ADOBE® INCOPY® CS5.5

ADOBE INCOPY CS5.5 SCRIPTING GUIDE: APPLESCRIPT

© 2011 Adobe Systems Incorporated. All rights reserved.

Adobe® InCopy® CS5.5 Scripting Guide: AppleScript

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

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

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

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

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

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

Contents

1 Introduction . . . 7

How to use the scripts in this document . . . 7
About the structure of the scripts . . . 7

For more information . . . 8

About InCopy . . . 8

Relationships between InCopy and InDesign files . . . 8
Stories . . . 8
Page geometry . . . 9
Metadata . . . 9
The document model . . . 9
User-interface differences . . . 10

Design and architecture . . . 10

2 Getting Started . . . 11

Installing scripts . . . 11

Running scripts . . . 12

Using the scripts panel . . . 12

AppleScript language details . . . 12

Using the scripts in this document . . . 12

Your first InCopy script . . . 13
Walking through the script . . . 13

Scripting terminology and the InCopy object model . . . 14
Scripting terminology . . . 14
Understanding the InDesign and InCopy object model . . . 17
Measurements and positioning . . . 21

Adding features to “Hello World” . . . 22

3 Scripting Features . . . 23

Script preferences . . . 23

Getting the current script . . . 24

Script versioning . . . 24
Targeting . . . 25
Compilation . . . 25
Interpretation . . . 25

Using the do script method . . . 25
Sending parameters to do script . . . 26
Returning values from do script . . . 26

Running scripts at start-up . . . 27

4 Text and Type **28**

Entering and importing text 28
- Stories and text frames 28
- Adding text to a story 28
- Replacing text 29
- Inserting special characters 29

Placing text and setting text-import preferences 30

Exporting text and setting text-export preferences 33

Text objects 37
- Selections 38
- Moving and copying text 38
- Text objects and iteration 40

Formatting text 41
- Setting text defaults 41
- Fonts 44
- Applying a font 45
- Changing text properties 45
- Changing text color 46
- Creating and applying styles 47
- Deleting a style 49
- Importing paragraph and character styles 49

Finding and changing text 50
- Find/change preferences 50
- Finding text 51
- Finding and changing formatting 52
- Using grep 53
- Using glyph search 55

Tables 55

Autocorrect 59

Footnotes 59

5 User Interfaces **61**

Dialog-box overview 61

Your first InCopy dialog box 62

Adding a user interface to “Hello World” 63

Creating a more complex user interface 64

Working with ScriptUI 65
- Creating a progress bar with ScriptUI 66
- Creating a button-bar panel with ScriptUI 66

6 Menus **69**

Understanding the menu model 69
- Localization and menu names 72

Running a menu action from a script 72

Adding menus and menu items 73

Menus and events 73
Working with script menu actions 74

7 **Events 78**
Understanding the event scripting model 78
About event properties and event propagation 80
Working with eventListeners 81
A sample "afterNew" eventListener 84

8 **Notes 85**
Entering and importing a note 85
Adding a note to a story 85
Replacing text of a note 86
Converting between notes and text 86
Converting a note to text 86
Converting text to a note 86
Expanding and collapsing notes 87
Collapsing a note 87
Expanding a note 87
Removing a note 87
Navigating among notes 87
Going to the first note in a story 87
Going to the next note in a story 88
Going to the previous note in a story 88
Going to the last note in a story 88

9 **Tracking Changes 89**
Tracking Changes 89
Navigating tracked changes 89
Accepting and reject tracked changes 89
Information about tracked changes 90
Preferences for tracking changes 91

10 **Assignments 94**
Assignment object 94
Opening assignment files 94
Iterating through assignment properties 94
Assignment packages 95
An assignment story 95
Assigned-story object 96
Iterating through the assigned-story properties 96

11 **XML 97**
Overview 97
The best approach to scripting XML in InCopy 97

Scripting XML Elements 98
- Setting XML preferences 98
- Setting XML import preferences 98
- Importing XML 99
- Creating an XML tag 100
- Loading XML tags 100
- Saving XML tags 100
- Creating an XML element 101
- Moving an XML element 101
- Deleting an XML element 101
- Duplicating an XML element 101
- Removing items from the XML structure 102
- Creating an XML comment 102
- Creating an XML processing instruction 102
- Working with XML attributes 103
- Working with XML stories 104
- Exporting XML 104

Adding XML elements to a story 104
- Associating XML elements with text 105
- Applying styles to XML elements 108
- Working with XML tables 109

1 Introduction

Chapter Update Status		
CS5.5	Unchanged	Content not guaranteed to be current.

This document shows how to do the following:

- Work with the Adobe® InCopy® scripting environment.
- Use advanced scripting features.
- Work with text and type in an InCopy 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.

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 InCopy scripting home page, at: http://www.adobe.com/products/InCopy/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 InCopy folder. At that point, you can run the scripts from the Scripts panel inside InCopy.

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’s no reason for you to construct your scripts this way. Most of the time, the part of the script you’ll be interested in will be inside the “mySnippet” handler.

For more information

For more information on InCopy scripting, you also can visit the InCopy 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.

About InCopy

InCopy is a collaborative, text-editing application developed for integrated use with Adobe InDesign®. InCopy enables you to track changes, add editorial notes, and fit copy tightly into the space designed for it. InCopy uses the same text-composition engine as InDesign, so InCopy and InDesign fit copy within a layout with identical composition.

InCopy is for the editorial environment. It allows editorial workflow participants to collaborate on magazines, newspapers, and corporate publishing, enabling concurrent text and layout editing. Its users are editors, writers, proofreaders, copy editors, and copy processors.

InCopy shares many panels and palettes with InDesign but also provides its own user-interface items.

Relationships between InCopy and InDesign files

Relationships between InDesign and InCopy files are important because of the division of labor in a publication workflow that occurs when much of the same material is opened and modified in both applications.

There are two common scenarios for exporting from InCopy:

- You can export an (IDML based) ICML file.
- You can export an (INX based) INCX file.

There are two common scenarios for exporting from InDesign that involve InCopy in some way:

- Stories exported from InDesign as InCopy files are XML files or streams; the InCopyExport and InCopyWorkflow plug-ins loaded into InDesign provide this function. Some practical implications of this approach for InCopy files are that they are much smaller, they are faster over the network, they do not contain any page geometry, and data within the XML file or stream is available outside InDesign/InCopy (for search engines, database tools, and so on).
- Groupings within an article (such as a headline, byline, copy, graphics, or captions) also can be exported. InDesign and Incopy support the creation of groupings with assignment files, which handle file management by adding an additional file that tracks the other files. In essence, an assignment is a set of files whose contents are assigned to one person for some work to be done (for example, copy edit, layout, and/or writing). Any stories in an assignment are exported as InCopy files. Geometry information and the relationship of the files are held in the assignment file. InDesign allows the user to export a given set of stories by exporting into an assignment. InCopy opens all stories that are in an assignment together (as one unit). For details, see Chapter 10, "Assignments."

Stories

Each InCopy file represents one story. An InDesign document containing several stories can be modularized to the same number of InCopy documents, through export. Those exported InDesign stories

contain a link, which may be viewed in the Links panel (InDesign) or the Assignments palette as assignment files (InCopy).

InCopy does not maintain a link to the InDesign document it is associated with (if one exists). InDesign maintains any links with InCopy files as bidirectional links.

Stories can be structured in XML. This means XML data can be contained within XML data. This feature can be used to design a data structure in which the raw text of a story is contained within an outer structure that contains data specific to InCopy (like styles).

Within InCopy, content can be saved in an ICML/INCX format or, if there is structure in the story, the logical structure can be exported in XML.

An ICML or INCX file can contain both InCopy data and marked-up text. If the file is exported as XML data, the data specific to InCopy is stripped out, leaving the marked-up content minus the information about how it is to be styled.

Page geometry

InCopy files do not contain page geometry. When geometry is needed, it must be obtained from the InDesign document. InCopy can open InDesign documents and extract design information and links to the exported stories where needed. When page geometry is desired from within InCopy, assignment files can be supplied with it.

Metadata

The Adobe Extensible Metadata Platform (XMP) provides a practical method for creating, interchanging, and managing metadata. InCopy files support XMP.

Just as InDesign provides the File > File Info command to view XMP data, InCopy provides the File > Content File Info command. System integrators can retain this data or strip it out during export.

Metadata added to stories by third-party software developers is preserved when incorporated into InDesign documents. Added metadata can be viewed within InDesign (from the File Info dialog box, available from the Links panel menu), as well as viewed within InCopy. Further, third-party software developers can add functionality to InDesign to view that metadata in a custom user interface.

An extensibility point exists for service providers to add metadata content to InCopy files. For more information, see Chapter 11, "XML."

The document model

InDesign documents are the basis for all content in InDesign. InCopy also uses InDesign documents, but they are not the default document type.

In both InDesign and InCopy, the basic document always is a database; in InCopy, however, this document may be an incomplete document. In InDesign, the main document typically is an opened InDesign file, but it also can be an opened INX or IDML file, which typically appears to be an unsaved InDesign document.

InCopy has other permutations. There is the basic InDesign file, as well as a new document with an InCopy story (or plain or RTF text) imported into it. Also, there are IDML- and INX-based assignment files, which have some part of an InDesign file stored in an XML file. The InDesign/InCopy document model corresponds to the base required model plug-in set, versioned against changes over time. It is important

that all IDML/INX scripting work in both InDesign and InCopy, so documents can be moved with high fidelity between the applications.

User-interface differences

InDesign and InCopy share most of their panels, but InCopy has a smaller set and several additional toolbars along the top, left, and bottom screen borders. Most InCopy panels also can be docked on these bars, providing a smaller but always-visible view of the panel.

InCopy also has a custom window layout with multiple views, in a main window with three tabs: Galley view, Story view, and Layout view. Layout view is the InDesign window view. Galley and story views are simply the story-editor view, with and without accurate line endings, respectively.

Design and architecture

Story/file relationship

ICML is an IDML-based representation of an InCopy story. It represents the future direction of InDesign/InCopy and is an especially good choice if you need to edit a file outside of InDesign.

ICML format

Each InCopy file or stream is in XML. An advantage of this is that InCopy files can be parsed easily and opened by any text editor.

INCX format

INCX is an INX-based representation of an InCopy story. This format is not as readable as ICML, but it is still available to support INCX-based workflows.

Document operations

InCopy provides default implementations of document operations (file actions) like New, Save, Save As, Save A Copy, Open, Close, Revert, and Update Design. All these InCopy file actions are in one plug-in (InCopyFileActions) in source-code form. Software developers or system integrators are expected to replace this with their own implementations, to customize the interaction for their workflow system.

Using XMP metadata

Users can enter and edit metadata by choosing File > Content File Info. This metadata is saved in the InCopy file. Software developers and system integrators can create and store their own metadata using the XMP SDK.

2 Getting Started

Chapter Update Status		
CS5.5	Unchanged	Content not guaranteed to be current.

Scripting is the most powerful feature in Adobe® InCopy® CS5. No other feature—no tool, panel, or dialog box you see in the program's user interface—can save you as much time, trouble, and money as scripting.

This document is for every InCopy user. It does not matter if you havenever created a script before; this manual shows you how to get started. If you wrote scripts before for other applications, this manual shows you how to apply your knowledge to InCopy scripting. It covers installing and running an InCopy script, and it describes what InCopy scripting can and cannot do. It also discusses the software you need to get started writing your own scripts.

Almost anything you can do with the InCopy user interface, you can do with a script. You can enter and format text, find and change text, add notes, and print or export the pages of the document. Any action that can change a document or its contents can be scripted. There are even a few things that you can do in scripting that you cannot do using the user interface.

Scripts can create menus, add menu items, create and display dialogs and panels, and respond to your user-interface selections. Scripts can read and write text files, parse XML data, and communicate with other applications. Scripts can do everything from very small tasks (like setting a tab stop at the location of the text cursor) to providing complete features. You can start with very simple scripts that do only one thing and move on to scripts that automate your entire publishing workflow.

Most of the things scripting cannot do—like setting up a workspace or defining a set of keyboard shortcuts—are related to the user interface. In addition, scripts cannot add new kinds of objects to an InCopy document or add new, fundamental capabilities to the program, like a new text-composition engine. For that type of extensibility, you must to turn to the InCopy Software Development Kit (SDK), which shows you how to write compiled plug-ins using C++.

This document talks about Adobe InDesign® as well as InCopy, because InCopy almost always is used in conjunction with InDesign documents. In addition, InDesign and InCopy scripting are very similar. For more on InDesign scripting, see *Adobe InDesign Scripting Tutorial* and *Adobe InDesign Scripting Guide*.

Installing scripts

To install an InCopy script, just put the script file in the Scripts Panel folder in the Scripts folder in your InCopy application folder.

Alternately, put the script in the Scripts Panel folder in your user-preferences folder. You can find your user preferences folder at the following locations, where `<username>` is your user name and ~ (tilde) is your system volume:

Mac OS®: `/Users/<username>/Library/Preferences/Adobe InCopy/Version 7.0/<locale>/Scripts`

Once the script is in the folder, it appears in the Scripts panel inside InCopy (choose Window > Scripts to display the panel).

You also can put in the Scripts Panel folder aliases/shortcuts to scripts or folders containing scripts, and they will appear in the Scripts panel.

Running scripts

To run a script, display the Scripts panel (choose Window > Scripts), then double-click the script name in the Scripts panel. Many scripts display user-interface items (like dialogs or panels) and display alerts if necessary.

Using the scripts panel

The Scripts panel can run compiled or uncompiled AppleScripts (files with the file extension `.spt`, `.as`, or `.applescript`), JavaScripts (files with the file extension `.js` or `.jsx`), VBScripts (files with the extension `.vbs`), or executable programs from the Scripts panel.

To edit a script shown in the Scripts panel, hold down Option (Mac OS) or Alt (Windows) key and double-click the script's name. This opens the script in the editor you defined for the script file type.

To open the folder containing a script shown in the Scripts panel, hold down the Command (Mac OS) or Ctrl-Shift (Windows) keys and double-click the script's name. Alternately, choose Reveal in Finder (Mac OS) or Reveal in Explorer (Windows) from the Scripts panel menu. The folder containing the script opens in the Finder (Mac OS) or Explorer (Windows).

Scripts run as a series of actions, which means you can undo the changes the script made to a document by choosing Undo from the Edit menu. This can help you troubleshoot a script, as you can step backward through each change.

To add a keyboard shortcut for a script, choose Edit > Keyboard Shortcuts, select an editable shortcut set from the Set menu, then choose Product Area > Scripts. A list of the scripts in your Scripts panel appears. Select a script and assign a keyboard shortcut as you would for any other InCopy feature.

AppleScript language details

You must have AppleScript version 1.6 or higher and an AppleScript script editor. AppleScript comes with all Apple® systems, and it can be downloaded free from the Apple Web site. The Apple Script Editor is included with the Mac OS; access it from the menus:

Mac OSX 10.5 `Applications > AppleScript > Script Editor`

Mac OSX 10.6 `Applications > Utilities > AppleScript Editor`

Third-party script editors, such as Script Debugger (from Late Night Software, http://www.latenightsw.com) also are available.

Using the scripts in this document

To use any script from this document, you can either open the tutorial script file (the filename is given before each script) or copy the code shown in this chapter.

The script files are stored in a zip archive, `InCopyCS5ScriptingGuideScripts.zip`. When you uncompress the archive, you can move the folder containing the scripts written in the scripting language

you want to use (AppleScript, JavaScript, or VBScript) to your Scripts Panel folder. Working with the script files is much easier than entering the script yourself or copying and pasting from this document.

If you do not have access to the script archive, you can enter the scripting code shown in this chapter. To do this:

1. Copy the script from this Adobe PDF document and paste it into the Apple Script Editor.
2. Save the script as a plain-text file in the Scripts Panel folder (see "Installing scripts" on page 11), using the file extension `.applescript`.
3. Choose Windows > Scripts to display the Scripts panel.
4. Double-click the script name in the Scripts panel to run the script.

Entering scripts manually will work only for the scripts shown in this chapter. The scripts shown in the other chapters are script fragments, not complete scripts. To run these scripts, you must use the scripts from the script archive.

NOTE: If you are copying and pasting scripts from this document, be aware that line breaks caused by the layout of the document can cause errors in your script. As it can be very difficult to find such errors, we recommend hat you use the scripts in the zip archive.

Your first InCopy script

Next, we create an InCopy script that creates a new document, adds a text frame, then enters text in the text frame. While this seems trivial, it demonstrates how to do the following:

- Establish communication with InCopy.
- Create a new document.
- Add text to a story.

Start the Script Editor application (you can find it in your Applications folder, inside the AppleScript folder). Enter the following script (or open the `HelloWorld.applescript` tutorial script):

```
tell application "Adobe InCopy CS5"
    set myDocument to make document
    set myStory to story 1 of myDocument
    tell myStory
        set contents to "Hello World!"
    end tell
end tell
```

Save the script as text with the file extension `.applescript` in the Scripts Panel folder (see "Installing scripts" on page 11). To run the script, double-click the script name in the Scripts panel or click Run in the Script Editor window.

Walking through the script

Here is a step-by-step analysis of what the Hello World script does.

1. Establish communication with the InCopy application object:

```
tell application "Adobe InCopy CS5"
```

2. Create a new document and a reference to the document:

```
Set myDocument to make document
```

3. Get a reference to the first story in the document (a standalone document always contains a story):

```
set myStory to story 1 of myDocument
tell myStory
```

4. Add text to the story by setting the contents property to a string.

```
set contents of myStory to "Hello World!"
```

Scripting terminology and the InCopy object model

Now that you created your first InCopy script, it is time to learn more about the terminology of scripting languages in general and InCopy scripting in particular.

Scripting terminology

First, let's review a few common scripting terms and concepts.

Comments

Comments give you a way to add descriptive text to a script. The scripting system ignores comments as the script executes; this prevents comments from producing errors when you run your script. Comments are useful when you want to document the operation of a script (for yourself or someone else). In this document, we use comments in the tutorial scripts.

To include a comment in an AppleScript, type `--` to the left of your comment or surround the comment with `(*` and `*)`. For example:

```
--this is a comment
(* and so is this *)
```

Values

The point size of a text character, the contents of a note, and the filename of a document are examples of *values* used in InCopy scripting. Values are the data your scripts use to do their work.

The *type* of a value defines what sort of data the value contains. For example, the value type of the contents of a word is a text string; the value type of the leading of a paragraph is a number. Usually, the values used in scripts are numbers or text. The following table explains the value types most commonly used in InCopy scripting:

Value Type	What it is	Example
Boolean	Logical True or False.	`True`
Integer	Whole numbers (no decimal points). Integers can be positive or negative.	`14`

Value Type	What it is	Example
Fixed or real	A high-precision number that can contain a decimal point.	`13.9972`
String	A series of text characters. Strings appear inside (straight) quotation marks.	`"I am a string"`
List	A list of values (the values can be any type).	`{"0p0", "0p0", "16p4", "20p6"}`

Converting values from one type to another

AppleScript provides ways to convert variable values from one type to another. The most common converstions involved converting numbers to strings (so you can enter them in text or display them in dialogs) or converting strings to numbers (so you can use them to set a point size or page location).

```
--To convert from a number to a string:
set myNumber to 2
set myString to (myNumber as string)
--To convert from a string to a number:
set myString to "2"
set myNumber to (myString as integer)
--if your string contains a decimal value, use "as real" rather than "as integer"
```

Variables

A *variable* is a container for a value. They are called “variables” because the values they contain might change. A variable might hold a number, a string of text, or a reference to an InCopy object. Variables have names, and you refer to a variable by its name. To put a value into a variable, you *assign* the data to the variable.

In all examples and tutorial scripts that come with InCopy, all variables start with `my`. This enables you to easily differentiate variables we created in a script from scripting-language terms.

Assigning a value to a variable

Assigning values or strings to variables is fairly simple, as shown in these examples:

```
set myNumber to 10
set myString to "Hello, World!"
set myTextFrame to make text frame at page 1 of myDocument
```

Try to use descriptive names for your variables, like `firstPage` or `corporateLogo`, rather than `x` or `c`. This makes your script easier to read. Longer names do not affect the execution speed of the script.

Variable names must be one word, but you can use internal capitalization (like `myFirstPage`) or underscore characters (`my_first_page`) to create more readable names. Variable names cannot begin with a number, and they cannot contain punctuation or quotation marks.

Array variables

In AppleScript, an array is called a *list*. A list is a container for a series of values:

```
set myArray to {1, 2, 3, 4}
```

To refer to an item in an array, refer to its index in the array.In AppleScript, the first item in an array is item 1:

```
set myFirstArrayItem to item 1 of myArray
```

Arrays can include other arrays, as shown in the following examples:

```
set myArray to {{0, 0}, {72, 72}}
```

Finding the value type of a variable

Sometimes, your scripts must make decisions based on the value type of an object. If you are working on a script that operates on a text selection, for example, you might want that script to stop if nothing is selected.

```
-- Given a variable of unknown type, "myMysteryVariable"...
set myType to class of myMysteryVariable
--myType will be an AppleScript type (e.g., rectangle)
```

Operators

Operators use variables or values to perform calculations (addition, subtraction, multiplication, and division) and return a value. For example:

```
MyWidth/2
```

returns a value equal to half of the content of the variable `myWidth`.

You also can use operators to perform comparisons (equal to (=), not equal to(<>), greater than (>), or less than (<)). For example:

```
MyWidth > myHeight
```

returns the value `true` (or 1) if `myWidth` is greater than `myHeight`; otherwise, `false` (0).

In AppleScript, use the ampersand (&) to concatenate (or join) two strings. For example:

```
"Pride " & "and Prejudice"
```

returns the string:

```
"Pride and Prejudice"
```

Conditional statements

“If the size of the selected text is 12 points, set the point size to 10 points.” This is an example of a *conditional statement*. Conditional statements make decisions; they give your scripts a way to evaluate something (like the color of the selected text, number of pages in the document, or date), then act according to the result. Most conditional statements start with `if`.

Control structures

If you could talk to InCopy, you might say, “Repeat the following procedure 20 times.” In scripting terms, this is a *control structure*. Control structures provide repetitive processes, or *loops*. The idea of a loop is to repeat an action over and over again, with or without changes between instances (or *iterations*) of the loop, until a specific condition is met. Control structures usually start with the `repeat`.

Handlers

Handlers are scripting modules to which you can refer from within your script. Typically, you send a value or series of values to a handler and get back another value or values. There is nothing special about the code used in handlers; they are simply conveniences to avoid having to type the same lines of code repeatedly in your script. Handlers start with `on`.

Understanding the InDesign and InCopy object model

When you think about InCopy and InDesign documents, you probably organize the programs and their components in your mind. You know that paragraphs are contained by text frames, which in turn appear on a page. A page is a part of a spread, and one or more spreads make up a document. Documents contain colors, styles, layers, and master spreads. As you think about the objects in the documents you create, you intuitively understand that there is an order to them.

InDesign and InCopy "think" about the contents of a document the same way you do. A document contains pages, which contain page items (text frames, rectangles, ellipses, and so on). Text frames contain characters, words, paragraphs, and anchored frames; graphics frames contain images, EPSs, or PDFs; groups contain other page items. The things we mention here are the *objects* that make up an InDesign publication, and they are what we work with when we write InDesign and InCopy scripts.

Objects in your publication are arranged in a specific order: paragraphs are inside a story, which is inside a document, which is inside the InCopy application object. When we speak of an *object model* or a *hierarchy*, we are talking about this structure. Understanding the object model is key to finding the object you want to work with. Your best guide to InCopy scripting is your knowledge of InCopy itself.

Objects have *properties* (attributes). For example, the properties of a text object include the font used to format the text, point size, and leading applied to the text.

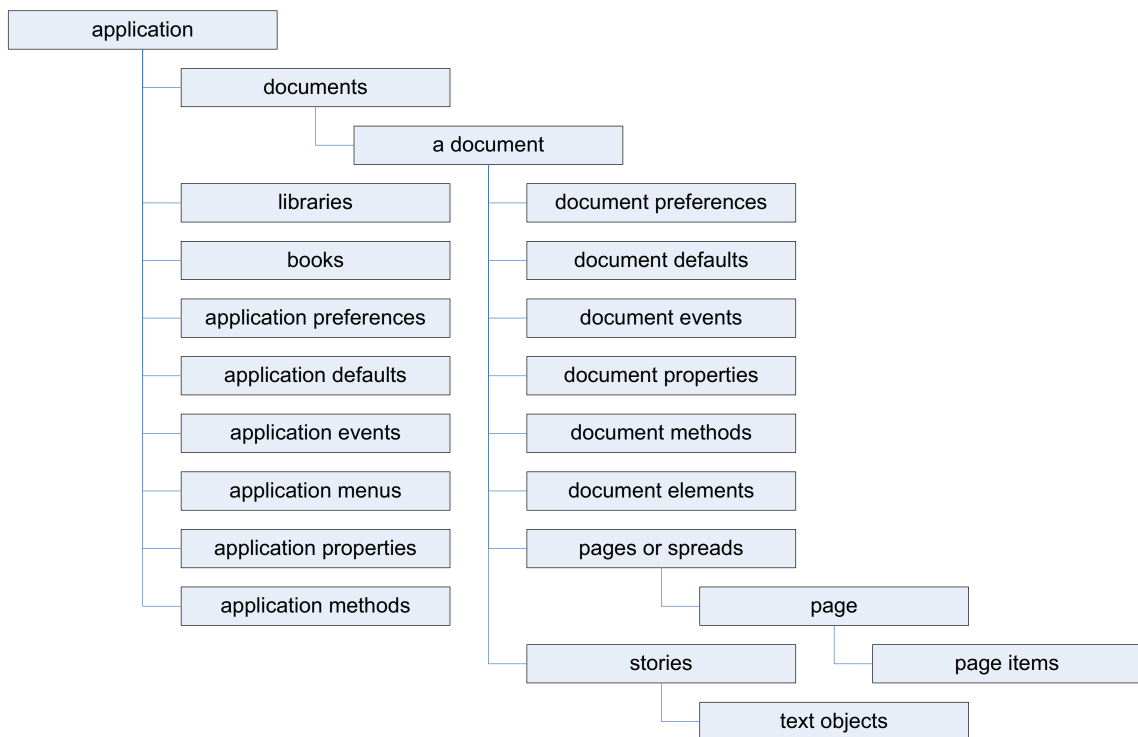
Properties have values; for example, the point size of text can be either a number (in points) or the string "Auto" for auto leading. The fill-color property of text can be set to a color, gradient, mixed ink, or swatch.

Properties also can be *read/write* or *read only*. Read/write properties can be set to other values; read-only properties cannot.

Objects also have *methods*. Methods are the verbs of the scripting world, the actions an object can perform. For example, the document object has print, export, and save methods.

Methods have *parameters*, or values that define the effect of the method. The open method, for example, has a parameter that defines the file you want to open.

The following block diagram is an overview of the InCopy object model. The diagram is not a comprehensive list of objects available to InCopy scripting; instead, it is a conceptual framework for understanding the relationships between the types of objects.



The objects in the diagram are explained in the following table:

Term	What it represents:
Application	InCopy.
Application defaults	Application default settings, such as colors, paragraph styles, and object styles. Application defaults affect all new documents.
Application events	The things that happen as a user or script works with the application. Events are generated by opening, closing, or saving a document or choosing a menu item. Scripts can be triggered by events.
Application menus	The menus, submenus, and context menus displayed in the InCopy user interface. Scripts can be attached to menu choices and can execute menu actions.
Application methods	The actions the application can take; for example, finding and changing text, copying the selection, creating new documents, and opening libraries.
Application preferences	Examples are text preferences, PDF export preferences, and document preferences. Many preferences objects also exist at the document level. Just as in the user interface, application preferences are applied to new documents; document preferences change the settings of a specific document.
Application properties	The properties of the application; for example, the full path to the application, the locale of the application, and the user name.
Books	A collection of open books.
Document	An InCopy document.

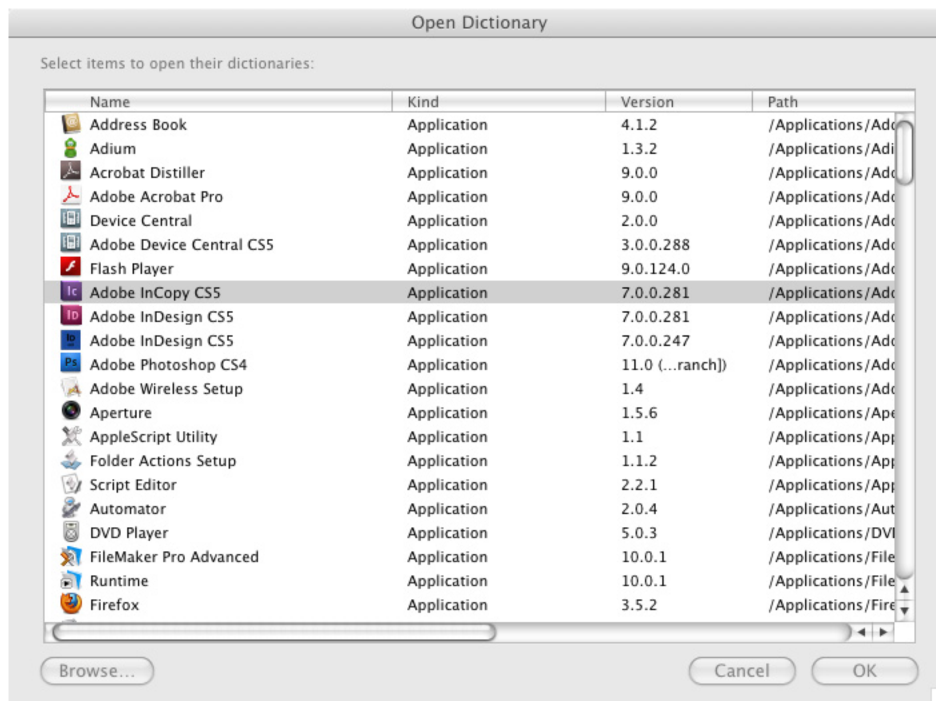
Term	What it represents:
Document defaults	Document default settings, such as colors, paragraph styles, and text formatting defaults.
Document elements	For example, the stories, imported graphics, and pages of a document. The figure above shows pages and stories, because those objects are extremely important containers for other objects; however, document elements also include rectangles, ovals, groups, XML elements, and any other type of object you can import or create.
Document events	Events that occur at the document level, such as importing text. See *application events* in this table.
Document methods	The actions the document can take; for example, closing a document, printing a document, and exporting a document.
Document preferences	The preferences of a document, such as guide preferences, view preferences, or document preferences.
Document properties	For example, the document filename, number of pages, and zero-point location.
Documents	A collection of open documents.
Libraries	A collection of open libraries.
Page	One page in an InCopy document.
Page item	Any object you can create or place on a page. There are many types of page items, such as text frames, rectangles, graphic lines, and groups.
Pages or spreads	The pages or spreads in an InCopy document.
Stories	The text in an InCopy document.
Text objects	Characters, words, lines, paragraphs, and text columns are examples of text objects in an InCopy story.

Looking at the InCopy object model

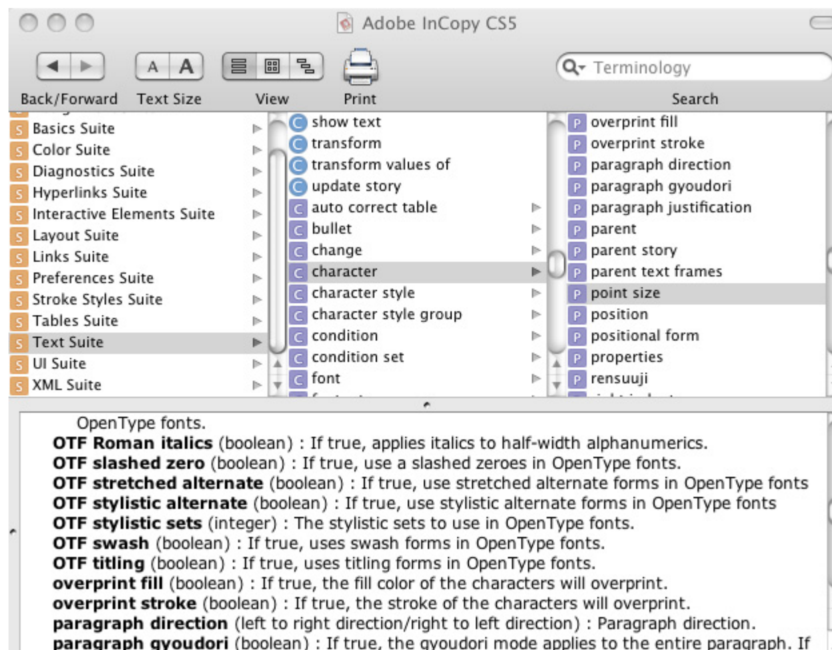
You can view the InCopy object model from inside your script-editing application. All reference information on objects and their properties and methods is stored in the model and can be viewed,

To view the InCopy AppleScript dictionary:

1. Start InCopy.
2. Start the Apple Script Editor.
3. In the Script Editor, choose File > Open Dictionary. The Script Editor displays a list of scriptable applications:



4. Select your copy of InCopy and click OK. The Script Editor displays a list of suites in InCopy (collections of related objects):



5. Select a suite to see the objects and methods (commands) it contains, and select an object to see the properties associated with the object.

Measurements and positioning

All items and objects in InCopy are positioned on the page according to measurements you specify. It is useful to know how the InCopy coordinate system works and what measurement units it uses.

Coordinates

InCopy, like every other page-layout and drawing program, uses simple, two-dimensional geometry to set the position of objects on a page or spread. The horizontal component of a coordinate pair is referred to as *x*; the vertical component, *y*. You can see these coordinates in the Transform panel or Control when you select an object using the Selection tool. As in the InCopy user interface, coordinates are measured relative to the current location of the ruler's zero point.

There is one difference between the coordinates used in InCopy and the coordinate system used in a Geometry textbook: on the InCopy vertical (or *y*) axis, coordinates *below* the zero point are positive numbers; coordinates *above* the zero point are negative numbers.

Measurement units

When you send measurement values to InCopy, you can send numbers (for example, 14.65) or measurement strings (for example, "1p7.1"). If you send numbers, InCopy uses the publication's current units of measurement; if you send measurement strings (see the table below), InCopy uses the units of measurement specified in the string.

InCopy returns coordinates and other measurement values using the publication's current measurement units. In some cases, these units do not resemble the measurement values shown in the InCopy Transform panel. For example, if the current measurement system is picas, InCopy returns fractional values as decimals, rather than using the picas-and-points notation used by the Transform panel. So, for example, "1p6," is returned as "1.5." InCopy does this because your scripting system would have trouble trying to perform arithmetic operations using measurement strings. For instance, trying to add "0p3.5" to "13p4" produces a script error, while adding .2916 to 13.333 (the converted pica measurements) does not.

If your script depends on adding, subtracting, multiplying, or dividing specific measurement values, you might want to set the corresponding measurement units at the beginning of the script. At the end of the script, you can set the measurement units back to whatever they were before you ran the script. Alternately, you can use measurement overrides, like many of the sample scripts. A measurement override is a string containing a special character, as shown in the following table:

Override	Meaning	Example
c	ciceros (add didots after the c, if necessary)	1.4c
cm	centimeters	.635cm
i (or in)	inches	.25i
mm	millimeters	6.35mm
p	picas (add points after the p, if necessary)	1p6
pt	points	18pt

Adding features to “Hello World”

Next, we create a new script that makes changes to the “Hello World” publication we created with our first script. Our second script demonstrates how to do the following:

- Get the active document.
- Change the formatting of the text in the first story.
- Add a note.

Either open the ImprovedHelloWorld tutorial script or follow these steps to create the script:

1. Make sure you have the document you created earlier open. If you closed the document without saving it, simply run the `HelloWorld.applescript` script again to make a new document.
2. In the Script Editor, choose File > New to create a new script.
3. Enter the following code:

```
tell application "Adobe InCopy CS5"
    --Get a reference to a font.
    try
        --Enter the name of a font on your system, if necessary.
        set myFont to font "Arial"
    end try
    --Get the active document and assign the result to the variable "myDocument"
    set myDocument to document 1
    tell story 1 of myDocument
        --Change the font, size, and paragraph alignment.
        try
            set applied font to myFont
        end try
        set point size to 72
        set justification to center align
        --Enter the note at the last insertion point of the story.
        tell insertion point -1
            set myNote to make note
            set contents of text 1 of myNote to "This is a note."
        end tell
    end tell
end tell
```

4. Save the script.

Double-click the script name in the Scripts panel to run the new script.

3 Scripting Features

Chapter Update Status		
CS5.5	Unchanged	Content not guaranteed to be current.

This chapter covers scripting techniques that relate to InCopy's scripting environment. Almost every other object in the InCopy 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.
- Running scripts at InCopy start-up.

We assume that you have already read Chapter 2, "Getting Started" and know how to write, install, and run InCopy scripts in the scripting language of your choice.

Script preferences

The `script preferences` object provides objects and properties related to the way InCopy 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 that InCopy presents to the user. When you set this property to `never interact`, InCopy does not display any alerts or dialogs; set it to `interact with alerts` to enable alerts but disable dialogs; and 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, InCopy 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 24. Note that 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 InCopy 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

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

- **Targeting —** Scripts must be targeted to the version application in which they are being run (that is, 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. InCopy correctly interprets a script written for an earlier version of the scripting object model. To do this, run the script from a folder in the Scripts panel folder named `Version 4.0 Scripts` (for InCopy CS2 scripts) or `Version 3.0 Scripts` (for InCopy CS scripts) or explicitly set the application's script preferences to the old object model within the script (as shown below).

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 that launches the script.

```
--Target InCopy CS5
tell application "Adobe InCopy 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 InCopy CS5" --target CS5
    using terms from application "InCopy CS5" --compile using CS5
        --InCopy CS5 version script goes here.
    end using terms from
end tell
```

To generate a CS5 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, the `/Users/<username>/Library/Preferences/Adobe InCopy/Version 7.0/<locale>/Scripting Support/7.0` folder.

```
tell application "Adobe InCopy CS5"
    --publish the InCopy CS dictionary (version 7.0 DOM)
    publish terminology version 7.0
end tell
```

Interpretation

The InCopy 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.

```
--Set to 4.0 scripting object model
set version of script preferences to 4.0
```

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 either an AppleScript or a JavaScript; on Windows, you can run a VBScript or a 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 start-up" on page 27.

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 InCopy 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 InCopy 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

To return a value from a script executed by `do script`, you can 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 the DoScriptReturnValue tutorial script):

```
tell application "Adobe InCopy CS5"
    --Create a string to be run as an AppleScript.
    set myAppleScript to "tell application \"Adobe InCopy 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
```

Running scripts at start-up

To run a script when InCopy starts, put the script in the Startup Scripts folder in the Scripts folder (for more information, see "Installing scripts" on page 11).

4 Text and Type

Chapter Update Status		
CS5.5	Unchanged	Content not guaranteed to be current.

Entering, editing, and formatting text make up the bulk of the time spent working on most InCopy documents. As a result, automating text and type operations can result in large productivity gains.

This tutorial 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 Chapter 2, "Getting Started" and know how to create, install, and run a script. We also assume that you have some knowledge of working with text in InCopy and understand basic typesetting terms.

Entering and importing text

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

Stories and text frames

All text in an InCopy layout is part of a story, and every story can contain one or more text frames. If you are working with a standalone InCopy document, the document contains one story, and InCopy adds text frames only when necessary to display the text of the story. This also is true for stories exported from InDesign as InCopy stories (`.icml` files).

When you work with an InCopy story within an InDesign document, the document can contain any number of stories, and you will see the text frames as they were created in the InDesign layout. Unlike InDesign, InCopy cannot add new text frames using scripting.

For more on understanding the relationships between text objects in an InCopy document, see "Text objects" on page 37.

Adding text to a story

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

```
--Adds text to the default story.
tell application "Adobe InCopy CS5"
    set myDocument to make document
    tell story 1 of myDocument
        set contents to "This is the first paragraph of example text."
        --To add more text to the story, we'll use the last insertion point
        --in the story. ("return" is a return character in AppleScript.)
        set the contents of insertion point -1 to return & "This is the second
paragraph."
    end tell
end tell
```

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):

```
--Enters text in the default story and then replaces
--a word in the story with a different phrase.
tell application "Adobe InCopy CS5"
    set myDocument to make document
    tell story 1 of myDocument
        set contents to "This is some example text."
        --To add more text to the story, we'll use the last insertion point
        --in the story. ("return" is a return character in AppleScript.)
        set the contents of word 3 to "a little bit of"
    end tell
end tell
```

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

```
--Enters text in the default story and then replaces
--the text in the second paragraph.
tell application "Adobe InCopy CS5"
    set myDocument to make document
    tell story 1 of myDocument
        set contents to "Paragraph 1." & return & "Paragraph 2." & return &
        "Paragraph 3." & return
        --Replace the text in the second paragraph without replacing
        --the return character at the end of the paragraph.
        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 the preceding script, we excluded the return character in the second paragraph, because deleting the return might change the paragraph style applied to the paragraph. To do this, we supplied two characters—the first character of the paragraph, and the last character before the return character ending the paragraph—as parameters.

Inserting special characters

Because the Script Editor supports Unicode, you can simply enter Unicode characters in text strings you send to InCopy. Alternately, you can use an InCopy shortcut to explicitly enter Unicode characters by their glyph ID number: <*nnnn*> (where *nnnn* is the Unicode code for the character). The following script shows several ways to enter special characters (for the complete script, see SpecialCharacters):

```
--Shows how to enter special characters in text.
tell application "Adobe InCopy CS5"
    set myDocument to make document
    set myStory to story 1 of myDocument
    --Entering special characters directly:
    set contents of myStory to "Registered trademark: ˝" & return & "Copyright:  ©" &
return & "Trademark: ª" & 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 InCopy 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
end tell
```

The easiest way to find the Unicode ID for a character is to use the Glyphs palette in InCopy (choose Type > Glyphs to display the palette)—move the cursor over a character in the palette, and InCopy will display its Unicode value. You can find out more about Unicode by visiting http://www.unicode.org.

Placing text and setting text-import preferences

In addition to entering text strings, you can place text files created with word processors and text editors. The following script shows you how to place a text file in the default story of a new document (for the complete script, see PlaceTextFile):

```
tell application "Adobe InCopy CS5"
    set myDocument to make document
    tell insertion point -1 of story 1
        place "yukino:test.txt"
    end tell
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). Comments in the script show the possible values for each property.

```
tell application "Adobe InCopy CS5"
    tell text import preferences
        --Options for character set:
        --ansi
        --chineseBig5
        --gb18030
        --gb2312
        --ksc5601
        --macintoshCE
        --macintoshCyrillic
        --macintoshGreek
        --macintoshTurkish
        --recommendShiftJIS83pv
        --shiftJIS90ms
        --shiftJIS90pv
```

```
        --UTF8
        --UTF16
        --windowsBaltic
        --windowsCE
        --windowsCyrillic
        --windowsEE
        --windowsGreek
        --windowsTurkish
        set character set to UTF16
        set convert spaces into tabs to true
        set spaces into tabs count to 3
        --The dictionary property can take any of the following
        --language names (as strings):
        --Bulgarian
        --Catalan
        --Croatian
        --Czech
        --Danish
        --Dutch
        --English: Canadian
        --English: UK
        --English: USA
        --English: USA Legal
        --English: USA Medical
        --Estonian
        --Finnish
        --French
        --French: Canadian
        --German: Reformed
        --German: Swiss
        --German: Traditional
        --Greek
        --Hungarian
        --Italian
        --Latvian
        --Lithuanian
        --Neutral
        --Norwegian: Bokmal
        --Norwegian: Nynorsk
        --Polish
        --Portuguese
        --Portuguese: Brazilian
        --Romanian
        --Russian
        --Slovak
        --Slovenian
        --Spanish: Castilian
        --Swedish
        --Turkish
        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
end tell
```

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

```
tell application "Adobe InCopy CS5"
    tell tagged text import preferences
        set remove text formatting to false
        --style conflict property can be:
        --publication definition
        --tag file definition
        set style conflict to publication definition
        set use typographers quotes to true
    end tell
end tell
```

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

```
tell application "Adobe InCopy CS5"
    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
end tell
```

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

```
tell application "Adobe InCopy 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 InCopy document. You must use text or story objects to export in text-file formats; you cannot export all the text in a document in one operation. (For the complete script, see ExportTextFile.)

```
tell application "Adobe InCopy CS5"
	set myDocument to make document
	set myStory to story 1 of myDocument
	--Fill the story with placeholder text.
	set myTextFrame to item 1 of text containers of myStory
	set contents of myTextFrame to placeholder text
	--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 myStory
		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; see the ExportTextRange tutorial script.)

```
tell application "Adobe InCopy CS5"
    set myDocument to make document
    set myStory to story 1 of myDocument
    --Fill the story with placeholder text.
    set myTextFrame to item 1 of text containers of myStory
    set contents of myTextFrame to placeholder text
    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 "yukino:test.txt" format text type
end tell
```

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

```
tell application "Adobe InCopy CS5"
    tell text export preferences
        --Options for character set:
        --UTF8
        --UTF16
        --default platform
        set character set to UTF16
        --platform options:
        --macintosh
        --pc
        set platform to macintosh
    end tell
end tell
```

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

```
tell application "Adobe InCopy CS5"
    tell tagged text export preferences
        --Options for character set:
        --ansi
        --ascii
        --gb18030
        --ksc5601
        --shiftJIS
        --UTF8
        --UTF16
        set character set to UTF16
        --tag form options:
        --abbreviated
        --verbose
        set tag form to verbose
    end tell
end tell
```

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-markup scheme you prefer. Here is a very simple example that shows how to export InCopy text as HTML (for the complete script, see ExportHTML):

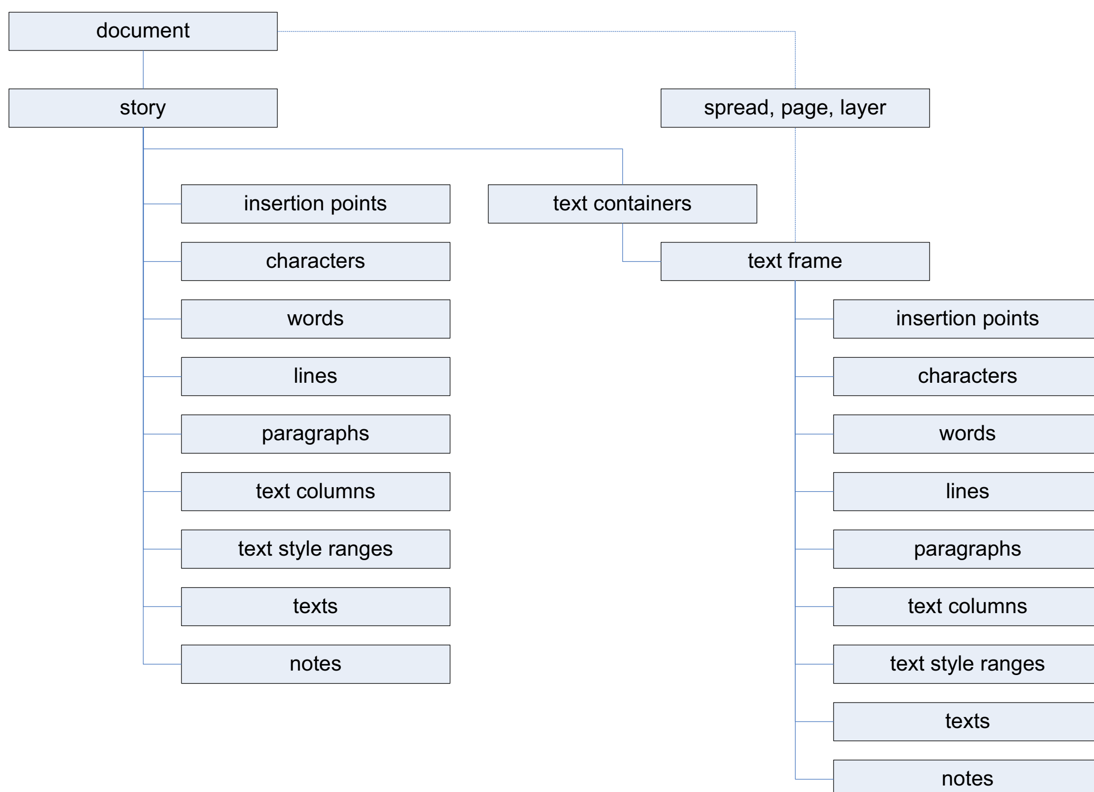
```
on mySnippet()
    tell application "Adobe InCopy CS5"
        --Use the myStyleToTagMapping array to set up your
        --paragraph style to tag mapping.
        set myStyleToTagMapping to {}
        --For each style to tag mapping, add a new item to the array.
        copy {"body_text", "p"} to 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..
                                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 myText to object reference of text from
                            character 1 to character -2 of myTextStyleRange
                            set myString to contents of myText
                        else
                            set myString to contents of myText
                        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 = 0 then
                            set myString to "<th>"
                            set myString to myString & text of cell
                            myColumnCounter of row myRowCounter of myTable
                            set myString to myString & "</th>"
                        else
                            set myString to "<td>"
                            set myString to myString & text 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
end mySnippet
on myFindTag(myStyleName, myStyleToTagMapping)
    local myTag, myDone, myStyleName
    set myTag to ""
    set myDone to false
    repeat with myStyleToTagMap in myStyleToTagMapping
```

```
        if item 1 of myStyleToTagMap = myStyleName then
            set myTag to item 2 of myStyleToTagMap
            exit repeat
        end if
    end repeat
    return myTag
end myFindTag
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
```

Text objects

The following diagram shows a view of InCopy's text-object model. There are two main types of text object: *layout* objects (text frames) and *text-stream* objects (stories, insertion points, characters, and words, for example). The diagram uses the natural-language terms for the objects; when you write scripts, you will use the corresponding terms from your scripting language:



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 a given 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.

Selections

Usually, InCopy scripts act on a text selection. The following script shows how to determine the type of the current selection. Unlike many other sample scripts, this script does not actually do anything; it simply presents a selection filtering routine you can use in your own scripts. (For the complete script, see TextSelection.)

```
tell application "Adobe InCopy CS5"
    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
            end if
        else
            display dialog ("Please select some text and try again.")
        end if
    else
        display dialog ("No documents are open.")
    end if
end tell
```

Moving and copying text

To move a text object to another location in text, use the `move` method. To copy the text, use the `duplicate` method (which has exactly the same parameters as the `move` method). The following script fragment shows how it works (for the complete script, see MoveText):

```
tell application "Adobe InCopy CS5"
    set myDocument to make document
    --Create a series of paragraphs in the default story.
    set myStory to story 1 of myDocument
    set contents of myStory to "WordA" & return & "WordB" & return &
    "WordC" & return & "WordD"
    tell myStory
        --Move WordC before WordA.
        move paragraph 3 to before paragraph 1
        --Move WordB after WordD (into the same paragraph).
        move paragraph 3 to after word 1 of paragraph -1
        --Note that moving text removes it from its original location.
    end tell
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` (for the complete script, see MoveTextBetweenDocuments):

```
tell application "Adobe InCopy CS5"
    set mySourceDocument to make document
    --Add text to the default story.
    set mySourceStory to story 1 of mySourceDocument
    set contents of mySourceStory to "This is the source text." & return &
    "This text is not the source text."
    set point size of paragraph 1 of mySourceStory to 24
    --Create a new document to move the text to.
    set myTargetDocument to make document
    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
end tell
```

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 InCopy CS5"
    set myDocument to make document
    set myStory to story 1 of myDocument
    set myString to "This is a formatted string." & return
    set myString to myString & "Text pasted after this text will retain its
    formatting." & return
    set myString to myString & "Text moved to the following line will take on the
    formatting of the insertion point." & return
    set myString to myString & "Italic: " & return
    set contents of myStory to myString
    --Apply formatting to the first paragraph.
    set font style of paragraph 1 of myStory to "Bold"
    --Apply formatting to the last paragraph.
    set font style of paragraph -1 of myStory to "Italic"
    --Copy from one frame to another using a simple copy.
    select word 1 of paragraph 1 of myStory
    copy
    select insertion point -1 of paragraph 2 of myStory
    paste
    --Copy the unformatted string from the first word to the end of the story
    --by getting and setting the contents of text objects. Note that this
    --doesn't really copy the text it replicates the text string from one text
    --location to another.
    set contents of insertion point -1 of myStory to contents of word 1 of
    paragraph 1 of myStory
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 (for the complete script, see TextIterationWrong):

```
--Shows how *not* to iterate through text.
tell application "Adobe InCopy CS5"
    set myDocument to make document
    set myString to "Paragraph 1.
Delete this paragraph.
Paragraph 2.
Paragraph 3.
Paragraph 4.
Paragraph 5.
Delete this paragraph.
Paragraph 6.
"
    set myStory to story 1 of myDocument
    set contents of myStory to myString
    --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
end tell
```

In the preceding example, some 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 beginning with "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 (from the TextIterationRight tutorial script):

```
--Shows the correct way to iterate through text.
tell application "Adobe InCopy CS5"
    set myDocument to make document
    set myString to "Paragraph 1.
Delete this paragraph.
Paragraph 2.
Paragraph 3.
Paragraph 4.
Paragraph 5.
Delete this paragraph.
Paragraph 6.
"
    set myStory to story 1 of myDocument
    set contents of myStory to myString
    --The following for loop will format all of the paragraphs by iterating
    --backwards through the paragraphs in the story.
    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
end tell
```

Formatting text

In the previous sections of this chapter, we added text to a document and worked with stories and text objects. In this section, we apply formatting to text. All the typesetting capabilities of InCopy 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.)

```
tell application "Adobe InCopy CS5"
    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
end tell
```

Fonts

The fonts collection of an InCopy application object contains all fonts accessible to InCopy. By contrast, the fonts collection of a document 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 InCopy. The following script shows the difference between application fonts and document fonts (for the complete script, see FontCollections):

```
tell application "Adobe InCopy CS5"
    set myApplicationFonts to the name of every font
    set myDocument to make document
    tell myDocument
        set myStory to story 1 of myDocument
        set myDocumentFonts to name of every font
        if (count of fonts) > 0 then
            set myDocumentFonts to name of every font
        end if
    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 myStringend tell
```

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:

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

Changing text properties

Text objects in InCopy have literally dozens of properties corresponding to their formatting attributes. Even a single 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 follows:

```
--Shows how to set all read/write properties of a text object.
tell application "Adobe InCopy CS5"
    set myDocument to make document
    set myStory to story 1 of myDocument
    set contents of myStory to "x"
    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
        --More text properties in the tutorial script.
    end tell
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):

```
my main()
on main()
    my mySetup()
    my mySnippet()
end main
on mySetup()
    tell application "Adobe InCopy CS5"
        --Create an example document.
        set myDocument to make document
        --Create a color.
        try
            set myColorA to color "DGC1_664a" of myDocument
        on error
            set myColorA to make color with properties
            {name:"DGC1_664a", model:process, color value:{90, 100, 70, 0}}
        end try
        --Create another color.
        try
            set myColorB to color "DGC1_664b" of myDocument
        on error
            set myColorB to make color with properties
            {name:"DGC1_664b", model:process, color value:{70, 0, 30, 50}}
        end try
        set myStory to story 1 of myDocument
        --Enter text in the text frame.
        set contents of myStory to "Text" & return & "Color"
    end tell
end mySetup
on mySnippet()
    tell application "Adobe InCopy CS5"
        set myDocument to document 1
        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
    end tell
end mySnippet
```

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 key to text-formatting productivity and should be a central part of any script that applies text formatting.

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

```
tell application "Adobe InCopy CS5"
    --Create an example document.
    set myDocument to make document
    tell myDocument
        --Create a color for use by one of the paragraph styles we'll create.
        try
            set myColor to color "Red"
        on error
            --The color did not exist, so create it.
            set myColor to make color with properties {name:"Red",
            model:process, color value:{0, 100, 100, 0}}
        end try
        set myStory to story 1
        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 story 1
            apply paragraph style using myParagraphStyle
            tell text from character 13 to character 54
                apply character style using myCharacterStyle
            end tell
        end tell
    end tell
end tell
```

Why use the `apply paragraph style` method instead of setting the `applied paragraph style` property of the text object? The 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):

```
tell application "Adobe InCopy CS5"
    --Create an example document.
    set myDocument to make document
    tell myDocument
        --Create a color for use by one of the paragraph styles we'll create.
        try
            set myColor to color "Red"
        on error
            --The color did not exist, so create it.
            set myColor to make color with properties
            {name:"Red", model:process, color value:{0, 100, 100, 0}}
        end try
        --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. 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
        --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 applyStyle method.)
        tell story 1
            set myTextFrame to item 1 of text containers
            set contents of myTextFrame to placeholder text
            tell text 1
                apply paragraph style using myParagraphStyle
            end tell
        end tell
    end tell
end tell
```

Deleting a style

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

```
tell myDocument
    set myParagraphStyleA to paragraph style "myParagraphStyleA"
    set myParagraphStyleB to paragraph style "myParagraphStyleB"
    --Remove the paragraph style myParagraphStyleA and replace with
    --myParagraphStyleB.
    delete myParagraphStyleA replacing with myParagraphStyleB
end tell
```

Importing paragraph and character styles

You can import paragraph and character styles from other InCopy documents. The following script fragment shows how (for the complete script, see ImportTextStyles):

```
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 InCopy 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 InCopy user interface. InCopy has three ways of searching for text:

- You can find text and text formatting and change it to other text and/or text formatting. This type of find and change operation uses the `find text preferences` and `change text preferences` objects to specify parameters for the `find text` and `change text` methods.
- You can find text using regular expressions, or "grep." This type of find and change operation uses the `find grep preferences` and `change grep preferences` objects to specify parameters for the `find grep` and `change grep` methods.
- You can find specific glyphs (and their formatting) and replace them with other glyphs and formatting. This type of find and change operation uses the`find glyph preferences` and `change glyph preferences` objects to specify parameters for the `find glyph` and `change glyph` methods.

All find and change methods take a single optional parameter, `reverse order`, 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 in "Text objects and iteration" on page 40. 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.

Find/change preferences

Before searching 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 a few find and 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 InCopy CS5"
set find text preferences to nothing
set change text preferences to nothing
end tell
--find/change grep preferences
tell application "Adobe InCopy CS5"
set find grep preferences to nothing
set change grep preferences to nothing
end tell
--find/change glyph preferences
tell application "Adobe InCopy CS5"
set find glyph preferences to nothing
set change glpyh preferences to nothing
end tell
```

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

Finding text

The following script fragment shows how to find a specified string of text. While the script fragment searches the entire document, you also can search stories, text frames, paragraphs, text columns, or any other text object. The `find text` 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 document 1
    set myFoundItems to find text
    display dialog ("Found " & (count myFoundItems) & " instances of the search
string.")
```

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

```
--Clear the find/change preferences.
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
--Search the document for the string "copy" and change it to "text".
--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"
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
```

Finding and changing formatting

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

```
--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
--Search the document for the 24 point text and change it to 10 point text.
set point size of find text preferences to 24
set point size of change text preferences to 10
tell document 1
    change text
end tell
--Clear the find/change preferences after the search.
set find text preferences to nothing
set change text preferences to nothing
```

You also can search for a string of text and apply formatting, as shown in the following script fragment (from the FindChangeStringFormatting 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 find what of find text preferences to "WIDGET^9^9^9^9"
--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 font style of change text preferences to "Bold"
--Search the document. In this example, we'll use the
--InCopy 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

InCopy supports regular expression find/change through the `findGrep` and `changeGrep` methods. Regular-expression find and change also can find text with a specified format or replace text formatting 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 document 1
    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 `find change text options` object: `whole word`and `case sensitive`. 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 markup (that is, some form of tagging plain text with formatting instructions) into InCopy formatted text. PageMaker paragraph tags (which are not the same as PageMaker tagged text-format files) are an example of a simplified text-markup scheme. In a text file marked up using this scheme, paragraph style names appear at the start of a paragraph, as shown in these examples:

```
<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 markup tags, as shown in the following script fragment (from the ReadPMTags tutorial script):

```
on myReadPMTags(myStory)
    tell application "Adobe InCopy 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 `find glyph` and `change glyph` methods and the associated `find glyph preferences` and `change glyph preferences` objects. The following scripts fragment shows how to find and change a glyph in a sample document (for the complete script, see FindChangeGlyphs):

```
--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 500
--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 "Times New RomanRegular"
set glyph ID of change glyph preferences to 374
tell myDocument
    change glyph
end tell
--Clear glyph search preferences.
set find glyph preferences to nothing
set change glyph preferences to nothing
```

Tables

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

```
set 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 tag 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):

```
set myDocument to document 1
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):

```
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 document 1
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
    --Use everyItem to set the formatting of multiple cells at once.
    tell every cell in myTable
        --myTable.cells.everyItem().topEdgeStrokeColor to
myDocument.swatches.item("DGC1_446b")
        set top edge stroke color to swatch "DGC1_446b" of myDocument
        --myTable.cells.everyItem().topEdgeStrokeWeight to 1
        set top edge stroke weight to 1
        --myTable.cells.everyItem().bottomEdgeStrokeColor to
myDocument.swatches.item("DGC1_446b")
        set bottom edge stroke color to swatch "DGC1_446b" of myDocument
        --myTable.cells.everyItem().bottomEdgeStrokeWeight to 1
```

```
        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.
        --myTable.cells.everyItem().leftEdgeStrokeColor to
myDocument.swatches.item("None")
        set left edge stroke color to swatch "None" of myDocument
        --myTable.cells.everyItem().leftEdgeStrokeWeight to 0
        set left edge stroke weight to 0
        --myTable.cells.everyItem().rightEdgeStrokeColor to
myDocument.swatches.item("None")
        set right edge stroke color to swatch "None" of myDocument
        --myTable.cells.everyItem().rightEdgeStrokeWeight to 0
        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):

```
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
    --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.)

```
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
   end if
end if
```

Autocorrect

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

```
--The autocorrect preferences object turns the
--autocorrect feature on or off.
tell application "Adobe InCopy CS5"
   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 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 {{}}
end tell
```

Footnotes

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

```
tell application "Adobe InCopy CS5"
    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
end tell
end mySnippet
--This function gets a random integer in the range myStart to myEnd.
on myGetRandom(myStart, myEnd)
    set myRange to myEnd - myStart
    set myRandomInteger to (myStart + (random number from myStart to myEnd))
    as integer
    return myRandomInteger
end myGetRandom
```

5 User Interfaces

Chapter Update Status		
CS5.5	Unchanged	Content not guaranteed to be current.

AppleScript can create dialog boxes for simple yes/no questions and text entry, but you probably will need to create more complex dialog boxes for your scripts. InCopy scripting can add dialog boxes and can 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.

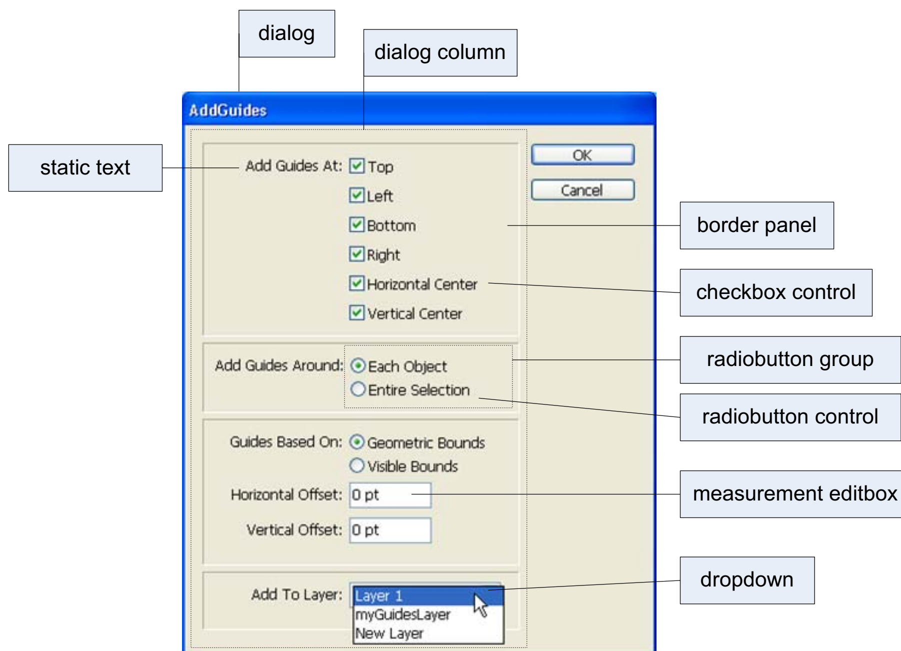
This chapter shows how to work with InCopy dialog scripting. The sample scripts in this chapter are presented in order of complexity, starting with very simple scripts and building toward more complex operations.

NOTE: InCopy scripts written in JavaScript also can include user interfaces created using the Adobe *ScriptUI* component. This chapter includes some ScriptUI scripting tutorials; for more information, see *Adobe Creative Suite® 3 JavaScript Tools Guide*.

We assume that you have already read Chapter 2, "Getting Started" and know how to create, install, and run a script.

Dialog-box overview

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



The items in the figure are defined in the following table:

Dialog-box element	**InCopy 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 InCopy scripting object, each part of a dialog box has its own properties. For example, a `checkbox control` 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 InCopy'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 InCopy dialog box

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

```
tell application "Adobe InCopy CS5"
    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
end tell
```

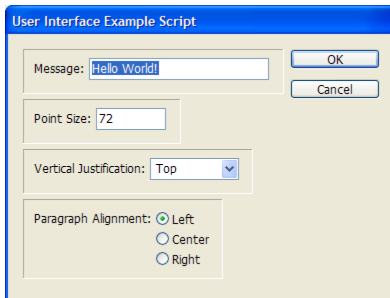
Adding a user interface to “Hello World”

In this example, we add a simple user interface to the Hello World tutorial script presented in Chapter 2, “Getting Started.” The options in the dialog box provide a way for you to specify the sample text and change the point size of the text. For the complete script, see HelloWorldUI.

```
tell application "Adobe InCopy CS5"
    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 real editbox with properties
            {edit contents:"72"}
        end tell
        show
        --Get the settings from the dialog box.
        --Get the point size from the point size field.
        set myPointSize to edit contents of myPointSizeField as real
        --Get the example text from the text edit field.
        set myString to edit contents of myTextEditField
        --Remove the dialog box from memory.
        destroy myDialog
    end tell
    set myDocument to make document
    tell story 1 of myDocument
        --Apply the settings from the dialog box to the text.
        set contents to myString
        --Set the point size of the text.
        set point size of text 1 to myPointSize
    end tell
end tell
```

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.

```
tell application "Adobe InCopy CS5"
    set myDocument to make document
    set mySwatchNames to name of every swatch of myDocument
    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 real editbox with properties
                    {edit contents:"72"}
                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
        set myBorderPanel to make border panel
        tell myBorderPanel
            make static text with properties {static label:"Text Color:"}
            set mySwatchesDropdown to make dropdown with properties
            {string list:mySwatchNames, selected index:1}
        end tell
    end tell
    show
    --Get the settings from the dialog box.
    --Get the point size from the point size field.
    set myPointSize to edit contents of myPointSizeField as real
    --Get the example text from the text edit field.
    set myString to edit contents of myTextEditField
    --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
    --Get the text color selected in the dropdown.
    set mySwatchName to item ((selected index of mySwatchesDropdown) + 1)
    of mySwatchNames
    --Remove the dialog box from memory.
    destroy myDialog
end tell
tell story 1 of myDocument
    --Apply the settings from the dialog box to the text frame.
    set contents to myString
    --Apply the paragraph alignment ("justification").
    --"text 1 of myStory" is all of the text in the text story.
    set justification of text 1 to myParagraphAlignment
    --Set the point size of the text in the text frame.
    set point size of text 1 to myPointSize
    --Set the fill color of the text to the selected swatch.
    set fill color of text 1 to swatch mySwatchName of myDocument
end tell
end tell
```

Working with ScriptUI

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

This does not mean, however, that user-interface elements written using Script UI are not accessible to AppleScript users. InCopy 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 how to use the progress bar from an AppleScript (for the complete script, see ProgressBar):

```
#targetengine "session"
var myProgressPanel;
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 preceding script using an AppleScript (for the complete script, see CallProgressBar):

```
tell application "Adobe InCopy CS5"
    set myDocument to make document
    --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);"
    set myJavaScript to myJavaScript & return
    set myJavaScript to myJavaScript & "myProgressPanel.show();" & return
    do script myJavaScript language javascript
    repeat with myCounter from 1 to 100
        set myJavaScript to "#targetengine \"session\"" & return
        set myJavaScript to myJavaScript & "myProgressPanel.myProgressBar.value"
        set myJavaScript to myJavaScript & "=" & myCounter & "/myIncrement;"
        set myJavaScript to myJavaScript & return
        do script myJavaScript language javascript
        tell insertion point -1 of story 1 of myDocument
            set contents to "x"
        end tell
        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
end tell
```

Creating a button-bar panel with ScriptUI

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

The following tutorial script shows how to create a simple floating panel. The panel can contain a series of buttons, with each button being associated with a script stored on disk. Click the button, and the panel

runs the script (the script, in turn, can display dialog boxes or other user-interface elements). The button in the panel can contain text or graphics. (For the complete script, see ButtonBar.)

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

```
<buttons>
    <button>
        <buttonType></buttonType>
        <buttonName></buttonName>
        <buttonFileName></buttonFileName>
        <buttonIconFile></buttonIconFile>
    </button>
    ...
</buttons>
```

For example:

```
<buttons>
    <button>
        <buttonType>text</buttonType>
        <buttonName>FindChangeByList</buttonName>
        <buttonFileName>/c/buttons/FindChangeByList.jsx</buttonFileName>
        <buttonIconFile></buttonIconFile>
    </button>
    <button>
        <buttonType>text</buttonType>
        <buttonName>SortParagraphs</buttonName>
        <buttonFileName>/c/buttons/SortParagraphs.jsx</buttonFileName>
        <buttonIconFile></buttonIconFile>
    </button>
</buttons>
```

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

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

```
                    }
                    else{
                        myButton = add('iconbutton', undefined,
                        File(myButtonIconFile));
                    }
                    myButton.scriptFile = myButtonFileName;
                    myButton.onClick = function(){
                        myButtonFile = File(this.scriptFile)
                        app.doScript(myButtonFile);
                    }
                }
            }
        }
    }
    return myButtonBar;
}
function myReadXMLPreferences(){
    myXMLFile = File.openDialog("Choose the file containing your
    button bar defaults");
    var myResult = myXMLFile.open("r", undefined, undefined);
    var myButtons = "";
    if(myResult == true){
        var myXMLDefaults = myXMLFile.read();
        myXMLFile.close();
        var myXMLDefaults = new XML(myXMLDefaults);
        var myButtons = myXMLDefaults.xpath("/buttons/button");
    }
    return myButtons;
}
```

6 Menus

Chapter Update Status		
CS5.5	Unchanged	Content not guaranteed to be current.

InCopy 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 InCopy menu scripting. The sample scripts in this chapter are presented in order of complexity, starting with very simple scripts and building toward more complex operations.

We assume that you have read Chapter 2, "Getting Started" and know how to create, install, and run a script.

Understanding the menu model

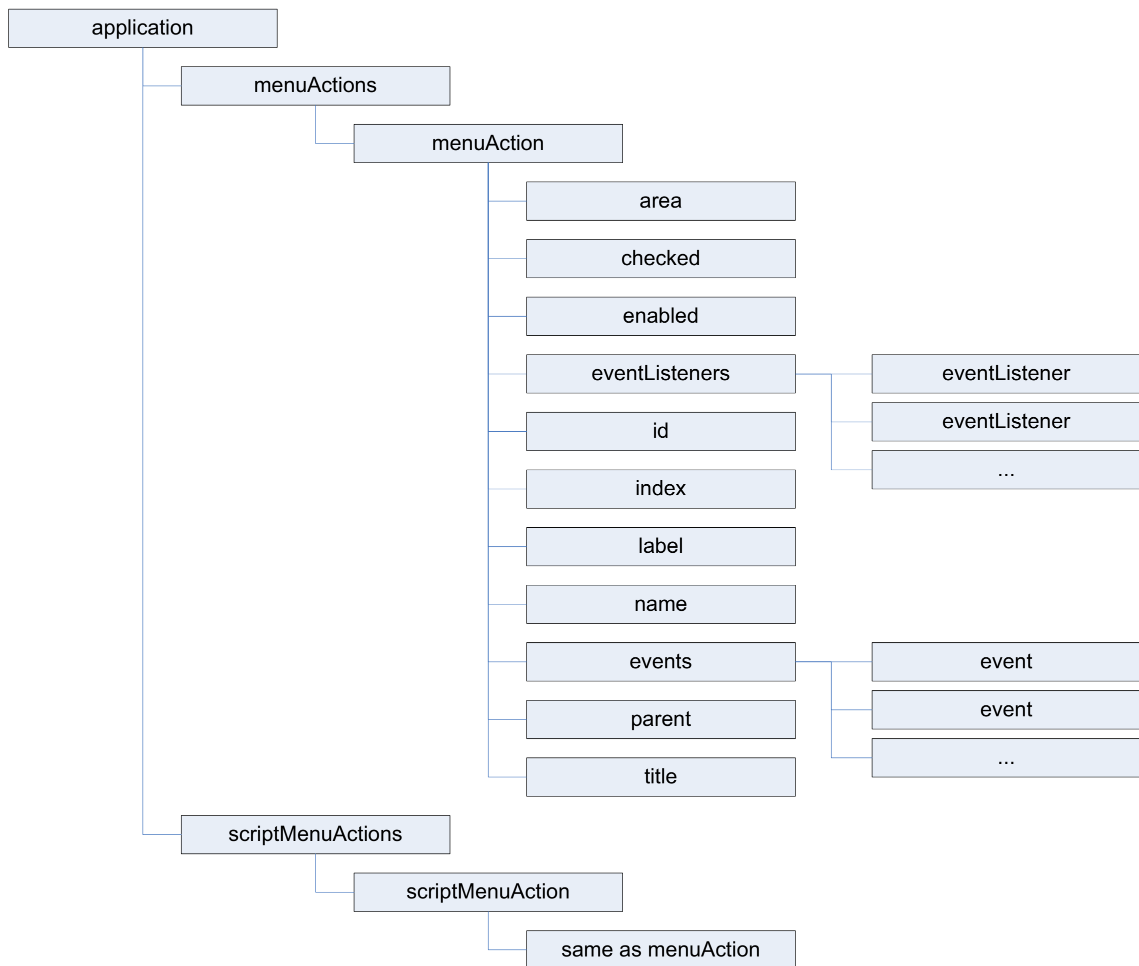
The InCopy menu-scripting model is made up of a series of objects that correspond to the menus you see in the application's user interface, including menus associated with panels as well as those displayed on the main menu bar. A `menu` object contains the following objects:

- `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, InCopy 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 visible 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 InCopy 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 listing, refer to the GetMenuNames tutorial script). Note that these scripts can be very slow, as there are a large number of menu names in InCopy.

```
--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 InCopy 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 the 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 InCopy CS5"
    set myMenuAction to menu action "$ID/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 `get 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 InCopy.

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

```
tell application "Adobe InCopy CS5"
    set myString to translate key string "$ID/Convert to Note"
    display dialog(myString)
end tell
```

Running a menu action from a script

Any of InCopy'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. 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 InCopy CS5"
    --Get a reference to a menu action.
    set myMenuAction to menu action "$ID/Convert to Note"
    --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 InCopy processes by scripting menu actions and user-interface selections; InCopy'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 InCopy 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 InCopy 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 InCopy 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 added by the preceding script, run the RemoveSpecialFontMenu script.

```
set myMainMenu to menu item "Main"
set mySpecialFontMenu to submenu "Kozuka Mincho Pro" of myMainMenu
tell mySpecialFontMenu to delete
```

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 7, “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 script menu actions

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

```
tell application "Adobe InCopy 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 preceding script, run the following script fragment (from the RemoveScriptMenuAction tutorial script):

```
tell application "Adobe InCopy 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 InCopy 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 GetScriptMenuActions 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 InCopy 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` (for example, 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 InCopy 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 InCopy 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 InCopy 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
```

7 Events

Chapter Update Status		
CS5.5	Unchanged	Content not guaranteed to be current.

InCopy scripting can respond to common application and document events, like opening a file, creating a new file, printing, and importing text and graphic files from disk. In InCopy 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, as the corresponding event occurs, rather than being run by the user (from the Scripts palette).

This chapter shows how to work with InCopy 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 Chapter 2, "Getting Started" and know how to create, install, and run a script.

This chapter covers application and document events. For a discussion of events related to menus, see Chapter 6, "Menus."

The InCopy 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 InCopy event scripting model is made up of a series of objects that correspond to the events that occur as you work with the application. The first object is the `event`, which corresponds to one of a limited series of actions in the InCopy 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).

The following table shows a list of events to which `event listeners` can respond. These events can be triggered by any available means, including menu selections, keyboard shortcuts, or script actions.

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

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

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 three 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.
- **Capturing** — The event starts at the top of the scripting object model—the application—then propagates through the model to the target of the event. Any `event listeners` capable of responding to the event registered to objects above the `target` will process the event.
- **Bubbling** — The event starts propagation at its `target` and triggers any qualifying `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.
`Captures`	If true, the `event` may be handled by `event listeners` registered to scripting objects above the target object of the event during the capturing phase of event propagation. This means an `event listener` on the application, for example, can respond to a document event before an `event listener` is triggered.
`CurrentTarget`	The current scripting object processing the `event`. See `target` in this table.
`DefaultPrevented`	If true, the default behavior of the `event` on the current `target` (see `target` in this table) was prevented, thereby cancelling the action.
`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 the `event` occurred.

Working with eventListeners

When you create an `event listener`, you specify the event type (as a string) the event handler (as a file reference), and whether the `event listener` can be triggered in the capturing phase of the event. 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 InCopy CS5"
    make event listener with properties {event type:"afterNew",
handler:"yukino:ICEventHandlers:Message.applescript", captures:true}
end tell
```

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

```
tell application "Adobe InCopy 3.0"
    --"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
```

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

```
tell application "Adobe InCopy CS5"
    remove event listener event type "afterNew" handler file
"yukino:IDEventHandlers:Message.applescript" without captures
end tell
```

When an `event listener` responds 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 InCopy session. To make an `event listener` available in every InCopy session, add the script to the startup scripts folder (for more on installing scripts, see Chapter 2, "Getting Started."). When you add an `event listener` script to a document, it is not saved with the document or exported to INX.

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

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 InCopy 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:"yukino:EventInfo.applescript", captures:true}
    tell myDocument
        make event listener with properties {event type:"beforeImport",
        handler:"yukino:EventInfo.applescript", captures:true}
    end tell
end tell
```

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

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

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

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

```
--Turns on all event listeners on the application object.
tell application "Adobe InCopy 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:GetEventInfo.applescript", captures:false}
    end repeat
end tell
```

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

```
main(evt)
on main(myEvent)
    tell application "Adobe InCopy 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 & "Captures: " & captures of 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 InCopy 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 capturing then
            set myString to "Capturing"
        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` for the application object. For the complete script, see EventListenersOff.

```
--EventListenersOff.applescript
--An InCopy CS5 AppleScript
--
--Turns off all of the event listeners on the application object.
tell application "Adobe InCopy CS5"
   tell event listeners to delete
end tell
```

A sample “afterNew” eventListener

The `afterNew` event provides a convenient place to add information to the document, like user name, document creation date, copyright information, and other job-tracking information. The following sample script shows how to add this sort of information to document metadata (also known as file info or XMP information). For the complete script listing, refer to the AfterNew tutorial script.

```
tell application "Adobe InCopy CS5"
   make event listener with properties {event type:"afterNew",
handler:"yukino:AfterNewHandler.applescript", captures:true}
end tell
```

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

```
--AfterNewHandler.applescript
--An InCopy CS5 AppleScript
--
main(evt)
on main(myEvent)
   tell application "Adobe InCopy CS5"
      set user name to "Adobe"
      set myDocument to document 1
      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
```

8 Notes

Chapter Update Status		
CS5.5	Unchanged	Content not guaranteed to be current.

With the InDesign and InCopy inline editorial-notes features, you can add comments and annotations as notes directly to text without affecting the flow of a story. Notes features are designed to be used in a workgroup environment. Notes can be color coded or turned on or off based on certain criteria.

Notes can be created using the Note tool in the toolbox, the Notes > New Note command, or the New Note icon on the Notes palette.

We assume that you have 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 notes in InCopy.

Entering and importing a note

This section covers the process of getting a note into your InCopy document. Just as you can create a note and replace the text of the note using the InCopy user interface, you can create notes and insert text into a note using scripting.

Adding a note to a story

To add note to a story, use the `add` method. The following sample adds a note at the last insertion point. For the complete script, see InsertNote.

```
tell application "Adobe InCopy CS5"
    set myStory to story 1 of active document
    tell myStory
        --Add note to a default story. We'll use the last insertion point in the story.
        tell insertion point -1
            --Add text to the note
            set myNote to make note
            set contents of text 1 of myNote to "This is a note."
        end tell
    end tell
end tell
```

Replacing text of a note

To replace the text of a note, use the `contents` property, as shown in the following sample. For the complete script, see Replace.

```
tell application "Adobe InCopy CS5"
    set myStory to story 1 of active document
    set myNote to note 1 of myStory
    -- We'll use the last insertion point in the story.
    tell insertion point -1
        --Add text to the note
        set contents of text 1 of myNote to "This is a note."
    end tell
end tell
```

Converting between notes and text

Converting a note to text

To convert a note to text, use the `convert to text` method, as shown in the following sample. For the complete script, see ConvertToText.

```
tell application "Adobe InCopy CS5"
    set myStory to story 1 of active document
    set myNote to note 1 of myStory
    --Convert the note to text
    tell myNote
        convert to text
    end tell
end tell
```

Converting text to a note

To convert text to a note, use the `convert to note` method, as shown in the following sample. For the complete script, see ConvertToNote.

```
tell application "Adobe InCopy CS5"
    --Select a text
    set myStory to story 1 of active document
    select word 1 of myStory
    tell selection
        -- Convert the text to a note
        set myWords to convert to note
    endtell
end tell
```

Expanding and collapsing notes

Collapsing a note

The following script fragment shows how to collapse a note. For the complete script, see CollapseNote.

```
tell application "Adobe InCopy CS5"
    set myStory to story 1 of active document
    set myNote to note 1 of myStory
    --Collapse a note
    set collapsed of myNote to true
end tell
```

Expanding a note

The following script fragment shows how to expand a note. For the complete script, see ExpandNote.

```
tell application "Adobe InCopy CS5"
    set myStory to story 1 of active document
    set myNote to note 1 of myStory
    --Expand a note
    set collapsed of myNote to false
end tell
```

Removing a note

To remove a note, use the `delete` method, as shown in the following sample. For the complete script, see RemoveNote.

```
tell application "Adobe InCopy CS5"
    set myStory to story 1 of active document
    set myNote to note 1 of myStory
    --Remove the note
    delete myNote
end tell
```

Navigating among notes

Going to the first note in a story

The following script fragment shows how to go to the first note in a story. For the complete script, see FirstNote.

```
tell application "Adobe InCopy CS5"
    set myStory to story 1 of active document
    set myNotes to notes of myStory
    set FirstNote to first item of myNote
    --Add text to the first note
    set contents of text 1 of FirstNote to "This is the first note."
end tell
```

Going to the next note in a story

The following script fragment shows how to go to the next note in a story. For the complete script, see NextNote.

```
tell application "Adobe InCopy CS5"
    set myStory to story 1 of active document
    -- current Note
    set myCount to 1
    set myNote to note myCount of myStory
    set myCount to myCount + 1
    set myNextNote to note 2 of myStory
    --Add text to the next note
    set contents of text 1 of myNextNote to "This is the next note."
end tell
```

Going to the previous note in a story

The following script fragment shows how to go to the previous note in a story. For the complete script, see PreviousNote.

```
tell application "Adobe InCopy CS5"
    set myStory to story 1 of active document
    -- current Note
    set myCount to 2
    set myNote to note myCount of myStory
    -- previous Note
    set myCount to myCount - 1
    set myPrevNote to note myCount of myStory
    --Add text to the previous note
    set contents of text 1 of myPrevNote to "This is the previous note."
end tell
```

Going to the last note in a story

The following script fragment shows how to go to the last note in a story. For the complete script, see LastNote.

```
tell application "Adobe InCopy CS5"
    set myStory to story 1 of active document
    set myNotes to notes of myStory
    set LastNote to last item of myNotes
    --Add text to the last note
    set contents of text 1 of LastNote to "This is the last note."
end tell
```

9 Tracking Changes

Chapter Update Status		
CS5.5	Unchanged	Content not guaranteed to be current.

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

We assume that you have already read Chapter 2, "Getting Started" and know how to create, install, and run a script. We also assume that you have some knowledge of working with text in InCopy 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 scripts show how to navigate the tracked changes.

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

```
tell application "Adobe InCopy 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:

```
tell application "Adobe InCopy 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:

```
tell application "Adobe InCopy 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:

```
--Shows how to get track change informations.
tell application "Adobe InCopy 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:

```
tell application "Adobe InCopy 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
```

10 Assignments

Chapter Update Status		
CS5.5	Unchanged	Content not guaranteed to be current.

An *assignment* is a container for text and graphics in an InDesign file that can be viewed and edited in InCopy. Typically, an assignment contains related text and graphics, such as body text, captions, and illustrations that make up a magazine article. Only InDesign can create assignments and assignment files.

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

We assume that you have already read Chapter 2, "Getting Started" and know how to create, install, and run a script.

Assignment object

The section shows how to work with assignments and assignment files. Using scripting, you can open the assignment file and get assignment properties.

Opening assignment files

The following script shows how to open an existing assignment file:

```
tell application "Adobe InCopy CS5"
    set myDesktopFolder to path to desktop as string
    set myFile to myDesktopFolder & "a.icma"
    --Opens an existing document. You'll have to fill in your own file path
    --in the variable "myFile".
    set myDocument to open myFile
    tell myDocument
        set myAssignement to assignment 1
        end tell
end tell
```

Iterating through assignment properties

The following script fragment shows how to get assignment properties, such as the assignment name, user name, location of the assignment file, and export options for the assignment.

```
tell application "Adobe InCopy CS5"
    set myDocument to document 1
    tell myDocument
        set myAssignement to assignment 1
        tell myAssignement
            set myuserName to user name
            set myFilePath to file path
            set myDocPath to document path
            set myFramecolor to frame color
            set myincludeLinksWhenPackage to include links when package
            -- Export options for assignment files.
            -- ASSIGNED_SPREADS: Exports only spreads with assigned frames
            -- EMPTY_FRAMES: Exports frames but does not export content
            -- EVERYTHING: Exports the entire document.
            set myExportOptions to export options
        end tell
    end tell
end tell
```

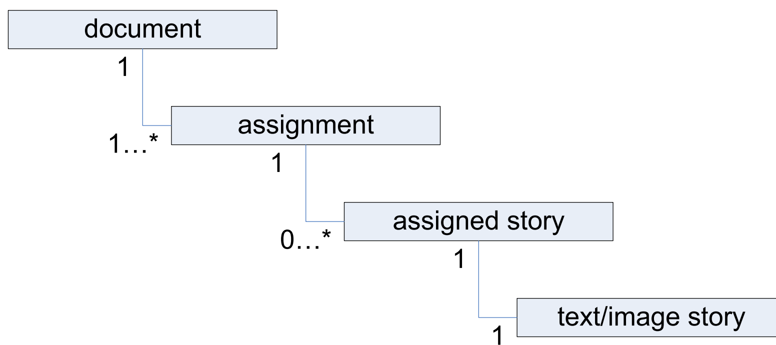
Assignment packages

Assignment packages (`.incp` files created by InCopy) are compressed folders that contain assignment files. An assignment can be packaged using the `createPackage` method. The following sample script uses this technique to create a package file:

```
tell application "Adobe InDesign CS5"
    set myDocument to document 1
    tell myDocument
        set myAssignement to assignment 1
        tell myAssignement
            if packaged is false then
                set myDesktopFolder to path to desktop as string
                set myFile to myDesktopFolder & "b.icap"
                create package myFile with FORWARD_PACKAGE
            end if
        end tell
    end tell
end tell
```

An assignment story

The following diagram shows InCopy's assignment object model. An assignment document contains one or more assignments; an assignment contains zero, one, or more assigned stories. Each assigned story references a text story or image story.



This section covers the process of getting assigned stories and assignment story properties.

Assigned-story object

The following script shows how to get an assigned story from an assignment object:

```
tell application "Adobe InCopy CS5"
    set myDocument to document 1
    tell myDocument
        set myAssignement to assignment 1
        tell myAssignement
            set myAssignmentStory to assigned story 1
        end tell
    end tell
end tell
```

Iterating through the assigned-story properties

In InCopy, assigned-story objects have properties. The following script shows how to get all properties of an assigned-story object:

```
tell application "Adobe InCopy CS5"
    set myDocument to document 1
    tell myDocument
        set myAssignement to assignment 1
        tell myAssignement
            set myAssignmentStory to assigned story 1
            tell myAssignmentStory
                set myName to name
                set myFilePath to file path
                set myStoryReference to story reference
            end tell
        end tell
    end tell
end tell
```

11 XML

Chapter Update Status		
CS5.5	Unchanged	Content not guaranteed to be current.

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. InCopy includes a complete set of features for importing XML data into page layouts, and these features can be controlled using scripting.

We assume that you have already read Chapter 2, “Getting Started” and know how to create, install, and run a script. We also assume that you have some knowledge of XML, DTDs, and XSLT.

Overview

Because XML is entirely concerned with content and explicitly *not* concerned with formatting, making XML work in a page-layout context is challenging. InCopy’s approach to XML is quite complete and flexible, but it has a few limitations:

- Once XML elements are imported into an InCopy document, they become InCopy elements that correspond to the XML structure. *The InCopy 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 depends largely 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 InCopy

You might want to do most of the work on an XML file outside InCopy, before importing the file into an InCopy layout. Working with XML outside InCopy, you can use a wide variety of excellent tools, like 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.

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 story" on page 104.

Setting XML preferences

You can control the appearance of the InCopy structure panel using the XML view-preferences object, as shown in the following script fragment (from the XMLViewPreferences tutorial script):

```
tell application "Adobe InCopy 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 InCopy 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 InCopy 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:completeDocument.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):

```
tell myDocument
    set myXMLTag to make xml tag with properties{name:"xml_element"}
    set myXMLElement to make XML elemement with properties{markup tag:myXMLTag]
    --Import into the new XML element.
    tell myXMLElement
        import from "yukino:completeDocument.xml"
    end tell
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 ImportXMLIntoSelectedXMLElement tutorial script):

```
tell myDocument
    set myXMLTag to make XML tag with properties{name: "xml_element"}
    tell XML element 1
        set myXMLElement to make XML element with properties{markup tag:
        myXMLTag}
    end tell
    tell XML import preferences
        set import to selected to true
    end tell
    select myXMLElement
    import XML from "yukino:text.xml"
end tell
```

Creating an XML tag

XML tags are the names of XML elements that 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 InCopy 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 importing the XML data, as shown in the following script fragment (from the LoadXMLTags tutorial script):

```
tell myDocument
    load xml tags from "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):

```
save XML tags to "yukino:xml_tags.xml" version comments "Tag set created October 5, 2006"
```

Creating an XML element

Ordinarily, you create XML elements by importing an XML file, but you also can create an XML element using InCopy scripting, as shown in the following script fragment (from the CreateXMLElement tutorial script):

```
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
```

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 InCopy 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
        --Place the root XML element so that you can see the result.
        tell story 1
            place XML using myRootElement
        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 myRootXMLElement 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 InCopy 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
        --Place the root XML element so that you can see the result.
        tell story 1
            place XML using myRootElement
        end tell
    end tell
end tell
```

Removing items from the XML structure

To break the association between a text object 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.)

```
tell XML element -2 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 myRootXMLElement
    set myXMLElement to make XML element with properties{markup tag:myXMLTag}
    tell myXMLElement
        make xml comment with properties{contents:"This is an XML comment."}
    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 InCopy but can be inserted in an InCopy 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 of an XML processing instruction:

```
<?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 myRootXMLElement
    make xml processing instruction with properties{target:"xml-stylesheet
type=\"text/css\".", data:"href=\"generic.css\""}
end tell
```

Working with XML attributes

XML attributes are "metadata" that can be associated with an XML element. To add an attribute to an element, use something like the following script fragment. 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").

The following script fragment shows how to add an XML attribute to an XML element (for the complete script, see MakeXMLAttribute):

```
tell application "Adobe InCopy 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 an 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 the 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 the ConvertElementToAttribute tutorial script):

```
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
```

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 InCopy 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:"A"}
        set myXMLElementB to make XML element with properties
        {markup tag:myXMLTag, contents:"B"}
        set myXMLElementC to make XML element with properties
        {markup tag:myXMLTag, contents:"C"}
        set myXMLElementD to make XML element with properties
        {markup tag:myXMLTag, contents:"D"}
    end tell
    tell myXMLElementB
        set myAttribute to make XML attribute with properties
        {name:"example_attribute", value:"This is an XML attribute."}
    end tell
    tell myAttribute
        convert to element located element end markup tag myXMLTag
    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):

```
set myXMLStory to xml story 1 of myDocument
set the point size of text 1 of myXMLStory to 72
```

Exporting XML

To export XML from an InCopy document, export either the entire XML structure in the document or one XML element (including any child XML elements it contains). The following script fragment shows how to do this (for the complete script, see ExportXML):

```
export to "yukino:test.xml" format "XML"
```

Adding XML elements to a story

Previously, we covered the process of getting XML data into InCopy documents and working with the XML structure in a document. In this section, we discuss techniques for getting XML information into a story and applying formatting to it.

Associating XML elements with text

To associate 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 myDocument
    tell story 1
        plac XML using myRootXMLElement
    end tell
end tell
```

To associate an existing text object with an existing XML element, use the `markup` method. This merges the content of the text object 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 application "Adobe InCopy CS5"
    set myDocument to make document
    tell myDocument
        set myXMLTag to make XML tag with properties {name:"myXMLElement"}
        set myRootXMLElement to XML element 1
        --Place the XML root element so that you can see
        --the result in the Structure panel.
        tell story 1
            place XML using myRootXMLElement
            set myString to "This is the first paragraph in the story.\r"
            set myString to myString & "This is the second paragraph in
            the story.\r"
            set myString to myString & "This is the third paragraph in
            the story.\r"
            set myString to myString & "This is the fourth paragraph in
            the story.\r"
            set contents to myString
        end tell
        tell myRootXMLElement
            set myXMLElement to make XML element with properties
            {markup tag:myXMLTag}
        end tell
        tell paragraph 3 of story 1
            markup using myXMLElement
        end tell
    end tell
end tell
```

Inserting text in and around XML text elements

When you place XML data into an InCopy story, 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 InCopy 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
        tell story 1
            place XML using myRootXMLElement
        end tell
    end tell
end tell
```

Mapping tags to styles

One of the quickest ways to apply formatting to XML text elements is to use `xml import maps`, also known as tag-to-style-mappings. When you do this, you can associate a specific XML tag with a paragraph or character style. When you use the `map tags to styles` method of the document, InCopy applies the style to the text, as shown in the following script fragment (from the MapTagsToStyles tutorial script):

```
tell application "Adobe InCopy 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"}
        --Apply the xML tag to style mapping
        map XML tags to styles
    end tell
    --Place the story so that you can see the result of the change.
    tell story 1 of myDocument
        place XML using XML element 1 of myDocument
    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 maps` objects to create the links between XML tags and styles, then use the `map styles to tags` method to create the corresponding XML elements, as shown in the following script fragment (from the MapStylesToTags tutorial script):

```
tell application "Adobe InCopy 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 xml 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 InCopy 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
        --Apply the style to XML tag mapping.
        map styles to XML tags
    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 the methods `apply paragraph style` and `apply character style`. (For the complete script, see ApplyStylesToXMLElements.)

```
tell application "Adobe InCopy 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
  --Place the story so that you can see the result of the change.
  tell story 1 of myDocument
    place XML using myRootXMLElement
  end tell
end tell
```

Working with XML tables

InCopy automatically imports XML data into table cells when the data is marked up using HTML standard table tags. If you cannot or prefer not to use the default table mark-up, InCopy 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 InCopy 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
        --Place the story so that you can see the result of the change.
        tell story 1 of myDocument
            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 `applyTableStyle` and `applyCellStyle` methods, as shown in the following script fragment (from the ApplyTableStyle tutorial script):

```
tell application "Adobe InCopy 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"}
        --Create a table style and a cell style.
        set myTableStyle to make table style with properties
        {name:"myTableStyle", start row fill color:color "Black",
        start row fill tint:25, end row fill color:color "Black",
        end row fill tint:10}
        set myCellStyle to make cell style with properties
        {name:"myCellStyle", fill color:color "Black", fill tint:45}
        --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
                set myTable to convert element to table row tag myRowTag
                cell tag myCellTag
            end tell
        end tell
        set myTableXMLElement to XML element 1 of XML element 1
        tell myTableXMLElement
            apply table style using myTableStyle with clearing overrides
            tell XML element 1 to apply cell style using myCellStyle
            tell XML element 4 to apply cell style using myCellStyle
            tell XML element 9 to apply cell style using myCellStyle
            tell XML element 14 to apply cell style using myCellStyle
            tell XML element 15 to apply cell style using myCellStyle
            tell XML element 20 to apply cell style using myCellStyle
        end tell
    end tell
    set alternating fills of myTable to alternating rows
    --Place the story so that you can see the result of the change.
    tell story 1 of myDocument
        place XML using XML element 1 of myDocument
    end tell
end tell
```