# Contents

## 1 Introduction .............................................. 5
- Is scripting hard to learn? .................................. 5
- Why use scripting? .......................................... 5
- How do I know when to use scripting? ....................... 5
- What about actions or macros? .............................. 5
- What exactly is scripting? .................................. 6
  - AppleScript ................................................. 6
  - VBScript .................................................. 6
  - JavaScript ................................................ 6
- How do I begin? ............................................ 7
  - AS .......................................................... 7
  - JS .......................................................... 8
  - VBS ....................................................... 8

## 2 Scripting Basics ............................................ 9
- The building blocks of scripting ............................. 9
- Understanding objects, properties, methods, and commands ........................................... 9
- Using Objects ............................................... 9
  - DOM Concepts ............................................ 9
  - Variables ................................................ 10
    - Object references make life better .................... 11
    - Variables provide a nice shortcut ..................... 12
    - Naming variables ....................................... 12
  - Object collections or elements as object references .................................................. 13
    - How elements and collections number subsequent items ........................................ 14
    - Referring to the current or active object .......... 14
- Using properties ............................................. 16
  - AS .......................................................... 17
  - JS .......................................................... 18
  - VBS ....................................................... 18
  - Understanding read-only and read-write properties .................................................. 19
    - Using alert boxes to show a property’s value 19
  - Constant values and enumerations ........................ 20
  - AS .......................................................... 21
  - JS .......................................................... 21
  - VBS ....................................................... 22
  - Using variables for property values ..................... 22
- Using methods or commands ................................ 23
  - Command or method parameters .......................... 23
    - Required parameters .................................... 23
    - Multiple parameters ...................................... 24
  - Tell statements (AS only) ................................ 25

---

3 Scripting Basics ............................................ 9

4 VBS .......................................................... 18

5 JS .......................................................... 21

6 AS .......................................................... 21
1 Introduction

Scripting is a powerful tool that can be used to control and automate many features of many Adobe® Creative Suite® applications—saving you so much time and effort that it can completely change the way you approach your work.

Is scripting hard to learn?

Scripting is not programming. You do not need a degree in computer science or mathematics to write basic scripts that automate a wide variety of common tasks.

Each scripting item corresponds to a tool or a palette or menu item in an Adobe application. In other words, each scripting element is something you already know through your Adobe expertise. If you know what you'd like your Adobe applications to do, you can easily learn to write scripts.

Why use scripting?

Your work is characterized by creativity, but many of the actual hands-on tasks are anything but creative. Most likely, you spend a lot of time doing the same or similar procedures over and over again.

Wouldn't it be great to have an assistant—one that happily does the mind-numbing tasks, follows your instructions with perfect and predictable consistency, is available any time you need help, works at lightning speed, and never even sends an invoice?

Scripting can be that assistant. With a small investment of time, you can learn to script the simple but repetitive tasks that eat up your time. However, while it's easy to get started, modern scripting languages provide the necessary depth to handle very sophisticated jobs. As your scripting skills grow, you may move on to more complex scripts that work all night while you're sleeping.

How do I know when to use scripting?

Think about your work—is there a repetitive task that's driving you crazy? If so, you've identified a candidate for a script. Next, you simply figure out:

- What are the steps involved in performing the task?
- What are the conditions in which you need to do the task?

Once you understand the process you go through to perform the task manually, you are ready to turn it into a script.

What about actions or macros?

If you have used Actions or written macros, you have some idea of the efficiency of using scripts. But scripting goes beyond the capability of Actions or macros by allowing you to manipulate multiple documents and multiple applications in a single script. For example, you can write a script that manipulates an image in Adobe Photoshop® software and then tells Adobe InDesign® software to incorporate the image.
Additionally, your script can very cleverly get and respond to information. For example, you may have a document that contains photos of varying sizes. You can write a script that figures out the size of each photo and creates a different colored border based on the size, so that icons have blue borders, small illustrations have green borders, and half-page pictures have silver borders.

If you like using Actions, keep in mind that your script can execute Actions within the application.

What exactly is scripting?

A script is a series of statements that tells an application to perform a set of tasks.

The trick is writing the statements in a language that the applications understand. Scriptable Adobe applications support several scripting languages.

If you work in Mac OS®, your choices are:

- AppleScript
- JavaScript

If you work in Windows®, your choices are:

- VBScript (Visual Basic and VBA will also work)
- JavaScript

The brief descriptions below can help you decide which language will work best for you.

AppleScript

AppleScript is a “plain language” scripting language developed by Apple. It is considered one of the simplest scripting languages to use.

To write AppleScript scripts, you can use Apple's Script Editor application, which, in a default Mac OS installation, is located at:

`system drive:Applications:AppleScript:Script Editor`

For information about using the Script Editor, see the Script Editor Help.

VBScript

VBScript is a scaled-down version of the Visual Basic programming language developed by Microsoft. VBScript talks to host applications using ActiveX Scripting. While VBScript is the Visual Basic language version officially supported by CS6, you can also write scripts in VBA and Visual Basic itself.

You can find several good VBScript editors on the Internet. If you have any Microsoft Office applications, you can also use the built in Visual Basic editor by selecting `Tools > Macro > Visual Basic Editor`.

JavaScript

JavaScript is a very common scripting language developed originally to make Web pages interactive. Like AppleScript, JavaScript is easy to learn.
**NOTE:** Adobe has developed an extended version of JavaScript, called ExtendScript, that allows you to take advantage of certain Adobe tools and scripting features. As a beginner, the difference between these two languages will not affect you. However, you should get in the habit of giving your JavaScript scripts a `.jsx` extension, rather than the usual `.js` extension.

JavaScript has some small advantages over AppleScript and Visual Basic:

- Your scripts can be used in either Windows or Mac OS. If there is a chance you'll want to share or use your scripts on both platforms, you should learn to use JavaScript.

- In Adobe Illustrator® software and InDesign, you can access scripts in any of the supported languages from within the application. However, in Photoshop, you can access only `.jsx` files from within the application. You must run AppleScript or Visual Basic scripts from outside the application. This is not a major drawback, but it does require a few extra mouse clicks to run your scripts.

- You can set up `.jsx` scripts to run automatically when you open the application by placing the scripts in the application's Startup Scripts folder. For information on startup-script folders, see the scripting guide for your application.

To write scripts in JavaScript, you can use any text editor, or you can use the ESTK (ExtendScript Tool Kit) provided with your Adobe applications. The ESTK has many features that make it easier to use than a text editor, including a built-in syntax checker that identifies where the problems are in your script and tries to explain how to fix them, and the ability to run your scripts right from the ESTK without saving the file. This second feature can save you a lot of time, especially in the beginning when you may have to test and edit a script more than a few times to get it to work.

In a default Adobe installation, the ESTK is in the following location:

**Mac OS:** `system drive:Applications:Utilities:Adobe Utilities - CS6:ExtendScript Toolkit CS6`

**Windows:** `drive:/Program Files/Adobe/Adobe Utilities - CS6/ExtendScript Toolkit CS6`

For details, see the *JavaScript Tools Guide*.

**How do I begin?**

It’s time to write your first script.

**NOTE:** If you have problems running your script, see Chapter 5, “Troubleshooting.”

**AS**

1. Open the Script Editor and type the following (substituting any Adobe application name in the quotes):

   ```javascript
   tell application "Adobe Photoshop CS6"
   make document
   end tell
   ```

2. Press Run.
JS

1. Open the ESTK and select an application from the drop-down list in the upper left corner of a document window.

2. In the JavaScript Console palette, type the following:
   
   ```javascript
   app.documents.add()
   ```

3. Do any of the following:
   
   ▶ Click the Run icon in the toolbar at the top of the Document window.
   
   ▶ Press F5.
   
   ▶ Choose **Debug -> Run**.

VBS

1. In a text editor, type the following (substituting any Adobe application in the quotes in the second line):

   ```vbscript
   Set appRef = CreateObject("Photoshop.Application")
   appRef.Documents.Add()
   ```

2. Save the file as a text file with a `.vbs` extension (for example, `create_doc.vbs`).

3. Double-click the file in Windows Explorer.
Scripting Basics

This chapter covers the basic concepts of scripting in both Windows and Mac OS. For product-specific directions, see the scripting guide for your Adobe application.

The building blocks of scripting

Your first script, which created a new document, was constructed like an English sentence, with a noun (document) and a verb (make in AS, add() in JS, and Add in VBS). In scripting, a noun is called an object, and a verb is called a command (in AS) or a method (in JS and VBS).

Just as you can modify a noun using adjectives, you can modify a script object using properties. To modify a command or method, you use parameters.

Understanding objects, properties, methods, and commands

When you use an Adobe application, you open a file or document, and then, within the document, you create or manipulate layers, text, frames, channels, graphic lines, colors, and other design elements. These things are objects.

To create a script statement, you create an object or refer to an existing object, and then you do one of the following:

- Define values for the object's properties. For example, you can specify a document's name, height, or width. You can specify a layer's name, color, or opacity.
- Specify commands or methods that tell the script to do what to your objects. For example, you can open, close, save, and print a document. You can merge, move, or rasterize a layer.

The thing to remember when writing a script is that you can use only the properties or methods/commands that are allowed for the object. How do you know which properties and methods go with which object? For the most part, it's logical. Generally, if you can specify something in your Adobe application, you can specify it in a script.

However, Adobe also spells it out for you in great detail in scripting resources that contain the information you need to create, define, and manipulate scripting objects. For information on locating and using these resources, see Chapter 3, “Finding an Object's Properties and Methods.”

Using Objects

The main concept to understand when using objects in scripts is how to refer to an object. How do you let the application know which object you want your script to change? In the application's user interface, you can simply select the object by clicking it. In a script, there's a little bit more to it.

DOM Concepts

Scripting languages use something called a Document Object Model (DOM) to organize objects in a way that makes the objects easy to identify. The principle behind a DOM is the containment hierarchy. In other
words, top level objects contain next level objects, which contain the subsequent level of objects, and so on.

For example, the top level object in any Adobe application DOM is the application object. Next is the document object, which contains all other objects, such as layers, channels, pages, text frames, and so on. These objects can contain objects that the document cannot contain directly. For example, in InDesign or Illustrator, a text frame can contain words. A document cannot contain words unless it has a text frame. Similarly, in Photoshop, a document can contain a layer, and a layer can contain a text frame, but a document cannot contain a text frame unless the document contains a layer.

**NOTE:** An object’s containing object is also called its parent object.

In your first script, you first named the application object (or selected it in the ESTK), and then you created the document within that application. If, as your next step, you wanted to create a layer, your script would need to identify the document in which you want to create the layer. If your script does not tell the application exactly where to create an object, your script fails.

**NOTE:** To view a chart of the DOM for a specific application, please refer to the application’s scripting guide.

So, using your DOM principle, how would you add a layer in a document? (To modify this script for Photoshop, please note that a layer is called art layer in AS; and layers are called artLayers in JS or ArtLayers in VBS).

### AS

tell application "Adobe Illustrator CS6"
    make document
    make layer in document
end tell

### JS

app.documents.layers.add()

### VBS

Set appRef = CreateObject("Illustrator.Application")
docRef.Documents.Add
appRef.Documents.Layers.Add

If you try to run these scripts, you get an error because the application does not know which document you mean. Sure, you have only one document open, but that won’t always be the case. Therefore, scripting languages have strict requirements that all objects be explicitly identified in every script statement.

This guide introduces three ways to refer to objects:

- Variables
- Collection or element numbers
- The “current” object or “active” object property

### Variables

A variable is a thing that you create to hold data in your script. The data, called the variable’s value, can be an object in your script, or it can be a property that describes an object. You could almost think of a variable as a nickname that you give to an object or other data.

Using a variable to contain an object makes an object easy to refer to. Most scripters create a variable for each object in their script.
The following scripts create a document, just as you did in your first script. However, this version of the script creates a variable named `myDoc` to contain the document. Take a look at these scripts, and then compare them to your first script. (See “How do I begin?” on page 7.)

**AS**

To create a variable in AS, you use the command `set`, followed by the variable name. To assign a data value to the variable, you use `=` followed by the value.

```
tell application "Adobe Illustrator CS6"
    set myDoc to make document
end tell
```

**JS**

To create a variable in JS, you use `var`, followed by the variable name. To assign a data value, you use an equal sign (`=`) followed by the value. Spaces do not matter on either side of the equal sign.

```
var myDoc = app.documents.add()
```

**VBS**

To create a variable in VBS, you use the command `Set`, followed by the variable name. To assign a data value, you use an equal sign (`=`) followed by the value. Spaces do not matter on either side of the equal sign.

```
Set appRef = CreateObject("Illustrator.Application")
Set docRef = appRef.Documents.Add
```

**Object references make life better**

Now that you have a way to refer to the `document` object created in the script, it’s easy to add the layer. (To modify this script for Photoshop, note that a layer is called `art layer` in AS; and layers are called `artLayers` in JS or `ArtLayers` in VBS).

**AS**

```
tell application "Adobe Illustrator CS6"
    set myDoc to make document
    make layer in myDoc
end tell
```

Even better, we could create another variable to hold the layer. That would allow us to easily refer to the layer if we wanted to define its properties or add an object to the layer.

```
tell application "Adobe Illustrator CS6"
    set myDoc to make document
    set myLayer to make layer in myDoc
end tell
```

**JS**

```
var myDoc = app.documents.add()
    myDoc.layers.add()
```

The same script again, this time creating a variable to hold the layer.

```
var myDoc = app.documents.add()
    var myLayer = myDoc.layers.add()
```

**VBS**

```
Set appRef = CreateObject("Illustrator.Application")
Set docRef = appRef.Documents.Add
    docRef.Layers.Add
```

The same script again, this time creating a variable to hold the layer.

```
Set appRef = CreateObject("Photoshop.Application")
```
Variables provide a nice shortcut

Variables that hold objects also hold the entire containment hierarchy that identifies the object. For example, to refer to myLayer, you do not need to refer to the document that contains the layer. The following scripts create a text frame in myLayer. Notice that, when you use myLayer, you don’t need to provide any containment hierarchy information about the layer.

**Note:** The following script uses the contents property to add text to the frame. For now, don’t worry about the mechanics of using properties.

The following script uses objects and properties defined in the Illustrator CS6 object model, so it does not work, for example, in InDesign or Photoshop.

**AS**
```
tell application "Adobe Illustrator CS6"
    set myDoc to make document
    set myLayer to make layer in myDoc
    set myTextFrame to make text frame in myLayer
    set contents of myTextFrame to "Hello world!"
end tell
```

**JS**
```
var myDoc = app.documents.add()
var myLayer = myDoc.layers.add()
var myTextFrame = myLayer.textFrames.add()
    myTextFrame.contents = "Hello world!"
```

**VBS**
```
Set appRef = CreateObject("Illustrator.Application")
Set docRef = appRef.Documents.Add
Set layerRef = docRef.Layers.Add
Set frameRef = layerRef.TextFrames.Add
    myTextFrame.Contents = "Hello world!"
```

Naming variables

Your scripts will be easier to read if you create descriptive names for your variables. Variable names such as x or c aren’t helpful when you revisit a script. Better names are those that indicate the data the variable contains, such as theDocument or myLayer.

Giving your variable names a standard prefix helps your variables stand out from the objects, commands, and keywords of your scripting system. For example:

- You could use the prefix “doc” at the beginning of any variables that contain Document objects, such as docRef, or “layer” to identify variables that contain Art Layer objects, such as layerRef and layerRef2.

- You could use the prefix “my” to add a personal element that separates your variables from script objects. For example, myDoc or myLayer or myTextFrame.

All variable names must conform to the following rules:

- Variable names must be a single word (no spaces). Many people use internal capitalization (such as myFirstPage) or underscore characters (my_first_page) to create more readable names. The variable name cannot begin with an underscore character.
Variable names can contain numbers but cannot begin with a number.

Variable names cannot contain quotation marks or punctuation other than the underscore character.

Variable names in JavaScript and VBScript are case sensitive. `thisString` is not the same as `thisstring` or `ThisString`. Variable names in AppleScript are not case sensitive.

Each variable in your script must have a unique name.

Object collections or elements as object references

Scripting languages put each object in a collection (JS or VBS) or an element (AS), and then assign the object a number, called the index, within the element or collection. The objects in an element or collection are identical types of objects. For example, each channel object in your document belongs to a channels element or collection; each art layer object belongs to an art layers element or an artLayers collection.

In English, you could refer to a document by saying, “Give me the first document in the collection.” Scripting languages allow you to identify an object in similar fashion, using its element or collection name and index.

In AS, you refer to the first document in the `documents` element as `document 1`.

In JS, the first document is `documents[0]`, (note the square braces surrounding the index) because (and this is hard to remember at first) JavaScript begins numbering collection objects at 0.

In VBS, the first document is `Documents(0)`, (note the parentheses around the index). VBS begins numbering collection objects at 1.

The following scripts reference the `document` and `layer` objects by index in order to add new objects.

**NOTE:** Because the following script does not use variables, the entire containment hierarchy is required in each object reference. For example, in the statement that adds a layer, the script must identify the document to which the layer will be added. To add a text frame to the layer, the script must provide the index not only of the layer that will contain the frame, but it must also identify the document that contains the layer.

**AS**

```anyscript
tell application "Adobe InDesign CS6"
    make document
    make layer in document 1
    make text frame in layer 1 of document 1
end tell
```

**NOTE:** Beginning scripters using AppleScript are not encouraged to use element numbers as object references when the element contains more than one object. For details as to why, see "How elements and collections number subsequent items" on page 14.

**JS**

In JavaScript, you indicate an item’s index by using the collection name followed by the index in square brackets (`[]`).

```javascript
app.documents.add()
app.documents[0].layers.add()
app.documents[0].layers[0].textFrames.add()
```

**NOTE:** Remember, in JS, index numbers within a collection start at 0.
In VBScript, you indicate an item’s index by using the collection name followed by the index in parentheses.

```vb
appRef.Documents.Add
appRef.Documents(1).Layers.Add
appRef.Documents(1).Layers(1).TextFrames.Add
```

### How elements and collections number subsequent items

Here’s how the scripting languages handle the automatic numbering if you add a second object to a collection or element:

- **AS** assigns number 1 to the new object and renumbers the previously existing object so that it is now number 2. AppleScript object numbers shift among objects to indicate the object that you worked with most recently. This can become confusing in longer scripts. Therefore, beginning scripters are encouraged to use variables as object references and avoid using indexes.

- **JS** collection numbers are static; they don’t shift when you add a new object to the collection. Object numbering in JS indicates the order in which the objects were added to the collection. Because the first object you added was assigned the number 0, the next object you add to the collection is number 1; if you add a third object, it is number 2. For example, when you add a document, the document automatically contains a layer. The layer’s index is [0]. If you add a layer, the new layer’s index is [1]; if you add a second layer, its index is [2]. If you drag layer [2] to the bottom position in the Layers palette, it still has index [2].

- **VBS** collection numbers are also static and the numbering performs exactly as described for JS collections, with the exception that the first object in the collection is always (1) in VBS.

**Tip:** In JS and VBS scripts, you’ll find index numbers very useful as object references. For example, you may have several files in which you want to make the background layer white. You can write a script that says “Open all files in this folder and change the first layer’s color to white.” If you didn’t have the capability of referring to the layers by index, you’d need to include in your script the names of all of the background layers in all of the files.

**Note:** Scripts are compulsive organizers. They place objects in elements or collections even when there is only one object of that type in the entire collection.

**Note:** Objects can belong to more than one collection or element. For example, in Photoshop, art layer objects belong to the art layers element or collection, and layer set objects belong to the layer sets element or collection, but both art layer objects and layer set objects belong to the layers element or collection. Similarly, in InDesign, rectangle objects belong to the rectangles element or collection and text frame objects belong to the text frames element or collection. However, both rectangle objects and text frame objects also belong to the page items element or collection, which contains all sorts of items on a page such as ellipses, graphic lines, polygons, buttons, and other items.

### Referring to the current or active object

When you ran your first script and created a new document, the application opened, and then it created a document. If you wanted to modify that document in the application’s user interface, you could have just gone to work with your mouse, menus, toolbox, and palettes, because the document was automatically selected.
This is true for all objects you create in a script. Until the script does something else, the new object is the active object, ready for modifications.

Conveniently, many parent objects contain properties that allow you to refer easily to the active object. (You’ll learn about properties in detail a little later in this guide. For now, you can just copy the script statements in this section and watch how they work without understanding completely why they look the way they do.)

► In AS, the property that refers to an active object consists of the word `current` and the object name. Some examples are:
  ```
  current document  
current layer  
current channel  
current view  
  ```

► In JS, the property name is a compound word that combines `active` with the object name, in standard JS case usage:
  ```
  activeDocument  
activeLayer  
activeChannel  
activeView  
  ```

► VBS is exactly the same as JS, except that all words in the combined term use initial caps. Some examples are:
  ```
  ActiveDocument  
ActiveLayer  
ActiveChannel  
ActiveView  
  ```

The following scripts create a document and then use this principle to create a layer in the new document.

**AS**
```
tell application "Adobe Illustrator CS6"
  make document
  make layer in current document
end tell
```

**JS**
```
app.documents.add()
app.activeDocument.layers.add()
```

**NOTE:** Be sure to type `activeDocument` without an `s` at the end.

**VBS**
```
Set appRef = CreateObject("Illustrator.Application")
docRef.Documents.Add
```

**NOTE:** Be sure to type `ActiveDocument` without an `s` at the end.
Using properties

To define or modify a property of an object, you do three things:

1. Name the object.
2. Name the property.
3. Specify the value for the property.

The value can be any of the following datatypes:

- A string, which is alphanumeric text that is interpreted as text. You enclose strings in quotes ("""). Strings include such values as an object’s name.
- Numeric, which is a number value that can be used in mathematical operations like addition or division. Mathematical numbers include the length of one side of a frame or the space between paragraphs, the opacity percentage, font size, stroke weight, and so on.

Note that some values that look like numbers are really strings. For example, a phone number or social security number are numbers, but you would format them as strings (enclose them in quotes) because the data would not be considered mathematical numbers.

Within the numeric category, there are different types of numbers:

- Integer, which is a whole number without any decimal points
- Real, fixed, short, long, or double, which are numbers that can include decimal digits, such as 5.9 or 1.0.

Note: These differences may not seem important now, but keep them in mind for later.

- A variable. When you use a variable as a property value, you do not enclose the variable in quotes as you would a string.
- A Boolean value, which is either true or false.

Note: In many cases, Boolean values act as an on/off switch.

- A constant value (also called an enumeration), which is a pre-defined set of values from which you can choose. Using constant values for a property is conceptually similar to using a drop-down menu in an Adobe application. Constants, and how and when to use them, are explained in “Constant values and enumerations” on page 20.

- A list (AS) or an array (JS and VBS).

Some properties require multiple values, such as the page coordinates of a point location (x and y coordinates), or the boundaries of a text frame or geometric object. Multiple values for a single property are called a list in AS and an array in JS or VBS. Each language specifies formatting rules.

- The list or array must be enclosed as follows:
  - In AS, the list is enclosed in curly braces: { }
  - In JS the array is enclosed in square brackets: []
  - In VBS, the array is enclosed in parentheses and follows the keyword Array: Array()
CHAPTER 2: Scripting Basics

Using properties 17

Values are separated by a comma (,). You can include or omit spaces after the commas; it doesn’t matter.

AS \{3,4,5\} or \{"string1", "string2", "string3"\}
JS [3,4,5] or ["string1", "string2", "string3"]
VBS Array(3,4,5) or Array("string1", "string2", "string3")

A list or array can include nested lists or arrays, such as a list of page coordinates. In the following samples, notice that each nested array is enclosed individually, and that the nested arrays are separated by commas.

AS \{{x1, y1}, {x2, y2}, {x3, y3}\}
JS [[x1, y1], [x2, y2], [x3, y3]]
VBS Array(Array(x1, y1), Array(x2, y2), Array(x3, y3))

AS

To use properties in AS, you use the set command followed by the property name, and then type of followed by the object reference. The following script defines the name property of the layer object.

tell application "Adobe Illustrator CS6"
    set myDoc to make document
    set myLayer to make layer in myDoc
    set name of myLayer to "My New Layer"
end tell

You can set several properties in a single statement using the properties property. You format the multiple properties as an array, enclosed in curly braces. Within the array, separate each property name/property value pair with a colon (:) The following script uses properties to define the layer’s name and visibility state.

tell application "Adobe Illustrator CS6"
    set myDoc to make document
    set myLayer to make layer in myDoc
    set properties of myLayer to \{name:"My New Layer", visible:false\}
end tell

NOTE: Notice in the preceding script that only the string value "My New Layer" is enclosed in quotes. The value for the visible property, false, may look like a string, but it is a Boolean value. To review value types, see "Using properties" on page 16.

You can define an object’s properties in the statement that creates the object, as in the following scripts.

tell application "Adobe Illustrator CS6"
    set myDoc to make document
    set myLayer to make layer in myDoc with properties \{name:"My New Layer"\}
end tell

tell application "Adobe Illustrator CS6"
    set myDoc to make document
    set myLayer to make layer in myDoc with properties \{name:"My New Layer", visible:false\}
end tell


**JS**

To use a property in JS, you name the object that you want the property to define or modify, insert a period (.), and then name the property. To specify the value, place an equal sign (=) after the property name, and then type the value.

```javascript
var myDoc = app.documents.add()
var myLayer = myDoc.layers.add()
    myLayer.name = "My New Layer"
```

To define multiple properties, you can write multiple statements:

```javascript
var myDoc = app.documents.add()
var myLayer = myDoc.layers.add()
    myLayer.name = "My New Layer"
    myLayer.visible = false
```

**NOTE:** Notice in the preceding script that only the string value "My New Layer" is enclosed in quotes. The value for the visible property, false, may look like a string, but it is a Boolean value. To review value types, see "Using properties" on page 16.

JS provides a shorthand for defining multiple properties, called a with statement. To use a with statement, you use the word with followed by the object whose properties you want to define, enclosing the object reference in parentheses (()). Do not type a space between with and the first parenthesis. Next, you type an opening curly brace ({), and then press Enter and type a property name and value on the following line. To close the with statement, you type a closing curly brace (}).

```javascript
var myDoc = app.documents.add()
var myLayer = myDoc.layers.add()
    with(myLayer){
        name = "My New Layer"
        visible = false
    }
```

Using a with statement saves you the trouble of typing the object reference followed by a period (in this case, myLayer.) for each property. When using a with statement, always remember the closing curly bracket.

JS also provides a properties property, which allows you to define several values in one statement. You enclose the entire group of values in curly braces ({>). Within the braces, you use a colon (:) to separate a property name from its value, and separate property name/property value pairs using a comma (,).

```javascript
var myDoc = app.documents.add()
var myLayer = myDoc.layers.add()
    myLayer.properties = {name:"My New Layer", visible:false}
```

**VBS**

To use properties in VBS, you name the object, insert a period (.), and then name the property. To specify the value, place an equal sign (=) after the property name, and then type the value.

```vbs
Set appRef = CreateObject("Illustrator.Application")
Set myDoc = appRef.Documents.Add
Set myLayer = myDoc.Layers.Add
    myLayer.Name = "My First Layer"
```
You can define only one property per statement. To define multiple properties, you must write multiple statements:

Set appRef = CreateObject("Illustrator.Application")
Set myDoc = appRef.Documents.Add
Set myLayer = myDoc.Layers.Add
myLayer.Name = "My First Layer"
myLayer.Opacity = 65
myLayer.Visible = false

**NOTE:** Notice in the preceding script that only the string value "My New Layer" is enclosed in quotes. The value for the Visible property, false, may look like a string, but it is a Boolean value. To review value types, see "Using properties" on page 16.

### Understanding read-only and read-write properties

When defining property values, you can write a script statement with perfect syntax, but the statement does not produce any results. This can happen when you try to define a property that is not "writable"; the property is read-only.

For example, the name property of the document object in most Adobe applications is read-only; therefore, you cannot use a script to define or change the name of an existing document (although you can use a save as command or method; see "Using methods or commands" on page 23 for information). So why bother to have a property that you can’t set, you might ask. The answer is that read-only properties are valuable sources of information. For example, you may want to find out what a document’s name is, or how many documents are in the Documents collection.

### Using alert boxes to show a property’s value

A good way to display information in a read-only property is to use the alert box, which is a small dialog that simply displays information. You can use alert boxes to display the value of any property: read-write or read-only.

**AS**

To display an alert box in AS, you type display dialog, and then type the dialog content in parentheses (()). To find out how many objects are in an element, use the count command with any element name.

**NOTE:** The element name is the plural form of the object. For example, the document object’s element is the documents object.

The following script displays an alert box that tells you how many documents are in the documents element, then adds a document and displays a new alert with the updated number.

```
tell application "Adobe Photoshop CS6"
  display dialog (count documents)
  set myDoc to make document
  display dialog (count documents)
end tell
```

To get a string value to display in an alert box, you must store the string value in a variable. The following script converts the document name to a variable named myName, and then displays the value of myName.

```
tell application "Adobe Photoshop CS6"
  set myDoc to make document
  set myName to name of myDoc
```
display dialog myName
end tell

JS

To display an alert box in JS, you use the alert() method by typing alert, and then typing the dialog content in parentheses (()). Do not type a space between alert and the first parenthesis. To find out how many objects are in a collection, use the (read-only) length property of any collection object. The following script displays an alert box that tells you how many documents are in the documents collection, then adds a document and displays a new alert with the updated number.

NOTE: The collection object name is the plural form of the object. For example, the document object’s collection object is the documents object.

alert(app.documents.length)
var myDoc = app.documents.add()
alert(app.documents.length)

The following script displays the document’s name in an alert box.

var myDoc = app.documents.add()
alert(myDoc.name)

VBS

To display an alert box in VBS, you use the MsgBox method by typing MsgBox, and then typing the dialog content in parentheses (()). Do not type a space between MsgBox and the first parenthesis. To find out how many objects are in a collection, use the (read-only) Count property of any collection object. The following script displays an alert box that tells you how many documents are in the Documents collection, then adds a document and displays a new alert with the updated number.

NOTE: The collection object is the plural form of the object. For example, the Document object’s collection object is the Documents object.

Set appRef = CreateObject("Photoshop.Application")
MsgBox(appRef.Documents.Count)
Set myDoc = appRef.Documents.Add
MsgBox(appRef.Documents.Count)

The following script displays the document’s name in an alert box.

Set appRef = CreateObject("Photoshop.Application")
Set myDoc = appRef.Documents.Add
MsgBox(myDoc.Name)

Constant values and enumerations

Some properties’ values are pre-defined by the application. For example, in most applications, the page orientation can be either landscape or portrait. The application accepts only one of these two values; it will not accept “vertical” or “upright” or “horizontal” or “on its side.” To make sure your script provides an acceptable value for a document’s page orientation property, the property has been written so that it can accept only a pre-defined value.

In scripting, these pre-defined values are called constants or enumerations.

Using a constant or an enumeration is similar to using a drop-down list in the application’s user interface.
NOTE: To find whether you must use an enumeration for a property’s value, look up the property in one of the scripting references provided by Adobe. For information, see Chapter 3, “Finding an Object’s Properties and Methods.”

AS

In AS, you use constants as you would any other property definition. Do not enclose the constant in quotes. The following script uses the constant value dark green to set the layer color of a new layer.

```
tell application "Adobe Illustrator CS6"
    set myDoc to make document
    set myLayer to make layer in myDoc
    set layer color of myLayer to dark green
end tell
```

NOTE: If dark green were a string value rather than a constant, the value would be enclosed in quotes:

```
set layer color of myLayer to "dark green"
```

JS

In JS, you type the enumeration name, a period (.), and then the enumeration value. You must use the exact spelling and capitalization as defined in the scripting references provided by Adobe. Formatting is different in different Adobe applications. For example:

► In InDesign:
  
  ▶ Each enumeration begins with an upper case letter, and all words within the combined term also begin with an upper case letter.

  ▶ The enumeration value begins with a lower case letter.

  The following example uses the UIColor enumeration to set the layer color to dark green.

  ```
  var myDoc = app.documents.add()
  var myLayer = mydoc.layers.add()
  myLayer.layerColor = UIColor.darkGreen
  ```

► In Illustrator:

  ▶ Each enumeration begins with an upper case letter, and all words within the combined term also begin with an upper case letter.

  ▶ Some enumeration values begin with an upper case letter and then use lower case letters. Others use all upper case. You must be sure to use the value exactly as it appears in the scripting reference.

  The following example uses the RulerUnits enumeration to set the default unit to centimeters.

  ```
  var myDoc = app.documents.add()
  myDoc.rulerUnits = RulerUnits.Centimeters
  ```
CHAPTER 2: Scripting Basics

The next script uses the BlendModes enumeration, whose values are expressed in all upper case letters.

```javascript
var myDoc = app.documents.add()
var myLayer = myDoc.layers.add()
    myLayer.blendingMode = BlendModes.COLORBURN
```

▶ In Photoshop:

- Each enumeration begins with an upper case letter, and all words within the combined term also begin with an upper case letter.
- Enumeration values are all upper case.

The following example uses the LayerKind enumeration to make the layer a text layer.

```javascript
var myDoc = app.documents.add()
var myLayer = myDoc.artLayers.add()
    myLayer.kind = LayerKind.TEXT
```

VBS

In VBS, you use numeric values for constants.

```javascript
Set appRef = CreateObject("Photoshop.Application")
Set docRef = appRef.Documents.Add
Set layerRef = docRef.ArtLayers.Add
    layerRef.Kind = 2
```

Using variables for property values

You can use variables to contain property values. This can help you update a script quickly and accurately. For example, you may have a publication in which all photos are 3 x 5 inches. If you use a variable to set the photo height and the photo width, and then the measurements change, you only have to change the values in one variable, rather than the measurements for each photo in the document.

The following script creates variables to contain the values of the document's width and height, and then uses the variables as values in the statement that changes the width and height.

AS

```javascript
tell application "Adobe Illustrator CS6"
    set myDoc to make document with properties {height:10, width:7}
    set docHeight to height of myDoc
    set docWidth to width of myDoc
    set myDoc with properties {height:docHeight - 2, width:docWidth - 2}
end tell
```

JS

```javascript
var myDoc = app.documents.add(7, 10)
    var docHeight = myDoc.height
    var docWidth = myDoc.width
    myDoc.resizeCanvas((docHeight - 2), (docWidth - 2))
```

VBS

```javascript
Set appRef = CreateObject("Photoshop.Application")
Set myDoc = appRef.Documents.Add(7, 10)
    docHeight = myDoc.Height
```
Using methods or commands

Commands (in AS) and methods (in VBS and JS) are directions you add to a script to perform tasks or obtain results. For example, you could use the `print/print() /PrintOut` command/method to print a document.

AS

AS commands appear at the beginning of a script statement as an imperative verb. The command is followed by a reference to the object upon which you want the command to act.

The following script prints the active document:

```asciitext
tell application "Adobe InDesign CS6"
  print current document
end tell
```

JS

You insert methods at the end of JS statements. You must place a period before the method name, and then follow the method name with parentheses (()).

```asciitext
app.activeDocument.print()
```

VBS

You insert methods at the end of VBS statements. You must place a period before the method name.

```asciitext
Set appRef = CreateObject("Photoshop.Application")
appRef.ActiveDocument.PrintOut
```

Command or method parameters

Some commands or methods require additional data, called `arguments or parameters`. Commands or methods can also have optional parameters.

**Required parameters**

The following scripts use the `merge` command, which requires some indication of the layers you want to merge into the selected layer. Just like properties, command parameters are enclosed in curly braces ({}).

However, you include only the parameter value, and not the parameter name, within the braces.

**Note:** This script is for InDesign. There is no merge operation in Illustrator. To modify this script for Photoshop, note that a layer is called `art layer` in AS; and layers are called `artLayers` in JS or `ArtLayers` in VBS.

AS

```asciitext
tell application "Adobe InDesign CS6"
  set myDoc to make document
  set myLayer to make layer in myDoc
  set myLayer2 to make layer in myDoc
```
merge myLayer2 with {myLayer}
end tell

**JS**

The method parameter is enclosed in the parentheses that follow the method name.

```js
var myDoc = app.documents.add()
var myLayer = myDoc.layers.add()
var myLayer2 = myDoc.layers.add()

myLayer2.merge(myLayer)
```

**VBS**

Notice that the method parameter is enclosed in parentheses after the method name. Do not type a space before the first parenthesis.

```vbs
Set appRef = CreateObject(“InDesign.Application”)
Set myDoc = appRef.Documents.Add
Set myLayer = myDoc.Layers.Add
Set myLayer2 = myDoc.Layers.Add

myLayer2.Merge(myLayer)
```

**Multiple parameters**

When you define more than one parameter for a command or method, you must follow specific rules.

**AS**

There are two types of parameters for AS commands:

- A *direct* parameter, which defines the direct object of the action performed by the command
- *Labeled* parameters, which are any parameters other than direct parameters

The direct parameter must follow the command directly. In the following statement, the command is `make` and the direct parameter is `document`.

```
make document
```

You can insert labeled parameters in any order. The following script creates two layers, and defines the location and name of each layer. Notice that, in the statements that create the layers, the location and name parameters appear in different orders.

```
tell application "Adobe InDesign CS6"
    set myDoc to make document
    tell myDoc
        set myLayer to make layer at beginning of myDoc with properties {name:"Lay1"}
        set myLayer2 to make layer with properties {name:"Lay2"} at end of myDoc
    end tell
end tell
```
CHAPTER 2: Scripting Basics

Tell statements (AS only)

In JS, you must enter parameter values in the order they are listed in the scripting reference resources so that the script compiler knows which value defines which parameter.

**NOTE:** For information on scripting reference resources, see Chapter 3, “Finding an Object’s Properties and Methods.”

To skip an optional parameter, type the placeholder `undefined`. The following statement creates a Photoshop CS6 document whose width is 4000 pixels, height is 5000 pixels, resolution is 72, name is “My Document,” and document mode is bitmap.

```javascript
app.documents.add(4000, 5000, 72, "My Document", NewDocumentMode.BITMAP)
```

The next statement creates an identical document except that the resolution is left undefined.

```javascript
app.documents.add(4000, 5000, undefined, "My Document", NewDocumentMode.BITMAP)
```

**NOTE:** Use the `undefined` placeholder only to “reach” the parameters you want to define. The following statement defines only the document’s height and width; placeholders are not needed for subsequent optional parameters.

```javascript
app.documents.add(4000, 5000)
```

In VBS, you must enter parameter values in the order they are listed so that the script compiler knows which value defines which parameter.

To skip an optional parameter, type the placeholder `undefined`. The following statement creates a Photoshop CS6 document whose width is 4000 pixels, height is 5000 pixels, resolution is 72, name is “My Document,” and document mode is bitmap.

```vbs
Set appRef = CreateObject("Photoshop.Application")
Set myDoc = appRef.Documents.Add(4000, 5000, 72, "My Document", 5)
```

The next statement creates an identical document except the resolution is left undefined.

```vbs
Set appRef = CreateObject("Photoshop.Application")
```

**NOTE:** Use the `undefined` placeholder only to “reach” the parameters you want to define. The following statement defines only the document’s height and width; placeholders are not needed for subsequent optional parameters.

```vbs
Set appRef = CreateObject("Photoshop.Application")
Set myDoc = appRef.Documents.Add(4000, 5000)
```

The `undefined` placeholder is not case-sensitive.

Tell statements (AS only)

You may have noticed that AppleScript examples start and end with the statements:
A `tell` statement names the default object that performs all commands contained within the statement. In the preceding sample, the `tell` statement targets the application object. Therefore, any commands contained within the statement must be performed by the application object unless another object is explicitly named in a script statement within the `tell` statement.

The following script carefully outlines the full containment hierarchy of each object to indicate which object the command must work upon:

```plaintext
tell application "Adobe InDesign CS6"
    set myDoc to make document
    set myLayer to make layer in myDoc
    set myLayer2 to make layer in myDoc
end tell
```

You can create a shortcut by changing the command target. To do so, you add a nested `tell` statement. The following script performs the exact same operation as the previous script. Because the nested `tell` statement targets the document object, it is not necessary to refer to the document object in the statements that create the layers.

```plaintext
tell application "Adobe InDesign CS6"
    set myDoc to make document
    tell myDoc
        set myLayer to make layer
        set myLayer2 to make layer
    end tell
end tell
```

Notice that each `tell` statement must be closed with its own `end tell` statement.

You can nest as many `tell` statements as you wish.

## Notes about variables

This section provides additional information about using variables.

### Changing a variable’s value

You can change a variable’s value at any time. To do so, you simply use the variable name followed by the assignment operator (to in AS; = in JS or VBS) and the new value. The following scripts create the variable `layerRef` to contain a new layer, and then immediately create a second layer and assign it as `layerRef`’s new value.

**AS**

To change a variable’s value in AS, you use the `set` command.

```plaintext
tell application "Adobe Illustrator CS6"
    set docRef to make document
    set layerRef to make layer in myDoc with properties {name: "First Layer"}
    set layerRef to make layer in myDoc with properties {name: "Second Layer"}
end tell
```

**JS**

To change a variable’s value in JS, you use the variable name followed an equal sign (=) and the new value. Do not begin the reassignment statement with `var`; you use `var` only when creating a new variable.
CHAPTER 2: Scripting Basics

Making script files readable

This section covers two options that help make your script files more readable:

- Comments
- Line breaks

Commenting the script

A script comment is text that the scripting engine ignores when it executes your script.

Comments are very useful when you want to document the operation or purpose of a script (for yourself or for someone else). Most programmers, even the most advanced, take the time to insert comments for almost every element in a script. Comments may not seem important to you when you are writing your scripts, but you will be glad you included comments a month or a year later when you open a script and wonder what you were trying to do and why.

To comment all or part of a single line in an AS, type two hyphens (--) at the beginning of the comment. To comment multiple lines, surround the comment with (* and *).

tell application "Adobe InDesign CS6"
  --This is a single-line comment
  print current document --this is a partial-line comment
  --the hyphens hide everything to their right from the scripting engine
(* This is a multi-line
  comment, which is completely

VBS

To change a variable's value in VBS, you use the Set command.

Set appRef = CreateObject("Illustrator.Application")
Set docRef = appRef.Documents.Add
Set layerRef = docRef.Layers.Add
  layerRef.Name = "First Layer"
Set layerRef = docRef.Layers.Add
  layerRef.Name = "Second Layer"

Using variables to refer to existing objects

You can also create variables to contain existing objects.

AS
tell application "Adobe Photoshop CS6"
  set myDoc to active document
end tell

JS

var myDoc = app.activeDocument

VBS

Set appRef = CreateObject("Illustrator.Application")
Set docRef = appRef.ActiveDocument
CHAPTER 2: Scripting Basics

Making script files readable

ignored by the scripting engine, no matter how many lines it contains.
The trick is to remember to close the comment.
If you donít the rest of your script is hidden from the scripting engine!*)
end tell

NOTE: The only thing this script does is print the current document.

JS

To comment all or part of a single line in JS, type two forward slashes (//) at the beginning of the comment. To comment multiple lines, surround the comment with /* and */.

//This is a single-line comment
app.activeDocument.print() //this part of the line is also a comment

/* This is a multi-line
   comment, which is completely ignored by the scripting engine, no matter how many lines it contains.
   Donít forget the closing asterisk and slash or the rest of your script will be commented out...*/

NOTE: The only thing this script does is print the active document.

VBS

In VBS, type Rem (for “remark”) or ‘ (a single straight quote) at the beginning of the comment. VBS does not support comments that span more than one line. To comment several lines in a row, start each line with either comment format.

'This is a comment.
Set appRef = CreateObject("Photoshop.Application")
Rem This is also a comment.
appRef.ActiveDocument.PrintOut 'This part of the line is a comment.
' This is a multi-line ' comment that requires ' a comment marker at the beginning ' of each line.
Rem This is also a multi-line comment. Generally, multi-line Rem comments in VBS are easier for you to identify (and read) in your scripts Rem if they begin with a single straight quote ('') rather than if they begin Rem with Rem, because Rem can look like any other text in the script '
The choice is yours but isnít this more easily identifiable as a comment than the preceding ' four lines were?

NOTE: The only thing this script does is print the active document.

Continuing long lines in AppleScript and VBScript

In both AppleScript and VBScript, a carriage return at the end of a line signals the end of a statement. When your script lines are too long to fit on one line, you can use special continuation characters—characters that break a line but direct the script to read the broken line as a legitimate instruction.

NOTE: You can also expand the scripting editor window to continue the statement on a single line.

AS

Type the character ¬ (Option+Return) to break a long line but continue the statement.

tell application "Adobe InDesign CS6"
   set myDoc to make document
set myLayer to make layer in myDoc with properties {name:"My First Layer"} at the beginning of myDoc (* without the line break character, AS would consider this line an incomplete statement*)
(* note that line continuation characters are not required in a multi-line comment such as this one*)
set myLayer2 to make layer in myDoc with properties {name:"My Other Layer"} before myLayer
end tell

**VBS**

Type an underscore (_) followed by a carriage return to break a long line but continue the statement.

**NOTE:** In both languages, the continuation character loses its functionality if it is placed inside a string (that is, within the quotes). If the line break occurs within a string value, place the break character before the string and insert the line break early.

**NOTE:** In JavaScript, statements can contain carriage returns, so there is no need for a continuation character. However, the ExtendScript interpreter interprets each line as a complete statement. In general, therefore, it’s best to insert returns only at the ends of statements.

### Using Arrays

In VBScript and JavaScript, arrays are similar to collections; however, unlike collections, arrays are not created automatically.

You can think of an array as a list of values for a single variable. For example, the following JavaScript array lists four values for the variable myFiles:

```javascript
var myFiles = new Array ()
    myFiles[0] = "clouds.bmp"
    myFiles[1] = "clouds.gif"
    myFiles[2] = "clouds.jpg"
    myFiles[3] = "clouds.pdf"
```

Notice that each value is numbered. To use a value in a statement, you must include the number. The following statement opens the file clouds.gif:

```javascript
open(myFiles[1])
```

The following sample includes the same statements in VBScript:

```vbnet
Dim myFiles (4)
    myFiles(0) = "clouds.bmp"
    myFiles(1) = "clouds.gif"
    myFiles(2) = "clouds.jpg"
    myFiles(3) = "clouds.pdf"
appRef.Open myFiles(1)
```

**NOTE:** While indexes in VBS collections always begin numbering at (1), you can stipulate in your VBS scripts whether arrays that you create begin numbering at (1) or (0). To find out how to set the array index starting number, refer to any VBScript text book. For information on collections and index numbers, see “Object collections or elements as object references” on page 13.
Creating objects

Your first script demonstrated how to create an object using the `make` command (AS), `add()` method (JS), or `Add` method (VBS) of the object’s collection object. For example:

**AS**
```
tell application "Adobe Photoshop CS6"
  make document
end tell
```

**JS**
```
app.documents.add()
```

**VBS**
```
Set appRef = CreateObject("Photoshop.Application")
appRef.Documents.Add()
```

However, some objects do not have a `make` command (AS), `add()` method (JS), or `Add` method (VBS). To create objects of these types, refer to the section “Creating new objects” in the chapter for your scripting language in the Adobe scripting guide for your application.

More information about scripting

At this point you have enough knowledge to create simple scripts that perform basic tasks. To further your scripting skills, use any of the following resources:

- “Advanced Scripting Techniques” on page 43
- The Adobe scripting guide for your application
- Chapter 6, “Bibliography”
Adobe provides the following resources to help you find and use the objects, methods or commands, properties, enumerations, and parameters you need to create effective scripts.

- Object dictionaries or type libraries. Each scriptable Adobe application provides a reference library or dictionary within your script editor environment.
- The Adobe scripting reference documents (in PDF format), which are located on your installation CD. (Scripting reference documents are not provided for all Adobe applications.)

Using scripting environment browsers

This section explains how to display and use the scripting environment object browsers for each scripting language.

AppleScript data dictionaries

The AppleScript dictionaries are available through Apple's Script Editor application.

Displaying the AppleScript dictionaries

NOTE: The default location for the Script Editor application is Applications > AppleScript > Script Editor.

1. In Script Editor, choose File > Open Dictionary. Script Editor displays an Open Dictionary dialog.
2. Choose your Adobe application, and then choose Open. Script Editor opens the Adobe application and then displays the application's dictionary.

Using the AppleScript dictionaries

The AS dictionary divides objects into suites. Suite names are indicative of the type of objects that the suite contains.

To view an object's properties:

1. In the upper left pane of the data dictionary screen, select the suite that contains the object.
2. Select the object in the upper middle pane.

NOTE: Objects are indicated by a square icon: ☐; commands are indicated by a round icon: ☐.

The object description appears in the lower viewing pane. The object's elements and properties are listed below the description. Each element name is a hyperlink to the element's object type.

3. Each property listing contains the following:
   - The property name
   - The data type in parentheses.
CHAPTER 3: Finding an Object’s Properties and Methods

Using scripting environment browsers

If the data type is an object, the data type is a hyperlink to the object.

If the data type is an enumeration, the data type is “anything.” The valid values are listed after the property description and separated by forward slashes (/), and are preceded by the notation “Can return:”.

- The access value:
  - If the object is read-only, r/o appears after the data type.
  - If the object is read-write, no access value is given.

- A description of the property.

Viewing commands and command parameters

**NOTE:** The data dictionary lists the objects you can use with a command. However, it does not list the commands you can use with an object. To view a list of commands you can use with an object, refer to the AppleScript scripting reference for your application. See “Using Adobe scripting reference documents” on page 38 for more information.

To view commands in the data dictionary, follow these steps:

1. In the upper left pane of the data dictionary screen, select the suite that contains the command.
   - The upper middle pane lists the commands and objects contained in the suite.

2. Select the command in the upper middle pane.
NOTE: Commands are indicated by a round icon: ↓; objects are indicated by a square icon: ☐.

The command description appears in the lower viewing pane.

Below the description, the objects with which you can use the command are listed.

Below the list of supported objects, the parameters are listed.

If the parameter is optional, it is enclosed in square brackets ([ ]).

If no brackets appear around the parameter name, the parameter is required.

Each parameter name is followed by the data type.

If the data type is an object, the data type is a hyperlink to the object.

If the data type is an enumeration, the valid values are preceded by the notation “Can accept:” and then listed, separated by forward slashes (/).

1. Select a suite to display the suite’s commands and objects in the upper middle pane.

2. Select the command.

3. View the command’s information in the lower pane:

   Command description

   List of objects that use the command

   Parameters, with data types and descriptions

   Optional parameters are enclosed in square brackets ([ ])

   Note: When the parameter value is an enumeration, the enumerated values are preceded by “Can accept:”

JavaScript object-model viewer

You can use the ExtendScript Tools Kit (ESTK), which is installed with your Adobe applications, to display the JavaScript objects and methods available for your Adobe application.

For information on displaying and using the JavaScript object model viewer for your Adobe application, see JavaScript Tools Guide.

VBScript type libraries

You can use the Visual Basic editor in any Microsoft Office application to display the VBScript objects and methods available for your Adobe application.
CHAPTER 3: Finding an Object’s Properties and Methods

NOTE: If you use a different editor, refer to the editor’s help system to find out how to display type libraries.

Displaying the VBScript type libraries

To view the VBS object library, follow these steps:

1. Start any Microsoft Office application, and then choose Tools > Macro > Visual Basic Editor.
2. In the Visual Basic editor window, choose Tools > References.
3. In the References dialog’s Available References list, select your Creative Suite application, and then click OK.
4. In the Visual Basic editor window, choose View > Object Browser.
5. Select your Adobe application in the drop-down list in the upper left corner of the Object Browser window.

Using the VBScript type libraries

The VBS object type library displays objects and constants in the Classes pane on the left side of the Object Browser window. In the Classes pane:

- Objects are indicated by the following icon: 📚
- Constants are indicated by the following icon: 📚

To display an object’s properties and method, you select the object type in the Classes pane. The properties and methods are listed in the Members pane to the right of the Classes pane.

- Properties are indicated by the following icon: 📚
Methods are indicated by the following icon: 

Understanding property listings in the Object Browser

When you select a property in the Members of pane, the property’s information is displayed in the information pane at the bottom of the Object Browser window as follows:

- The property name is followed by the data type.
  - If the data type is a constant, the constant appears as a hyperlink to the constant’s values. Constant names begin with a prefix that matches the Adobe application’s abbreviated name. For example:
    - The prefix `Ps` is used for enumerations in Photoshop CS6. Examples: `PsColorProfileType`, `PsBitsPerChannelType`
    - The prefix `id` is used for enumerations in InDesign CS6. Examples: `idRenderingIntent`, `idPageOrientation`
    - The prefix `Ai` is used for enumerations in Adobe Illustrator CS6. Examples: `AiCropOptions`, `AiBlendModes`
  - If the data type is an object, the object name is a hyperlink to the object type.
- The access value appears only if the property is read-only. If the property is read-write, no access value appears.

![Image of Object Browser window]

1. Select the property in the “Members of” pane.

- The data type appears next to the property name.
- The access value is listed only when access is read-only.
- The property description appears at the bottom of the information pane.
Finding a property's numeric value

In VBS, you use an enumeration's numeric value as a property value. For example, in the following script, the layer type, represented by the `Kind` property in the last line of the script, is defined by the numeric value 2, which represents the `TextLayer` constant value.

```vbs
Set appRef = CreateObject("Photoshop.Application")
Set docRef = appRef.Documents.Add
Set layerRef = docRef.ArtLayers.Add
layerRef.Kind = 2 'PsTextLayer
```

To find an enumeration's numeric value, follow these steps:

1. Click the link to the enumeration's information.
2. Click the enumeration value to display the numeric value in the bottom pane.

Understanding method listings

When you select a method in the “Members of” pane, the method’s information is displayed in the information pane at the bottom of the Object Browser window as follows:

- The method name is followed by the parameters.
  - Optional parameters are enclosed in square brackets ([ ]).
  - If no brackets appear around a parameter name, the parameter is required.
- Each parameter name is followed by the data type.
  - If the data type is an object, the data type is a hyperlink to the object.
  - If the data type is an enumeration, the enumeration name begins with the application’s initials and is a hyperlink to the enumeration’s information.
  - If a default value exists for a parameter, the value is listed after the datatype after an equal sign (=).

**NOTE:** The default value is used if you do not define a value for the parameter. Only optional parameters have default values.
Using Adobe scripting reference documents

Adobe provides scripting references for many applications. The references are located on your installation CD.

In the scripting references, each language is documented in a separate chapter. Within each chapter, the objects are listed alphabetically. For each object, the following tables are provided:

- Elements (AS only)
- Properties
- Methods, commands, or functions

Additionally, most object sections contain a scripting sample using the object and some of its properties and methods or commands. You can use any sample script as an example or a starting point for your script, in which you may change properties or methods.

Working with an object’s elements table (AS only)

Elements are the object collections contained by an object. When object contains elements, a table shows the various ways in which you can refer to the elements. For beginning scripters, the main thing to understand about the Elements table is the Name or Element column, which tells you which objects are...
just below the object in the containment hierarchy. For example, the following Elements table is taken from a document object in InDesign.

<table>
<thead>
<tr>
<th>Name</th>
<th>Refer to by</th>
</tr>
</thead>
<tbody>
<tr>
<td>character style</td>
<td>index, name, range, relative, satisfying a test, ID</td>
</tr>
<tr>
<td>layer</td>
<td>index, name, range, relative, satisfying a test, ID</td>
</tr>
<tr>
<td>story</td>
<td>index, name, range, relative, satisfying a test, ID</td>
</tr>
</tbody>
</table>

The information you can get from this table is that, in document objects that you create for this application, you can create character style, layer, and story objects.

For example:

```objectivec
tell application "Adobe InDesign CS6"
    set myDoc to make document
    set myCharStyle to make character style in myDoc with properties {name:"Bold"}
    set myLayer to make layer in myDoc
    set myStory to make story in myDoc
end tell
```

The following script statement would produce an error, because stroke style is not an element of this application’s document object.

```objectivec
tell application "Adobe InDesign CS6"
    set myDoc to make document
    set myStrokeStyle to make stroke style in myDoc with properties {name:"Erratic"}
end tell
```

**Working with an object’s properties table**

The properties table for an object lists the following:

- The properties you can use with the object
- The value type for each property
  - When the value type is a constant or enumeration, the value is presented either as a list of valid values or as a hypertext link to the constant’s listing.
  - When the value type is another object, the value is presented as a hypertext link to the object’s listing.
- The property’s input status: *Read-only* or *Read-write*
- A description, which includes the following:
  - An explanation of what the property defines or does
  - Ranges for valid values
  - Dependencies on other properties

The following sample Properties table for an art layer object in Photoshop contains samples of each type of data.
CHAPTER 3: Finding an Object’s Properties and Methods

Using Adobe scripting reference documents

For example:

**AS**

tell application "Adobe Photoshop CS6"
    set myDoc to make document
    set myLayer to make art layer in myDoc
    set properties of myLayer to {kind:text layer, name:"Captions", opacity:45.5, "visible":true}
    set contents of text object in myLayer to "Photo Captions"
end tell

**NOTE:** You cannot define the bounds of the layer because the `bounds` property is read-only.

**JS**

var myDoc = app.documents.add()
var myLayer = myDoc.artLayers.add()
    alert(myLayer.bounds) // can't set the property because it is read-only
    myLayer.kind = LayerKind.TEXT
    myLayer.name = "Captions"
    myLayer.opacity = 45.5 // can use a decimal point because the type is not integer
    myLayer.textItem.contents = "Day 1: John goes to school"
    //see the properties table for the textItem object to find the contents property
    myLayer.visible = true

**NOTE:** In JS and VBS, collection objects are kept in properties of the containing object. To determine an object’s containment hierarchy, you must locate the object or objects that use the object’s collection object (that is, the object’s plural form) as a property. For example, `documents.layers` or `layers.textFrames`.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value type</th>
<th>What it is</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounds</td>
<td>Array of 4 numbers</td>
<td>Read-only. An array of coordinates that describes the bounding rectangle of the layer in the format [y1, x1, y2, x2].</td>
</tr>
<tr>
<td>kind</td>
<td>LayerKind</td>
<td>Read-only. The type of layer.</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Read-write. The name of the layer.</td>
</tr>
<tr>
<td>opacity</td>
<td>number (double)</td>
<td>Read-write. The opacity as a percentage. (Range: 0.0 to 100.0)</td>
</tr>
<tr>
<td>textItem</td>
<td>TextItem object</td>
<td>Read-only. The text item that is associated with the layer.</td>
</tr>
<tr>
<td>visible</td>
<td>Boolean</td>
<td>Read-write. If true, the layer is visible.</td>
</tr>
</tbody>
</table>
Working with an object’s methods table

The Methods table for an object lists the following:

- The methods you can use with the object
- The parameter(s) for each method
  - When a parameter type is a constant or another object, the value is presented as a hypertext link to the constant or object’s listing. In the Methods table sample below, the parameter types NewDocumentMode and DocumentFill are constants.
  - Parameters can be required or optional. Optional parameters are indicated by square brackets ([]).
- Return value type(s), which is what the method produces
  - When a return is a constant or another object, the value is presented as a hypertext link to the constant or object’s listing. In the Methods table sample below, the return value Document is an object.
- A description, which defines what the method does

The following sample Methods table lists the parameters for the `add` method for a Photoshop CS6 document.

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter type</th>
<th>Returns</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>[width]</td>
<td>UnitValue</td>
<td>Document</td>
</tr>
<tr>
<td></td>
<td>[, height]</td>
<td>UnitValue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[, resolution]</td>
<td>number (double)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[, name]</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[, model]</td>
<td>NewDocumentMode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[, initialFill]</td>
<td>DocumentFill</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[, pixelAspectRatio]</td>
<td>number (double)</td>
<td></td>
</tr>
</tbody>
</table>

In the preceding table:

- All of the parameters are optional, as indicated by the square brackets.

- The `width` and `height` parameters default to the current ruler units, and therefore the data type is listed as `UnitValue`. In other words, if the current vertical ruler unit is inches and the horizontal ruler unit is centimeters, the following statement will create a document that is 5 inches wide and 7 centimeters tall:

  AS:    make document with properties {width:5, height:7}

  JS:    app.documents.add(5, 7)

  VBS:   appRef.Documents.Add(5, 7)

- `mode` and `initialFill` take constant values.

The following script statements define values for each of the parameters listed in the sample methods table.
**AS**

```
make document with properties {width:5, height:7, resolution:72," name:"Diary",
mode:bitmap, initial fill:transparent, pixel aspect ratio: 4.7}
```

**JS**

```
app.documents.add(5, 7, 72, "Diary", NewDocumentMode.BITMAP,
DocumentFill.TRANSPARENT, 4.7)
```

**VBS**

```
appRef.Documents.Add(5, 7, 72, "Diary", 5, 3, 4.7)
```
Advanced Scripting Techniques

Most scripts do not proceed sequentially from beginning to end. Often, scripts take different paths depending on data gleaned from the current document, or they repeat commands multiple times. Control structures are the script language features that enable your scripts to do such things.

Conditional statements

if statements

If you could talk to your Adobe application, you might say, “If the document has only a single layer, then create another layer.” This is an example of a conditional statement. Conditional statements make decisions—they give your scripts a way to evaluate something, such as the number of layers, and then act according to the result. If the condition is met, then the script performs the action included in the if statement. If the condition is not met, then the script skips the action included in the if statement.

Each of the following scripts opens a document and then checks whether the document contains a single layer. If only one layer exists, the script adds a layer and sets the new layer’s fill opacity to 65%.

AS

An if statement in AS begins with the word if, followed by the comparison phrase in parentheses, followed by the word then. You must close the if statement with end if.

tell application "Adobe Photoshop CS6"
   --modify the hard-drive name at the beginning of the filepath, if needed
   set myFilepath to alias "c:Applications:Adobe Photoshop CS6:Samples:Ducky.tif"
   open myFilepath
   set myDoc to current document
   tell myDoc
      if (art layer count = 1) then
         set myLayer to make art layer with properties {fill opacity:65}
      end if
   end tell
end tell

NOTE: AS uses a single equal sign (=) for comparing values.

Now close Ducky.tif and try the script again, but change the if statement to the following:

if (art layer count < 1) then

JS

An if statement in JS begins with the word if, followed by the comparison phrase in parentheses. Enclose the action in the if statement in curly braces ({}).

var myDoc = app.open(File("/c/Program Files/Adobe/Adobe Photoshop CS6/Samples/Ducky.tif"));
if(myDoc.artLayers.length == 1){
   var myLayer = myDoc.artLayers.add()
   myLayer.fillOpacity = 65
}
**NOTE:** JavaScript uses a double equal sign (==) for comparing values, as opposed to the single equal sign (=) used for assigning values to properties or variables.

Now close Ducky.tif and try the script again, but change the if statement to the following:

```javascript
if(myDoc.artLayers.length < 1){
```

**VBS**

An if statement in VBS begins with the word If, followed by the comparison phrase, followed by the word Then. You must close the if statement with End If.

```vbs
Set appRef = CreateObject("Photoshop.Application")
Set myDoc = appRef.Open("/c/Program Files/Adobe/Adobe Photoshop CS6/Samples/Ducky.tif")
If myDoc.ArtLayers.Count = 1 Then
    Set myLayer = myDoc.ArtLayers.Add
    myLayer.FillOpacity = 65
End If
```

**NOTE:** VBS uses a single equal sign for both comparing and assigning values.

Now close Ducky.tif and try the script again, but change the if statement to the following:

```vbs
If myDoc.ArtLayers.Count < 1 Then
```

### if else statements

Sometimes, you might have a slightly more complicated request, such as, "If the document has one layer, set the layer's fill opacity to 50%—but if the document has two or more layers, set the fill opacity of the active layer to 65%." This kind of situation calls for an if else statement.

**AS**

```applescript
tell application "Adobe Photoshop CS6"
    --modify the hard-drive name at the beginning of the filepath, if needed
    set myFilepath to alias "c:Applications:Adobe Photoshop CS6:Samples:Ducky.tif"
    open myFilepath
    set myDoc to current document
    tell myDoc
        if (count of art layers < 2) then
            set fill opacity of current layer to 50
        else
            set fill opacity of current layer to 65
        end if
    end tell
end tell
```

**JS**

```javascript
var myDoc = app.open(File("/c/Program Files/Adobe/Adobe Photoshop CS6/Samples/Ducky.tif"));
if(myDoc.artLayers.length < 2){
    myDoc.activeLayer.fillOpacity = 50
}
else{
    myDoc.activeLayer.fillOpacity = 65
}
```

**VBS**

```vbs
Set appRef = CreateObject("Photoshop.Application")
Set myDoc = appRef.Open("/c/Program Files/Adobe/Adobe Photoshop CS6/Samples/Ducky.tif")
If myDoc.ArtLayers.Count < 2 Then
    myDoc.ActiveLayer.FillOpacity = 50
```
You may want your script to find and change all objects of a certain type. For example, your document may have some visible layers and some invisible layers, and you want to make all of the layers visible. You would like this script to work for several documents, but your documents have varying numbers of layers.

This is a situation in which a repeat statement (AS) or a loop (JS and VBS) comes in handy. A loop “walks” through a collection of objects and performs an action on each object.

To use the scripts in this section, open your Adobe application and create a document that has at least nine layers. Make some of the layers visible, and hide other layers. Save the document, and then run the script, substituting the name of your application and the layer object name in your application's DOM.

The basic principle behind each of these loops is that the script identifies the first layer in the element or collection and sets the layer's visibility to true, then identifies the next layer and repeats the action, and then identifies the following layer until each layer has been acted upon.

**AS**

```
tell application "Adobe Illustrator CS6"
    set myDoc to current document
    tell myDoc
        set myLayerCount to (count layers)
        set myCounter to 1
        repeat while myCounter <= (myLayerCount + 1)
            set myLayer to layer myCounter
            set myLayer with properties {visible:true}
            --the next statement increments the counter to get the next layer
            set myCounter to myCounter + 1
        end repeat
    end tell
end tell
```

This script uses two variables, myLayerCount and myCounter, to identify a layer and then increment the layer number until all layers in the document have been identified.

**JS**

```
var myDoc = app.activeDocument
var myLayerCount = myDoc.layers.length
for(var myCounter = 0; myCounter < myLayerCount; myCounter++)
    {var myLayer = myDoc.layers[myCounter]
    myLayer.visible = true}
```

This script uses a for loop, which is one of the most common techniques in JavaScript. Like the AppleScript above, the script uses two variables, myLayerCount and myCounter, to identify a layer and then increment the layer number until all layers in the document have been identified. The increment takes place in the third statement within the for statement: myCounter++. The ++ syntax adds 1 to the current value, but does not add 1 until the loop's action has been done.

The for loop in this script would say the following in plain English:

1. Begin with the value of myCounter at 0.
2. If the value of myCounter is less than the value of myLayerCount, then use the value of myCounter as the index for the layer assigned to myLayer, and set the visibility of myLayer to true.
3. Add 1 to the value of myCounter, and then compare myCounter’s new value to the value of myLayerCount.

4. If myCounter is still less than myLayerCount, use the new value of myCounter as the index of myLayer and set the visibility of myLayer to true, then add 1 to the value of myCounter.

5. Repeat until myCounter is no longer less than myLayerCount.

VBS

```vbs
Set appRef = CreateObject("Illustrator.Application")
Set myDoc = appRef.ActiveDocument
For Each object in myDoc.Layers
    object.Visible = True
Next
```

The For Each Next loop in VBScript simply tells the application to set the Visible property of each object in the Layers collection in the active document to True. Notice that the collection is identified by the containment hierarchy of parent objects (in this case by the variable myDoc) followed by the collection name, which is the plural form of the object name (in this case Layers).

**Note:** The object named in the loop can be anything. The script works the same if you substitute x for object, as in the following script:

```vbs
Set appRef = CreateObject("Illustrator.Application")
Set myDoc = appRef.ActiveDocument
For Each x in myDoc.Layers
    x.Visible = True
Next
```

More information about scripting

Each scripting language contains many more devices and techniques for adding power and complexity to your scripts. To continue learning how to script your Adobe applications, please refer to the Adobe scripting guide for your application. Also, see Chapter 6, “Bibliography.”
5 Troubleshooting

This chapter explains how to interpret some basic error messages that you may receive when you run a script.

Reserved words

Script Editor and the ESTK, as well as many other scripting editors, highlight certain words when you type them.

For example, the Boolean values true and false are always highlighted. Other examples are listed below.

**AS**
- tell
- end
- with
- set

**JS**
- var
- if
- else
- with

**VBS**
- Dim
- Set
- MsgBox

These highlighted words are reserved by the scripting language for special purposes and cannot be used as variable names. You can use reserved words as part of a string, because they are enclosed in quotes. You can also use them in comments, because comments are ignored by the scripting engine.

If your script indicates a syntax error, check to make sure you have not improperly used a reserved word. For a full list of reserved words in your scripting language, refer to one of the resources listed in Chapter 6, “Bibliography.”

AppleScript Script Editor error messages

When your AppleScript script has an error, the Script Editor highlights the offending part of the script and displays an error message.

Check the highlighted portion of the script for spelling and punctuation. If you do not find an error in the highlighted text, check the text that immediately precedes the highlight. If the preceding text contains an error, the error may have caused the script engine to expect something other than what it found in the highlighted section.

Some common error messages are explained below.

- **Can't get object** — Usually, you have not adequately defined the object in the containment hierarchy. Try adding in parent-object (where parent-object is the object that contains the object indicated...
in the error message) after the object name in your script, or create a nested `tell` statement that names the parent object.

- **Expected "" but found end of script** — Make sure all quotes are closed around strings.
- **Requested property not available for this object** — Check the spelling of all properties.

**Tip:** Choose **Result Log** at the bottom of the Script Editor window to view your script’s progress line by line.

### ESTK error messages

The ESTK alerts you to errors in several ways:

- If your script contains a syntax error, the script does not run and the offending section of the script is highlighted in gray. Often, a description of the problem is displayed in the status bar at the bottom of the ESTK window.

When a syntax error occurs, check the following:

- Make sure your use of upper and lower case is correct. Remember, all terms in JavaScript (except enumeration names) begin with a lowercase letter and use upper case for the first letter in each word in a combined term, such as `artLayer`.

  Also, remember that variable names are case-sensitive.

- Close all parentheses, curly braces, quotes. Make sure each of these are in pairs.

- Make sure quote marks are straight quotes. Also, don’t mix single and double quotes. For example:

  Incorrect: `myDoc.name = "My Document"
  Correct: `myDoc.name = ‘My Document’
  Correct: `myDoc.name = "My Document"

**Note:** Some syntax errors, such as curly quotes or smart quotes, are highlighted in red. The status bar message says simply “Syntax error”. Make sure you use straight quotes.

- If your script contains a runtime error, such as an object that is not correctly identified or a property that does not exist for the object that is trying to use it, the offending statement is highlighted but the script keeps running, as indicated by the swirling icon in the lower right corner. Additionally, the error is described both in the JavaScript Console pane and in the status bar.

When a runtime error occurs:

- Choose **Debug > Stop**, or press **Shift+F5** to stop the script.

- Look in the JavaScript Console to find the nature of the error. The following brief descriptions of some common error messages can help you know where to start.

  *element is undefined* — If the undefined element is a variable, make sure the variable name is spelled correctly and uses the correct case. Also, make sure the variable has been either defined with a `var` statement or assigned a value.

  If the undefined element is a string value, make sure the value is in quotes.
undefined is not an object — Make sure the object in the highlighted statement is identified correctly in the containment hierarchy. For example, if the object is a layer, make sure you have defined which document contains the layer. For document objects, it may be necessary to include the parent object app.

VBScript error messages

When your VBScript script contains an error, a Windows Script Host displays an error message that identifies the line in which the error occurred and the position within the line where the offending syntax or object begins.

This message indicates the problem is located in at the beginning of line 3 in the script
This chapter contains a list of scripting books for beginners. This is only a partial list. You can also search the Internet for online tutorials in your scripting language.

**AppleScript**

For further information and instruction in using the AppleScript scripting language, see these documents and resources:


**JavaScript**

For further information and instruction in using the JavaScript scripting language, see these documents and resources:


**VBScript**

For further information and instruction in using VBScript and the VBSA scripting language, see these documents and resources:

- Microsoft Developers Network (MSDN) scripting website: [msdn.microsoft.com/scripting](http://msdn.microsoft.com/scripting)
Index

A
actions, 5
alert boxes, 19
AppleScript
definition, 6
dictionaries, 31
first script, 7
web site, 50
arguments
definition, 9
using, 23
arrays, 29
creating, 29
defined, 16

B
bibliography, 50
Boolean, 16

C
commands
properties, 23
using, 23
viewing in AS dictionaries, 31, 32, 33
comments, 27
conditional statements, 43
constants
defined, 16
using, 20
containment hierarchy, 9, 12
in scripting references, 39

datatypes, 16
dialogs, 19
dictionaries, 31
DOM
definition, 9
viewing, 10

elements
viewing in scripting references, 38

enumerations
defined, 16
using, 20
ESTK
default location, 7
troubleshooting in, 48
viewing JS object model, 33
ExtendScript
definition, 7

I
if else statements, 44
if statements, 43
Illustrator, See Adobe Illustrator
index
definition, 13
numbering schemes, 14

J
JavaScript
advantages of, 7
case usage, 15
definition, 6
first script, 8
JavaScript Tools Guide, 7

L
long lines, 28
loops, 45

M
macros, 5
methods
arguments, 23
definition, 9
using, 23
viewing in scripting references, 41
viewing in VBS type libraries, 37

O
objects
active, 15
collections, 13
current, 15
definition, 9
elements, 13
parent, 10
references, 10
using, 9
viewing in AS dictionaries, 31, 33
viewing in scripting references, 38
viewing in VBS type libraries, 34

P
parameters
definition, 9
direct (AS), 24
labeled (AS), 24
optional, 23
required, 23
using, 23
using multiple, 24
viewing in scripting references, 41
parent object, 10
properties
datatypes, 16
definition, 9
multiple values, 16
read-only, 19
read-write, 19
using, 16
viewing in AS dictionaries, 31
viewing in scripting references, 39
viewing in VBS type libraries, 35

S
script comments, 27
Script Editor
AppleScript dictionaries, 31
default location, 6
troubleshooting in, 47
scripting
about, 6
definition, 6
using, 5
scripts
running automatically, 7
Startup folder, 7
strings, 16

T
tell statements (AS), 26