren` function before making the change.
- **No repetitive processing —** Changes to nodes that were already processed will not cause the XML rule to be evaluated again.

Handling multiple matching rules

When multiple rules match an XML element, the XML-rules processor can apply some or all of the matching rules. XML rules are applied in the order in which they appear in the rule set, up to the point that one of the rule `apply` functions returns `true`. In essence, returning `true` means the element was processed. Once a rule returns `true`, any other XML rules matching the XML element are ignored. You can alter this behavior and allow the next matching rule to be applied, by having the XML rule `apply` function return `false`.

When an `apply` function returns `false`, you can control the matching behavior of the XML rule based on a condition other than the `XPath` property defined in the XML rule, like the state of another variable in the script.

XPath limitations

InDesign's XML rules support a limited subset of the XPath 1.0 specification, specifically including the following capabilities:

- Find an element by name, specifying a path from the root; for example, `/doc/title`.
- Find paths with wildcards and node matches; for example, `/doc/*/subtree/node()`.
- Find an element with a specified attribute that matches a specified value; for example, `/doc/para[@font='Courier']`.
- Find an element with a specified attribute that does not match a specified value; for example, `/doc/para[@font !='Courier']`.
- Find a child element by numeric position (but not `last()`); for example, `/doc/para[3]`.
- Find self or any descendent; for example, `//para`.
- Find comment as a terminal; for example, `/doc/comment()`.
- Find PI by target or any; for example, `/doc/processing-instruction('foo')`.
- Find multiple predicates; for example, `/doc/para[@font='Courier'][@size=5][2]`.
- Find along following-sibling axes; for example, `/doc/note/following-sibling::*`.

Due to the one-pass nature of this implementation, the following XPath expressions are specifically excluded:

- No ancestor or preceding-sibling axes, including `..`, `ancestor::`, `preceding-sibling::`.
- No path specifications in predicates; for example, `foo[bar/c]`.
- No `last()` function.
- No `text()` function or text comparisons; however, you can use InDesign scripting to examine the text content of an XML element matched by a given XML rule.
- No compound Boolean predicates; for example, `foo[@bar=font or @c=size]`.
- No relational predicates; for example, `foo[@bar < font or @c > 3]`.
- No relative paths; for example, `doc/chapter`.

Error handling

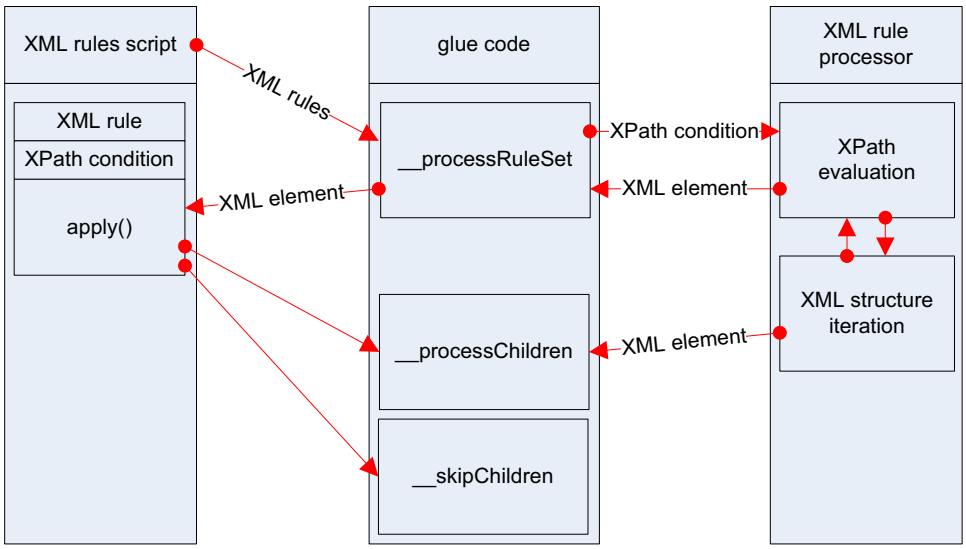
Because XML rules are part of the InDesign scripting model, scripts that use rules do not differ in nature from ordinary scripts, and they benefit from the same error-handling mechanism. When InDesign generates an error, an XML-rules script behaves no differently than any other script. InDesign errors can be captured in the script using whatever tools the scripting language provides to achieve that; for example, `try...catch` blocks.

InDesign does include a series of errors specific to XML-rules processing. An InDesign error can occur at XML-rules processor initialization, when a rule uses a non-conforming XPath specifier (see "XPath limitations" on page 200). An InDesign error also can be caused by a model change that invalidates the state of an XML-rules processor. XML structure changes caused by the operation of XML rules can invalidate the XML-rules processor. These changes to the XML structure can be caused by the script containing the XML-rules processor, another concurrently executing script, or a user action initiated from the user interface.

XML structure changes that invalidate an XML-rules processor lead to errors when the XML-rules processor's iteration resumes. The error message indicates which XML structural change caused the error.

XML rules flow of control

As a script containing XML rules executes, the flow of control passes from the script function containing the XML rules to each XML rule, and from each rule to the functions defined in the glue code. Those functions pass control to the XML-rules processor which, in turn, iterates through the XML elements in the structure. Results and errors are passed back up the chain until they are handled by a function or cause a scripting error. The following diagram provides a simplified overview of the flow of control in an XML-rules script:



XML Rules Examples

Because XML rules rely on XPath statements to find qualifying XML elements, XML rules are closely tied to the structure of the XML in a document. This means it is almost impossible to demonstrate a functional XML-rules script without having an XML structure to test it against. In the remainder of this chapter, we present a series of XML-rules exercises based on a sample XML data file. For our example, we use the product list of an imaginary integrated-circuit manufacturer. Each record in the XML data file has the following structure:

```
<device>
    <name></name>
    <type></type>
    <part_number></part_number>
    <supply_voltage>
        <minimum></minimum>
        <maximum></maximum>
    </supply_voltage>
    <package>
        <type></type>
        <pins></pins>
    </package>
    <price></price>
    <description></description>
</device>
```

The scripts are presented in order of complexity, starting with a very simple script and building toward more complex operations.

Setting up a sample document

Before you run each script in this chapter, import the `XMLRulesExampleData.xml` data file into a document. When you import the XML, turn on the Do Not Import Contents of Whitespace-Only Elements option in the XML Import Options dialog box. Save the file, then choose File > Revert before running each sample script in this section. Alternately, run the following script before you run each sample XML-rule script (see the `XMLRulesExampleSetup.jsx` script file):

```
//XMLRuleExampleSetup.jsx
//
main();
function main(){
    var myDocument = app.documents.add();
    myDocument.xmlImportPreferences.allowTransform = false;
    myDocument.xmlImportPreferences.ignoreWhitespace = true;
    var myScriptPath = myGetScriptPath();
    var myFilePath = myScriptPath.path + "/XMLRulesExampleData.xml"
    myDocument.importXML(File(myFilePath));
    var myBounds = myGetBounds(myDocument, myDocument.pages.item(0));
    myDocument.xmlElements.item(0).placeIntoFrame(myDocument.pages.item(0), myBounds);
    function myGetBounds(myDocument, myPage){
        var myWidth = myDocument.documentPreferences.pageWidth;
        var myHeight = myDocument.documentPreferences.pageHeight;
        var myX1 = myPage.marginPreferences.left;
        var myY1 = myPage.marginPreferences.top;
        var myX2 = myWidth - myPage.marginPreferences.right;
        var myY2 = myHeight - myPage.marginPreferences.bottom;
        return [myY1, myX1, myY2, myX2];
    }
    function myGetScriptPath() {
        try {
            return app.activeScript;
        }
        catch(myError){
            return File(myError.fileName);
        }
    }
}
```

Getting started with XML rules

Here is a very simple XML rule—it does nothing more than add a return character after every XML element in the document. The XML-rule set contains one rule. For the complete script, see AddReturns.

```
global myGlueCode
on run
    tell application "Adobe InDesign CS5"
        set myRootXML to first XML element of document 1
        set myApplicationPath to file path
        set myFilePath to file path as string
        set myFilePath to myFilePath & "Scripts:Xml rules:glue code.scpt"
        set myGlueCode to load script file myFilePath
        set myRuleSet to {my AddReturns()}
    end tell
    tell myGlueCode
        --The third parameter of __processRuleSet is a
        --prefix mapping table; we'll leave it empty.
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell
end run
to AddReturns()
    script AddReturns
        property name : "AddReturns"
        property xpath : "//*"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using return position element end
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return false
        end apply
    end script
end AddReturns
```

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.

```
global myGlueCode
on run
    tell application "Adobe InDesign CS5"
        set myRootXML to first XML element of document 1
        set myApplicationPath to file path
        set myFilePath to file path as string
        set myFilePath to myFilePath & "Scripts:Xml rules:glue code.scpt"
        set myGlueCode to load script file myFilePath
        set myRuleSet to {my ProcessDevice(), my ProcessName(), my ProcessType(), my
ProcessPartNumber(), my ProcessSupplyVoltage(), my ProcessPackageType(), my
ProcessPackageOne(), my ProcessPackages(), my ProcessPrice()}
    end tell
    tell myGlueCode
        --The third parameter of __processRuleSet is a
        --prefix mapping table; we'll leave it empty.
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell
end run
to ProcessDevice()
    script ProcessDevice
        property name : "ProcessDevice"
        property xpath : "/devices/device"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using return position after element
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessDevice
to ProcessName()
    script ProcessName
        property name : "ProcessName"
        property xpath : "/devices/device/name"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using "Device Name: " position
                        before element
                        insert text as content using return position after element
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessName
```

```
to ProcessType()
    script ProcessType
        property name : "ProcessType"
        property xpath : "/devices/device/type"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using "Circuit Type: " position
                        before element
                        insert text as content using return position after element
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessType
to ProcessPartNumber()
    script ProcessPartNumber
        property name : "ProcessPartNumber"
        property xpath : "/devices/device/part_number"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using "Part Number: " position before
                        element
                        insert text as content using return position after element
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessPartNumber
to ProcessSupplyVoltage()
    script ProcessSupplyVoltage
        property name : "ProcessSupplyVoltage"
        property xpath : "/devices/device/supply_voltage"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using "Supply Voltage: From: "
                        position before element
                        insert text as content using return position after element
                        tell XML element 1
                            insert text as content using " to " position after
                            element
```

```
                    end tell
                    tell XML element -1
                        insert text as content using " volts" position after
                        element
                    end tell
                end tell
            on error myError
                set myReturnString to myError
            end try
        end tell
        return true
    end apply
end script
end ProcessSupplyVoltage
to ProcessPackageType()
    script ProcessPackageType
        property name : "ProcessPackageType"
        property xpath : "/devices/device/package/type"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using "-" position after element
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessPackageType
--Add the text "Package:" before the list of packages.
to ProcessPackageOne()
    script ProcessPackageOne
        property name : "ProcessPackageOne"
        property xpath : "/devices/device/package[1]"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using "Package: " position before
                        element
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessPackageOne
--Add commas between the package types and a return at the end of the packages.
to ProcessPackages()
    script ProcessPackages
        property name : "ProcessPackages"
```

```
        property xpath : "/devices/device/package"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        set myIndex to index of myXMLElement
                        if myIndex is not 1 then
                            insert text as content using ", " position before element
                        end if
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessPackages
to ProcessPrice()
    script ProcessPrice
        property name : "ProcessPrice"
        property xpath : "/devices/device/price"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using return position before element
                        insert text as content using "Price: $" position
                        before element
                        insert text as content using return position after element
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessPrice
```

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):

```
global myGlueCode
on run
    tell application "Adobe InDesign CS5"
        set myRootXML to first XML element of document 1
        set myApplicationPath to file path
        set myFilePath to file path as string
        set myFilePath to myFilePath & "Scripts:Xml rules:glue code.scpt"
        set myGlueCode to load script file myFilePath
        set myRuleSet to {my ListItems()}
    end tell
    tell myGlueCode
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell
end run
to ListItems()
    script ListItems
        property name : "ListItems"
        property xpath : "/xmlElement/item[1]/following-sibling::*"
        on apply(myXMLElement, myRuleProcessor)
            tell application "Adobe InDesign CS5"
                tell myXMLElement
                    insert text as content using ", " position before element
                end tell
            end tell
            return false
        end apply
    end script
end ListItems
```

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.

```
global myGlueCode
on run
    tell application "Adobe InDesign CS5"
        set myRootXML to first XML element of document 1
        set myApplicationPath to file path
        set myFilePath to file path as string
        set myFilePath to myFilePath & "Scripts:Xml rules:glue code.scpt"
        set myGlueCode to load script file myFilePath
        set myRuleSet to {my MoveElement()}
    end tell
    tell myGlueCode
        --The third parameter of __processRuleSet is a
        --prefix mapping table; we'll leave it empty.
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell
end run
to MoveElement()
    script MoveElement
```

```
        property name : "MoveElement"
        property xpath : "/devices/device/part_number"
        on apply(myXMLElement, myRuleProcessor)
            tell myGlueCode
                __skipChildren(myRuleProcessor)
            end tell
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        set myParent to parent
                        set myTargetXMLElement to XML element 1 of myParent
                        move to before myTargetXMLElement
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end MoveElement
```

Duplicating XML elements with XML rules

As discussed in Chapter 12, "XML," XML elements have a one-to-one relationship with their expression in a layout. If you want the content of an XML element to appear more than once in a layout, you need to duplicate the element. The following script shows how to duplicate elements using XML rules. For the complete script, see DuplicateXMLElement. Again, this rule uses `__skipChildren` to avoid invalid XML object references.

```
global myGlueCode
on run
    tell application "Adobe InDesign CS5"
        set myRootXML to first XML element of document 1
        set myApplicationPath to file path
        set myFilePath to file path as string
        set myFilePath to myFilePath & "Scripts:Xml rules:glue code.scpt"
        set myGlueCode to load script file myFilePath
        set myRuleSet to {my DuplicateElement()}
    end tell
    tell myGlueCode
        --The third parameter of __processRuleSet is a
        --prefix mapping table; we'll leave it empty.
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell
end run
to DuplicateElement()
    script DuplicateElement
        property name : "DuplicateElement"
        property xpath : "/devices/device/part_number"
        on apply(myXMLElement, myRuleProcessor)
            --Because this rule makes changes to the XML structure,
            --you must use __skipChildren to avoid invalidating
            --the XML object references.
            tell myGlueCode
                __skipChildren(myRuleProcessor)
```

```
            end tell
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        duplicate
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end DuplicateElement
```

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.

```
global myGlueCode
on run
    tell application "Adobe InDesign CS5"
        set myRootXML to first XML element of document 1
        set myApplicationPath to file path
        set myFilePath to file path as string
        set myFilePath to myFilePath & "Scripts:Xml rules:glue code.scpt"
        set myGlueCode to load script file myFilePath
        set myRuleSet to {my AddAttribute()}
    end tell
    tell myGlueCode
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell
end run
to AddAttribute()
    script AddAttribute
        property name : "AddAttribute"
        property xpath : "/devices/device/part_number"
        --Adds the content of an XML element to an attribute of
        --the parent of the XML element. This can make finding the
        --element by its contnet much easier and faster.
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
```

```
                tell myXMLElement
                    set myString to contents of text 1
                    tell parent
                        make XML attribute with properties {name:"part_number",
                        value:myString}
                    end tell
                end tell
            on error myError
                set myReturnString to myError
            end try
        end tell
        return true
    end apply
end script
end AddAttribute
```

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):

```
global myGlueCode
on run
    tell application "Adobe InDesign CS5"
        set myRootXML to first XML element of document 1
        set myApplicationPath to file path
        set myFilePath to file path as string
        set myFilePath to myFilePath & "Scripts:Xml rules:glue code.scpt"
        set myGlueCode to load script file myFilePath
        set myRuleSet to {my ConvertToAttribute()}
    end tell
    tell myGlueCode
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell
end run
to ConvertToAttribute()
    script ConvertToAttribute
        property name : "ConvertToAttribute"
        property xpath : "/devices/device/part_number"
        on apply(myXMLElement, myRuleProcessor)
            tell myGlueCode
                __skipChildren(myRuleProcessor)
            end tell
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        convert to attribute ("PartNumber")
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ConvertToAttribute
```

To move data from an XML attribute to an XML element, use the `convertToElement` method, as described in Chapter 12, "XML."

Applying multiple matching rules

When the `apply` function of an XML rule returns true, the XML-rules processor does not apply any further XML rules to the matched XML element. When the `apply` function returns false, however, the XML-rules processor can apply other rules to the XML element. The following script shows an example of an XML-rule `apply` function that returns false. This script contains two rules that will match every XML element in the document. The only difference between them is that the first rule applies a color and returns false, while the second rule applies a different color to every other XML element (based on the state of a variable, `myCounter`). For the complete script, see ReturningFalse.

```
global myGlueCode, myCounter
on run
    set myCounter to 0
    tell application "Adobe InDesign CS5"
        set myRootXML to first XML element of document 1
        set myApplicationPath to file path
        set myFilePath to file path as string
        set myFilePath to myFilePath & "Scripts:Xml rules:glue code.scpt"
        set myGlueCode to load script file myFilePath
        set myRuleSet to {my ReturnFalse(), my ReturnTrue()}
        set myDocument to document 1
        tell myDocument
            set myColorA to make color with properties {name:"ColorA",
            model:process, color value:{0, 100, 80, 0}}
            set myColorB to make color with properties {name:"ColorB",
            model:process, color value:{100, 0, 80, 0}}
        end tell
    end tell
    tell myGlueCode
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell
end run
to ReturnTrue()
    script ReturnTrue
        property name : "ReturnTrue"
        property xpath : "//*"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        if myCounter mod 2 = 0 then
                            set fill color of text 1 to "ColorA"
                        end if
                        set myCounter to myCounter + 1
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            --Do not process the element with any further matching rules.
            return true
        end apply
    end script
```

```
end ReturnTrue
to ReturnFalse()
    script ReturnFalse
        property name : "ReturnFalse"
        property xpath : "//*"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        set fill color of text 1 to "ColorB"
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            --Leave the XML element available for further matching rules.
            return false
        end apply
    end script
end ReturnFalse
```

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.

```
global myGlueCode
on run
    tell application "Adobe InDesign CS5"
        set myFilePath to file path as string
        set myFilePath to myFilePath & "Scripts:Xml rules:glue code.scpt"
        tell document 1
            set myRootXML to first XML element
            set myColorA to make color with properties
            {name:"ColorA", model:process, color value:{0, 100, 80, 0}}
        end tell
    end tell
    set myGlueCode to load script file myFilePath
    set myRuleSet to {my AddAttribute()}
    tell myGlueCode
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell

    set myRuleSet to {my FindAttribute()}
    tell myGlueCode
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell
end run
to AddAttribute()
    script AddAttribute
        property name : "AddAttribute"
        property xpath : "/devices/device/part_number"
        --Adds the content of an XML element to an attribute of
```

```
        --the parent of the XML element. This can make finding the
        --element by its contnet much easier and faster.
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        set myString to contents of text 1
                        tell parent
                            make XML attribute with properties
                            {name:"part_number", value:myString}
                        end tell
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end AddAttribute
to FindAttribute()
    script FindAttribute
        property name : "FindAttribute"
        property xpath : "/devices/device[@part_number = 'DS001']"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        set fill color of text 1 to "ColorA"
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return false
        end apply
    end script
end FindAttribute
```

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

```
global myGlueCode
on run
    tell application "Adobe InDesign CS5"
        set myFilePath to file path as string
        set myFilePath to myFilePath & "Scripts:Xml rules:glue code.scpt"
        tell document 1
            set myRootXML to first XML element
            set myColorA to make color with properties {name:"ColorA",
            model:process, color value:{0, 100, 80, 0}}
        end tell
    end tell
    set myGlueCode to load script file myFilePath
    set myRuleSet to {my FindByFindTExt()}
    tell myGlueCode
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell
end run
to FindByFindTExt()
    script FindByFindTExt
        property name : "FindByFindText"
        property xpath : "/devices/device/description"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                set find text preferences to nothing
                set change text preferences to nothing
                set find what of find text preferences to "triangle"
                try
                    tell myXMLElement
                        set myFoundItems to find text
                        if (count myFoundItems) is greater than 0 then
                            set fill color of text 1 to "ColorA"
                        end if
                    end tell
                on error myError
                    set myReturnString to myError
                end try
                set find text preferences to nothing
                set change text preferences to nothing
            end tell
            return false
        end apply
    end script
end FindByFindTExt
```

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

```
global myGlueCode
on run
    tell application "Adobe InDesign CS5"
        set myFilePath to file path as string
        set myFilePath to myFilePath & "Scripts:Xml rules:glue code.scpt"
        tell document 1
            set myRootXML to first XML element
            set myColorA to make color with properties {name:"ColorA",
            model:process, color value:{0, 100, 80, 0}}
        end tell
    end tell
    set myGlueCode to load script file myFilePath
    set myRuleSet to {my FindByFindGrep()}
    tell myGlueCode
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell
end run
to FindByFindGrep()
    script FindByFindGrep
        property name : "FindByFindGrep"
        property xpath : "/devices/device/description"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                set find grep preferences to nothing
                set change grep preferences to nothing
                --Find all of the devices the mention both "pulse" and
                --"triangle" in their description.
                set find what of find grep preferences to
                "(?i)pulse.*?triangle|triangle.*?pulse"
                try
                    tell myXMLElement
                        set myFoundItems to find grep
                        if (count myFoundItems) is greater than 0 then
                            set fill color of text 1 to "ColorA"
                        end if
                    end tell
                on error myError
                    set myReturnString to myError
                end try
                set find grep preferences to nothing
                set change grep preferences to nothing
            end tell
            return false
        end apply
    end script
end FindByFindGrep
```

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.

```
global myGlueCode
on run
    tell application "Adobe InDesign CS5"
        set myFilePath to file path as string
        set myFilePath to myFilePath & "Scripts:Xml rules:glue code.scpt"
        tell document 1
            set myRootXML to first XML element
            set myXMLTag to make XML tag with properties {name:"VCOs"}
            tell myRootXML
                set myXMLElement to make XML element with properties
                {markup tag:myXMLTag}
            end tell
        end tell
    end tell
    set myGlueCode to load script file myFilePath
    set myRuleSet to {my ExtractVCO()}
    tell myGlueCode
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell
end run
to ExtractVCO()
    script ExtractVCO
        property name : "ExtractVCO"
        property xpath : "/devices/device/type"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    if contents of text 1 of myXMLElement is "VCO" then
                        set myNewXMLElement to duplicate parent of myXMLElement
                    end if
                    move myNewXMLElement to end of XML element -1 of XML element 1
                    of document 1
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ExtractVCO
```

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):

```
global myGlueCode
on run
    tell application "Adobe InDesign CS5"
        set myApplicationPath to file path
        set myFilePath to file path as string
        set myFilePath to myFilePath & "Scripts:Xml rules:glue code.scpt"
        tell document 1
            tell view preferences
                set horizontal measurement units to points
                set vertical measurement units to points
                set ruler origin to page origin
            end tell
            set myRootXML to first XML element
            set myColor to make color with properties
            {model:process, color value:{0, 100, 100, 0}, name:"Red"}
            make paragraph style with properties {name:"DeviceName",
            point size:24, leading:24, space before:24, fill color:"Red",
            rule above:true, rule above offset:24}
            make paragraph style with properties {name:"DeviceType",
            point size:12, font style:"Bold", leading:12}
            make paragraph style with properties {name:"PartNumber",
            point size:12, font style:"Bold", leading:12}
            make paragraph style with properties {name:"Voltage",
            point size:10, leading:12}
            make paragraph style with properties {name:"DevicePackage",
            pointSize:10, leading:12}
            make paragraph style with properties {name:"Price", point size:10,
            leading:12, font style:"Bold", space after:12}
        end tell
    end tell
    set myGlueCode to load script file myFilePath
    set myRuleSet to {my ProcessDevice(), my ProcessName(), my ProcessType(), my
ProcessPartNumber(), my ProcessSupplyVoltage(), my ProcessPackageType(), my
ProcessPrice(), my ProcessPackageOne(), my ProcessPackages()}
    tell myGlueCode
        __processRuleSet(myRootXML, myRuleSet, {})
    end tell
end run
to ProcessDevice()
    script ProcessDevice
        property name : "ProcessDevice"
        property xpath : "/devices/device"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using return position after element
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessDevice
to ProcessName()
    script ProcessName
```

```
        property name : "ProcessName"
        property xpath : "/devices/device/name"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using return position after element
                        apply paragraph style using "DeviceName"
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessName
to ProcessType()
    script ProcessType
        property name : "ProcessType"
        property xpath : "/devices/device/type"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using "Circuit Type: " position
                        before element
                        insert text as content using return position after element
                        apply paragraph style using "DeviceType"
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessType
to ProcessPartNumber()
    script ProcessPartNumber
        property name : "ProcessPartNumber"
        property xpath : "/devices/device/part_number"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using "Part Number: " position before
                        element
                        insert text as content using return position after element
                        apply paragraph style using "PartNumber"
                    end tell
                on error myError
                    set myReturnString to myError
                end try
```

```
            end tell
            return true
        end apply
    end script
end ProcessPartNumber
to ProcessSupplyVoltage()
    script ProcessSupplyVoltage
        property name : "ProcessSupplyVoltage"
        property xpath : "/devices/device/supply_voltage"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using "Supply Voltage: From: "
                        position before element
                        tell XML element 1
                            insert text as content using " to " position after
                            element
                        end tell
                        tell XML element -1
                            insert text as content using " volts" position after
                            element
                        end tell
                        insert text as content using return position after element
                        apply paragraph style using "Voltage"
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessSupplyVoltage
to ProcessPackageType()
    script ProcessPackageType
        property name : "ProcessPackageType"
        property xpath : "/devices/device/package/type"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using "-" position after element
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessPackageType
--Add the text "Package:" before the list of packages.
to ProcessPackageOne()
    script ProcessPackageOne
        property name : "ProcessPackageOne"
```

```
        property xpath : "/devices/device/package[1]"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        insert text as content using "Package: " position before
                        element
                        apply paragraph style using "DevicePackage"
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessPackageOne
--Add commas between the package types and a return at the end of the packages.
to ProcessPackages()
    script ProcessPackages
        property name : "ProcessPackages"
        property xpath : "/devices/device/package"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
                    tell myXMLElement
                        set myIndex to index of myXMLElement
                        if myIndex is not 1 then
                            insert text as content using ", " position before element
                        end if
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
            return true
        end apply
    end script
end ProcessPackages
to ProcessPrice()
    script ProcessPrice
        property name : "ProcessPrice"
        property xpath : "/devices/device/price"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                try
```

```
                tell myXMLElement
                    insert text as content using return position before element
                    insert text as content using "Price: $" position before
                    element
                    insert text as content using return position after element
                    apply paragraph style using "Price"
                end tell
            on error myError
                set myReturnString to myError
            end try
        end tell
        return true
    end apply
end script
end ProcessPrice
```

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:

```
to ProcessDevice()
    script ProcessDevice
        property name : "ProcessDevice"
        property xpath : "/devices/device"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                set myDocument to document 1
                tell myDocument
                    try
                        tell myXMLElement
                            insert text as content using return position after
                            element
                        end tell
                        if (count text frames of page 1) is greater than 0 then
                            set myPage to make page
                        else
                            set myPage to page 1
                        end if
                        set myBounds to my myGetBounds(myDocument, myPage)
                        set myTextFrame to place into frame myXMLElement on myPage
                        geometric bounds myBounds
                        tell text frame preferences of myTextFrame
                            set first baseline offset to leading offset
                        end tell
                    on error myError
                        set myReturnString to myError
                    end try
                end tell
            end tell
            return true
        end apply
    end script
end ProcessDevice
```

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

```
to ProcessType()
    script ProcessType
        property name : "ProcessType"
        property xpath : "/devices/device/type"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                set myDocument to document 1
                tell myDocument
                    try
                        tell myXMLElement
                            insert text as content using "Circuit Type: " position
                            before element
                            insert text as content using return position after
                            element
                            apply paragraph style using "DeviceType"
                        end tell
                        set myPage to page -1 of myDocument
                        set myBounds to my myGetBounds(myDocument, myPage)
                        set myX1 to item 2 of myBounds
                        set myY1 to (item 1 of myBounds) - 24
                        set myX2 to myX1 + 48
                        set myY2 to item 1 of myBounds
                        set myTextFrame to place into frame myXMLElement on myPage
                        geometric bounds {myY1, myX1, myY2, myX2}
                        set fill color of myTextFrame to "Red"
                        tell text frame preferences of myTextFrame
                            set inset spacing to {6, 6, 6, 6}
                        end tell
                    on error myError
                        set myReturnString to myError
                    end try
                end tell
            end tell
            return true
        end apply
    end script
end ProcessType
```

Creating Tables using XML Rules

You can use the `convert element to table` 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.

```
to ProcessDevice()
    script ProcessDevice
        property name : "ProcessDevice"
        property xpath : "//device[@type = 'VCO']"
        on apply(myXMLElement, myRuleProcessor)
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                set myDocument to document 1
                tell myDocument
                    try
                        set myContainerElement to XML element -1 of XML element -1
                        of XML element 1
                        tell myContainerElement
                            set myNewElement to make XML element with properties
                            {markup tag:"Row"}
                        end tell
                    on error myError
                        set myReturnString to myError
                    end try
                end tell
            end tell
            return false
        end apply
    end script
end ProcessDevice
```

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

```
to ProcessPrice()
    script ProcessPrice
        property name : "ProcessPrice"
        property xpath : "//device[@type = 'VCO']/price"
        on apply(myXMLElement, myRuleProcessor)
            tell myGlueCode
                __skipChildren(myRuleProcessor)
            end tell
            global myReturnString
            set myReturnString to "OK"
            tell application "Adobe InDesign CS5"
                set myDocument to document 1
                tell myDocument
                    try
                        set myRootElement to XML element 1
                        set myVCOs to XML element -1 of myRootElement
                        set myTable to XML element -1 of myVCOs
                        set myContainerElement to XML element -1 of myTable
                        tell myContainerElement
                            set myNewElement to make XML element with properties
                            {markup tag:"Column"}
```

```
                    end tell
                    set myXMLElement to move myXMLElement to beginning of
                    myNewElement
                    tell myXMLElement
                        insert text as content using "$" position before element
                    end tell
                on error myError
                    set myReturnString to myError
                end try
            end tell
        end tell
        return true
    end apply
end script
end ProcessPrice
```

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

```
tell myContainerElement
    set myTable to convert to table row tag myRowTag cell tag myColumnTag
end tell
```

Scripting the XML-rules Processor Object

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

```
set myXPath to {"/devices/device"}
set myXMLMatches to mySimulateXPath(myXPath)
--At this point, myXMLMatches contains all of the XML elements
--that matched the XPath expression provided in myXPath.
--In a real script, you could now process the elements.
--For this example, however, we'll simply display a message.
display dialog ("Found " & (count myXMLMatches) & " matching XML elements.")
on mySimulateXPath(myXPath)
    set myMatchingElements to {}
    tell application "Adobe InDesign CS5"
        set myRuleProcessor to make XML rule processor with properties
        {rule paths:myXPath}
        set myDocument to document 1
        set myRootXMLElement to XML element 1 of myDocument
        tell myRuleProcessor
            set myMatchData to start processing rule set start element
            myRootXMLElement
            repeat until myMatchData is nothing
                local myMatchData
                local myMatchingElements
                set myXMLElement to item 1 of myMatchData
                set myMatchingElements to myMatchingElements & myXMLElement
                set myMatchData to find next match
            end repeat
        end tell
        return myMatchingElements
    end tell
end mySimulateXPath
```

14 Track Changes

Writers can track, show, hide, accept, and reject changes as a document moves through the writing and editing process. All changes are recorded and visualized to make it easier to review a document.

This tutorial shows how to script the most common operations involving tracking changes.

We assume you already read Chapter 2, "Getting Started" 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.

Tracking Changes

This section shows how to navigate tracked changes, accept changes, and reject changes using scripting.

Whenever anyone adds, deletes, or moves text within an existing story, the change is marked in galley and story views.

Navigating tracked changes

If the story contains a record of tracked changes, the user can navigate sequentially through tracked changes. The following script show how to navigate the tracked changes (for the complete script, refer to GetTrackchange).

The script below uses the change index number to iterate through changes:

```
tell application "Adobe InDesign CS5"
    set myDocument to document 1
    tell myDocument
        set myStory to story 1
        tell myStory
            if (track changes) is true then
                set myChangeCounter to count
                set myChange to change 1
            end if
        end tell
    end tell
end tell
```

Accepting and reject tracked changes

When changes are made to a story, by you or others, the change-tracking feature enables you to review all changes and decide whether to incorporate them into the story. You can accept and reject changes—added, deleted, or moved text—made by any user.

In the following script, the change is accepted (for the complete script, refer to AcceptChange):

```
tell application "Adobe InDesign CS5"
    set myDocument to document 1
    tell myDocument
        set myStory to story 1
        tell  myStory
            set myChange = myStory change 1
            tell  myChange
                accept
            end tell
        end tell
    end tell
end tell
```

In the following script, the change is rejected (for the complete script, refer to RejectChange):

```
tell application "Adobe InDesign CS5"
    set myDocument to document 1
    tell myDocument
        set myStory to story 1
        tell  myStory
            set myChange = myStory change 1
            tell  myChange
                reject
            end tell
        end tell
    end tell
end tell
```

Information about tracked changes

Change information includes include date and time. The following script shows the information of a tracked change (for the complete script, refer to GetChangeInfo):

```
--Shows how to get track change informations.
tell application "Adobe InDesign CS5"
    set myDocument to document 1
    tell myDocument
        set myStory to story 1
        tell myStory
            set myChange to change 1
            tell myChange
            -- change type (inserted text/deleted text/moved text, r/o)
                set myTypes to change type
                set myCharacters to characters
                set myDate to date
                set myInsertionPoints to insertion points
                set myLines to lines
            -- paragraphs A collection of paragraphs.
                set myParagraphs to paragraphs
                set myStoryOffset to story offset
                set myTextColumns to text columns
                set myTextStyleRanges to text style ranges
                set myTextsetiableInstances to text variable instances
            -- The user who made the change. Note: Valid only when changes is true.
                set myUserName to user name
                -- Words  A collection of words
                set myWords to words
            end tell
        end tell
    end tell
end tell
```

Preferences for Tracking Changes

Track-changes preferences are user settings for tracking changes. For example, you can define which changes are tracked (adding, deleting, or moving text). You can specify the appearance of each type of tracked change, and you can have changes identified with colored change bars in the margins. The following script shows how to set and get these preferences (for the complete script, refer to GetChangePreference):

```
tell application "Adobe InDesign CS5"
    set myTrackChangesPreference to track changes preferences
    tell myTrackChangesPreference
        --  added background color choice (change background uses galley background
color/change background uses user color/change background uses change pref color) : The
background color option for added text.
        set myAddedBackgroundColorChoice to added background color choice
        set added background color choice to change background uses change pref color

        --added text color choice (change uses galley text color/change uses change pref
color) : The color option for added text.
        set myAddedTextColorChoice to added text color choice
        set added text color choice to change uses change pref color

        --background color for added text (any) : The background color for added text,
specified as an InCopy UI color. Note: Valid only when added background color choice is
change background uses change pref color.
        set myBackgroundColorForAddedText to background color for added text
        set background color for added text to gray

        --background color for deleted text (any) : The background color for deleted
```

```
text, specified as an InCopy UI color. Note: Valid only when deleted background color
choice is change background uses change pref color.
	set myBackgroundColorForDeletedText to background color for deleted text
	set background color for deleted text to red

	--background color for moved text (any) : The background color for moved text,
specified as an InCopy UI color. Note: Valid only when moved background color choice is
change background uses change pref color.
	set myBackgroundColorForMovedText to background color for moved text
	set background color for moved text to pink
	--change bar color (any) : The change bar color, specified as an InCopy UI color.
	set myChangeBarColor to change bar color
	set change bar color to charcoal
	--deleted background color choice (change background uses galley background
color/change background uses user color/change background uses change pref color) : The
background color option for deleted text.
	set myDeletedBackgroundColorChoice to deleted background color choice
	set deleted background color choice to change background uses change pref color

	--deleted text color choice (change uses galley text color/change uses change
pref color) : The color option for deleted text.
	set myDeletedTextColorChoice to deleted text color choice
	set deleted text color choice to change uses change pref color

	--location for change bar (left align/right align) : The change bar location.
	set myLocationForChangeBar to location for change bar
	set location for change bar to left align

	--marking for added text (none/strikethrough/underline single/outline) : The
marking that identifies added text.
	set myMarkingForAddedText to marking for added text
	set marking for added text to strikethrough

	--marking for deleted text (none/strikethrough/underline single/outline) : The
marking that identifies deleted text.
	set myMarkingForDeletedText to marking for deleted text
	set marking for deleted text to underline single

	--marking for moved text (none/strikethrough/underline single/outline) : The
marking that identifies moved text.
	set myMarkingForMovedText to marking for moved text
	set marking for moved text to outline

	--moved background color choice (change background uses galley background
color/change background uses user color/change background uses change pref color) : The
background color option for moved text.
	set myMovedBackgroundColorChoice to moved background color choice
	set moved background color choice to change background uses galley background
color

	-- moved text color choice (change uses galley text color/change uses change pref
color) : The color option for moved text.
	set myMovedTextColorChoice to moved text color choice
	set moved text color choice to change uses change pref color

	-- If true, displays added text.
	set myShowAddedText to show added text
	set show added text to true

	--  If true, displays change bars.
```

```
        set myShowChangeBars to show change bars
        set show change bars to true

        -- If true, displays deleted text.
        set myShowDeletedText to show deleted text
        set show deleted text to true

        -- If true, displays moved text.
        set myShowMovedText to show moved text
        set show moved text to true

        -- If true, includes deleted text when using the Spell Check command.
        set mySpellCheckDeletedText to spell check deleted text
        set spell check deleted text to true

        --The color for added text, specified as an InCopy UI color. Note: Valid only
when added text color choice is change uses change pref color.
        set myTextColorForAddedText to text color for added text
        set text color for added text to blue

        --  text color for deleted text (any) : The color for deleted text, specified as
an InCopy UI color. Note: Valid only when deleted text color choice is change uses
change pref color.
        set myTextColorForDeletedText to text color for deleted text
        set text color for deleted text to yellow

        --  The color for moved text, specified as an InCopy UI color. Note: Valid only
when moved text color choice is change uses change pref color.
        set myTextColorForMovedText to text color for moved text
        set text color for moved text to green
    end tell
end tell
```