



Photoshop 7.0 Scripting Guide



ADOBE SYSTEMS INCORPORATED

Corporate Headquarters


345 Park Avenue

San Jose, CA 95110-2704

(408) 536-6000

<http://partners.adobe.com>

March 2002



Adobe Photoshop Scripting Guide

Copyright 1991–2002 Adobe Systems Incorporated.
All rights reserved.

The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in this document. The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.

Adobe, Photoshop, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Apple, Macintosh, and Mac are trademarks of Apple Computer, Inc. registered in the United States and other countries. Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are the property of their respective owners.



Table of contents

Chapter 1	Introduction	5
1.1	About this manual	5
1.2	What is scripting?	6
1.3	Why use scripting?	6
1.4	What about actions?	6
1.5	System requirements	6
1.6	JavaScript	7
1.7	Choosing a scripting language	7
1.8	Legacy COM scripting	8
Chapter 2	Scripting basics	9
2.1	Documents as objects	9
2.2	Object model concepts	9
2.3	Documenting scripts	11
2.4	Values	12
2.5	Variables	14
2.6	Operators	16
2.7	Commands and methods	17
2.8	Handlers, subroutines and functions	20
2.9	The Scripts menu	21
2.10	Testing and troubleshooting	23
Chapter 3	Scripting Photoshop	33
3.1	Photoshop scripting guidelines	33
3.2	Viewing Photoshop objects, commands and methods	34
3.3	Your first Photoshop script	36
3.4	Object references	42
3.5	Working with units	44
3.6	Executing JavaScripts from AS or VB	48
3.7	The Application object	50

3.8 Document object	54
3.9 Layer objects	58
3.10 Text item object	64
3.11 Selections	67
3.12 Working with Filters	74
3.13 Channel object	75
3.14 Color objects	77
3.15 History object.	80
3.16 Clipboard interaction	81
3.17 Action Manager scripting.	84

1

Introduction

1.1 About this manual

This manual provides an introduction to scripting Adobe Photoshop 7.0 on Mac OS and Windows®. Chapter one covers the basic conventions used in this manual and provides an overview of requirements for scripting Photoshop.

Chapter two covers basic scripting terms, concepts and techniques. Experienced AppleScript writers and Visual Basic programmers may want to skip to Chapter three for specifics on scripting Photoshop.

1.1.1 Conventions in this guide

Code and specific language samples appear in monospaced courier font:

```
documents.add( ) ;
```

IMPORTANT: *Most code examples in this manual are brief snippets that will not work as fully functional scripts.*

Several conventions will be used when referring to AppleScript, Visual Basic and JavaScript. Please note the following shortcut notations:

- AS stands for AppleScript
- VB stands for Visual Basic
- JS stands for JavaScript

The term “commands” will be used to refer both to commands in AppleScript and methods in Visual Basic and JavaScript.

When referring to specific properties and commands, the manual will follow the AppleScript naming convention for that property and the Visual Basic and JavaScript names will appear in parenthesis. For example:

“The `display dialogs` (`DisplayDialogs/displayDialogs`) property is part of the Application object.”

In this case, `display dialogs` refers to the AppleScript property, `DisplayDialogs` refers to the Visual Basic property and `displayDialogs` refers to the JavaScript property.

1.2 What is scripting?

A script is a series of commands that tells Photoshop to perform a set of specified actions. These actions can be simple, and affect only a single object in the current document, or complex and affect many objects in a Photoshop document. The actions can call Photoshop alone or also invoke other applications.

Scripts can automate repetitive tasks and be used as a creative tool to streamline tasks that might be too time consuming to do manually. For example, you could write a script to generate a number of localized versions of a particular image; or to gather information about the various color profiles used by a collection of images.

1.3 Why use scripting?

Graphic design is a field characterized by creativity, but aspects of the actual work of illustration and image manipulation are anything but creative. Scripting provides a tool to help save time spent on repetitive production tasks such as resizing or reformatting documents.

Start with short, simple scripts to save a few seconds a day, and move on to more involved scripts. Any repetitive task is a good candidate for a script. Once you can identify the steps and conditions involved in performing the task, you're ready to write a script to take care of it.

1.4 What about actions?

Photoshop actions are different from scripts. A Photoshop action is a series of tasks you have recorded while using the application—menu choices, tool choices, selection, and other commands. When you “play” an action, Photoshop performs all of the recorded commands.

Actions and scripts are both ways of automating repetitive tasks, but they work very differently.

- You cannot add conditional logic to an action. Unlike a script, actions cannot make decisions based on the current situation.
- A single script can target multiple hosts. Actions can't. For example, you could target both Photoshop and Illustrator in the same script.

1.5 System requirements

The language you use to write scripts depends on your operating system: AppleScript for Mac; Visual Basic for Windows; or JavaScript, a cross-platform scripting language that can run on either Windows or Mac. While the scripting systems differ, the ways that they work with Photoshop are very similar.

1.5.1 Mac

Any system that runs Photoshop 7.0 will support scripting. You will also need AppleScript and a script editor installed. AppleScript and the Script Editor application from Apple are included with the Mac OS. For Mac OS 9.X the default location for the Script Editor application is the Apple Extras folder. For Mac OS X, they can be found in the Applications folder. If these items are not installed on your system, reinstall them from your original system software CD-ROM.

As your scripts become more complex, you may find the need for debugging and productivity features not found in the Script Editor. There are many third-party script editors that can write and debug Apple Scripts. Please check <http://www.apple.com/applescript> for more details.

We use the Script Editor from Apple in this manual.

1.5.2 Windows

Any Windows system that runs Photoshop 7.0 will support scripting. You will also need either the Windows Scripting Host, Microsoft Visual Basic, or one of the applications that contains a Visual Basic editor. Most Windows systems include the Windows Scripting Host. If you do not have Windows Scripting Host or would like more information about Windows Scripting Host, visit the Microsoft Windows Script Technologies Web site at (<http://msdn.microsoft.com/scripting/>).

We use Microsoft Visual Basic in this manual.

1.6 JavaScript

In addition to writing AppleScripts and Visual Basic scripts, you can also write cross-platform JavaScripts using any text editor. The easiest way to run your JavaScripts is to use the “Scripts” menu which is installed with Photoshop Scripting Support.

See section 2.9, “The Scripts menu” on page 21 for more information.

1.7 Choosing a scripting language

Your choice of scripting language is determined by two trade-offs:

1. Do you need to run the same script on both Macintosh and Windows computers?
2. Do you need to control multiple applications from the same script?

JavaScript is a cross-platform language that can work with Scripting Support for Photoshop 7.0 on both platforms. The same script will perform identically on Windows and Macintosh computers. However, JavaScript is invoked from a menu selection within Photoshop and lacks

the facilities to directly address other applications. For example, you cannot easily write a JavaScript to manage workflows involving Photoshop and a database management program.

AppleScript and Visual Basic are only offered on their respective platforms. However, you can write scripts in those languages to control multiple applications. For example, you can write an AppleScript that first manipulates a bitmap in Photoshop and then commands a web design application to incorporate it. This same cross-application capability is also available with Visual Basic on Windows.

You may also use other scripting languages when working with Photoshop Scripting Support. On Mac OS, any language which lets you send Apple events, such as MacPerl, TCL, or Latenight Software's JavaScript component, can be used to script Photoshop.

On Windows, any language which is COM aware can be used to script Photoshop. This includes languages available in Windows Scripting Host, such as VBScript and JScript, as well as other scripting languages like Perl, Tcl/Tk, and Python.

1.8 Legacy COM scripting

Photoshop 5, Photoshop 6 and Photoshop 7 support COM scripting without the optional Scripting Support plug-in. This scripting interface is described in the Photoshop SDK documentation in the "OLE Automation Programming Guide."

You can use both the old style COM scripts and the new style COM scripts with Photoshop 7.0, but you have to modify the way that you refer to the Photoshop application object in your old scripts after you install ScriptingSupport.

Typically you would create a Photoshop application instance by saying:

```
Set App = CreateObject("Photoshop.Application")
```

after installing ScriptingSupport, you must change the above code to

```
Set App = CreateObject("Photoshop.Application.7")
```

Note that the latter version will work both with and without ScriptingSupport installed.

2

Scripting basics

2.1 Documents as objects

If you use Photoshop, then you create documents, layers, channels and design elements and can think of a Photoshop document as a series of layers and channels — or objects. Automating Photoshop with scripting uses the same object-oriented way of thinking.

The heart of a scriptable application is the object model. In Photoshop, the object model is comprised of documents, layers and channels. Each object has its own special properties, and every object in a Photoshop document has its own identity.

This chapter covers the basic concepts of scripting within this object-oriented environment.

2.2 Object model concepts

The terminology of object oriented programming can be hard to understand, at first. “Objects” belong to “classes” and have “properties” you manipulate using “commands” (AppleScript) or “methods” (Visual Basic and JavaScript). What do these words mean in this context?

Here’s a way to think about objects and their properties as an object model. Imagine that you live in a house that responds to your commands. The house is an object, and its properties might include the number of rooms, the color of the exterior paint or the date of its construction.

Your house can also contain other objects within. Each room, for example, is an object in the house, while each window, door, or appliance is an object inside of the room. And each object can respond to various commands according to its capabilities.

Now apply this object model concept to Photoshop. The Photoshop application is the house, its documents are the rooms, and the objects in your documents are the windows and doors. You can tell Photoshop documents to add and remove objects or manipulate individual objects.

2.2.1 Object classes

Objects with the same properties and behaviors are grouped into “classes.” In the house example, windows and doors belong to their own classes because they have unique properties. In Photoshop, every type of object— document, art layer, etc.—belongs to its own class, each with its own set of properties and behaviors.

2.2.2 Object inheritance

Object classes may also “inherit,” or share, the properties of a parent, or superclass. When a class inherits properties, we call that class a child, or subclass of the class from which it inherits properties. So in our house example, windows and doors are subclasses of an openings class, since they are both openings in a house. In Photoshop, art layers, for example, inherit from the layer class.

Classes will often have properties that aren’t shared with their superclass. In our house, both a window and door inherit an opened property from the opening class, but a window has a number of panes property which the Opening class doesn’t. In Photoshop, art layers, for example, have the property grouped which isn’t inherited from the Layer class.

2.2.3 Object elements or collections

Object elements (AppleScript) or collections (Visual Basic, JavaScript) are objects contained within other objects. For example, rooms are elements (or collections) of our house, contained within the house object. In Photoshop, documents are elements of the application object, and layers are elements of a document object. To access an element (or member of a collection), you use an index. For example, to get the first document of the application you write:

```
AS: document 1
```

```
VB: appRef.Documents (1)
```

```
JS: documents[0];
```

IMPORTANT: *Indices in AppleScript and Visual Basic are 1 based. JavaScript indicies are 0 based.*

2.2.4 Object reference

The objects in your documents are arranged in a hierarchy like the house object — layers are in layer sets, which are inside a document which is inside Photoshop. When you send a command to a Photoshop object, you need to make sure you send the message to the right object. To do this, you identify objects by their position in the hierarchy — by an object reference. You might, for example, write the following statement.

AppleScript

```
layer 1 of layer set 1 of current document
```

Visual Basic

```
appRef.ActiveDocument.LayerSets(1).Layers(1)
```

JavaScript

```
activeDocument.layerSets[0].layers[0];
```

When you identify an object in this fashion, you're creating an *object reference*. While AppleScript, Visual Basic and JavaScript use different syntax for object references, each gives the script a way of finding the object you want.

2.3 Documenting scripts

Use comments within your scripts to explain what procedures are taking place. It's a quick way to document your work for others and an important element to remember when writing scripts. Comments are ignored by the scripting system as the script executes and cause no run-time speed penalty.

AppleScript

To enter a single-line comment in an AppleScript, type "--" to the left of your description. For multiple line comments, start your comment with the characters "(" and end it with "*")".

```
-- this is a single-line comment
(* this is a
multiple line comment *)
```

Visual Basic

In Visual Basic, enter "Rem" (for "remark") or "'" (a single straight quote) to the left of the comment.

```
Rem this is a comment
' and so is this
```

JavaScript

In JavaScript, use the double forward slash to comment a single line or a /* */ notation for multi-line comments

```
// This comments until the end of the line

/* This comments
this entire
block of text */
```

About long script lines

In some cases, individual script lines are too long to print on a single line in this guide.

AppleScript

AppleScript uses the special character (↵) to show that the line continues to the next line. This continuation character denotes a "soft return" in the script. You can enter this character in the script editor by pressing Option-Return at the end of the line you wish to continue.

Visual Basic

In Visual Basic, you can break a long statement into multiple lines in the Code window by using the line continuation character, which is a space followed by an underscore (`_`).

2.4 Values

Values are the data your scripts use to do their work. Most of the time, the values used in your scripts will be numbers or text.

TABLE 2.1 *AppleScript Values*

Value type:	What it is:	Example:
boolean	Logical true or false.	true
integer	Whole numbers (no decimal points). Integers can be positive or negative.	14
real	A number which may contain a decimal point.	13.9972
string	A series of text characters. Strings appear inside (straight) quotation marks.	"I am a string"
list	An ordered list of values. The values of a list may be any type.	{ 10.0, 20.0, 30.0, 40.0 }
object reference	A specific reference to an object.	current document
record	An unordered list of properties, Each property is identified by its label.	{ name: "you", index: 1 }

TABLE 2.2 *Visual Basic Values*

Value type:	What it is:	Example:
Boolean	Logical true or false	True
Long	Whole numbers (no decimal points). Longs can be positive or negative.	14
Double	A number which may contain a decimal point.	13.9972
String	A series of text characters. Strings appear inside (straight) quotation marks.	"I am a string"
Array	A list of values. Arrays contain a single value type unless the type is defined as Variant.	Array(10.0, 20.0, 30.0, 40.0)
Object reference	A specific reference to an object.	appRef.ActiveDocument

TABLE 2.3 *JavaScript Values*

Value type:	What it is:	Example:
String	A series of text characters. Strings appear inside (straight) quotation marks.	"Hello"
Number	Any number not inside double quotes.	3.7
Boolean	Logical true or false.	true
Null	Something that points to nothing.	null
Object	Properties and methods belonging to an object or array.	activeDocument
Undefined	Devoid of any value	undefined

2.5 Variables

Variables are containers for data. A variable might contain a number, a string, a list (or array), or an object reference. Variables have names, and you refer to a variable by its name. To put data into a variable, assign the data to the variable. The file name of the current Photoshop document or the current date are both examples of data that can be assigned to a variable.

By using variables the scripts you write will be reusable in a wider variety of situations. As a script executes, it can assign data to the variables that reflect the state of the current document and selection, for example, and then make decisions based on the content of the variables.

NOTE: In AppleScript, it is not important to declare your variables before assigning values to them. In Visual Basic and JavaScript, however, it is considered good form to declare all of your variables before using them. To declare variables in Visual Basic, use the `Dim` keyword. To declare variables in JavaScript, use the `var` keyword.

2.5.1 Assigning values to variables

The remainder of this section shows how to assign values to variables.

AS

```
set thisNumber to 10
set thisString to "Hello, World!"
```

VB

```
Option Explicit
Dim thisNumber As Long
Dim thisString As String
thisNumber = 10
thisString = "Hello, World!"
```

The `Dim` statement assigns a value type to the variable, which helps keep scripts clear and readable. Memory is also used more efficiently if variables are declared before use. If you start your scripts in Visual Basic with the line `Option Explicit`, you will have to declare all variables before assigning data to them. You will not have to declare them the next time they are used.

JS

```
var x = 8;
x = x + 4;
var thisNumber = 10;
var thisString = "Hello, World!";
```

The `var` keyword identifies variables the first time that you use the variable. The next time you use the variable you should not use the `var` keyword.

2.5.2 Using variables to store references

Variables can also be used to store references to objects. In AppleScript, a reference is returned when you create a new object in an Photoshop document as shown below:

```
set thisLayer to make new art layer in current document
```

Or you can fill the variable with a reference to an existing object:

```
set thisLayer to art layer 1 of current document
```

Visual Basic works similarly, however, there is an important distinction to note. If you are assigning an *object reference* to a variable you must use the `Set` command. For example, to assign a variable as you create a layer, use `Set`:

```
Set thisLayer = appRef.Photoshop.ActiveDocument.ArtLayers(1)
```

or in reference to an existing layer, since it is also an *object reference*, use `Set`:

```
Set thisLayer = appRef.Photoshop.ActiveDocument.ArtLayers(1)
```

If you are trying to assign a value to a variable in Visual Basic that is not an object reference, do not use `Set`. Use Visual Basic's assignment operator, the equals sign:

```
thisNumber = 12
```

JavaScript looks similar to Visual Basic. To assign a reference to an object, you would write:

```
var docRef = activeDocument;
```

and to assign a value use the following:

```
var thisNumber = 12
```

2.5.3 Naming variables

It's a good idea to use descriptive names for your variables—something like `firstPage` or `corporateLogo`, rather than `x` or `c`. You can also give your variable names a standard prefix so that they'll stand out from the objects, commands, and keywords of your scripting system.

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

2.6 Operators

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

```
docWidth/2
```

would return a value equal to half of the content of the variable `docWidth`. So if `docWidth` contained the number `20.5`, the value returned would be `10.25`.

You can also use operators to perform comparisons (equal to, not equal to, greater than, or less than, etc.). Some operators differ between AppleScript, Visual Basic and JavaScript. Consult your scripting language for operators that may be unique to your OS.

AppleScript and Visual Basic use the ampersand (&) as the concatenation operator to join two strings.

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

would return the string "Pride and Prejudice."

JavaScript uses the "+" operator to concatenate strings.

```
"Pride" + " and Prejudice"
```

would return the string "Pride and Prejudice."

2.7 Commands and methods

Commands (AppleScript) or methods (Visual Basic and JavaScript) are what makes things happen in a script. The type of the object you're working with determines how you manipulate it.

AS

In AppleScript, use the `make` command to create new objects, the `set` command to assign object references to variables and to change object properties, and the `get` command to retrieve objects and their properties.

VB

In Visual Basic, use the `Add` method to create new objects, the `Set` statement to assign object references to Visual Basic variables or properties and the assignment operator (`=`) to retrieve and change object properties.

JS

In JavaScript, use the `add()` method to create new objects, and the assignment operator (`=`) to assign both object references and variables

2.7.1 Conditional statements

Conditional statements make decisions — they give your scripts a way to evaluate something like the blend mode of a layer or the name or date of a history state — and then act according to the result. Most conditional statements start with the word `if` in all three scripting systems.

The following examples check the number of currently open documents. If no documents are open, the scripts display a messages in a dialog box.

AS

```
tell application "Adobe Photoshop 7.0"
    set documentCount to count every document
    if documentCount = 0 then
        display dialog "No Photoshop documents are open!"
    end if
end tell
```

VB

```
Private Sub Command1_Click()  
    Dim documentCount As long  
    Dim appRef As New Photoshop.Application  
    documentCount = appRef.Documents.Count  
    If documentCount = 0 Then  
        MsgBox "No Photoshop documents are open!"  
    End If  
End Sub
```

JS

```
var documentCount = documents.length;  
if (documentCount == 0)  
{  
    alert("There are no Photoshop documents open");  
}
```

2.7.2 Control structures

Control structures provide for repetitive processes, or “loops.” The idea of a loop is to repeat some action, with or without changes each time through the loop, until a condition is met.

Both AppleScript and Visual Basic have a variety of different control structures to choose from. The simplest form of a loop is one that repeats a series of script operations a set number of times.

AS

```
repeat with counter from 1 to 3  
    display dialog counter  
end repeat
```

VB

```
For counter = 1 to 3  
    MsgBox counter  
Next
```

JS

```
for (i = 1; i < 4; ++i)  
{  
    alert(i);  
}
```

A more complicated type of control structure includes conditional logic, so that it loops while or until some condition is true or false.

AS

```
set flag to false
repeat until flag = true
    set flag to button returned of (display dialog "Quit?" -
        buttons {"Yes", "No"}) = "Yes"
end repeat

set flag to false
repeat while flag = false
    set flag to button returned of (display dialog "Later?" -
        buttons {"Yes", "No"}) = "No"
end repeat
```

VB

```
flag = False
Do While flag = False
    retVal = MsgBox("Quit?", vbOKCancel)
    If (retVal = vbCancel) Then
        flag = True
    End If
Loop

flag = False
Do Until flag = True
    retVal = MsgBox("Quit?", vbOKCancel)
    If (retVal = vbOK) Then
        flag = True
    End If
Loop
```

JS

```
var flag = false;
while (flag == false)
{
    flag = confirm("Are you sure?");
}

var flag = false;
do
{
    flag = confirm("Are you sure?");
}
while (flag == false);
```

2.8 Handlers, subroutines and functions

Subroutines, or handlers (in AppleScript) and functions (in JavaScript), are scripting modules you can refer to from within your script. These subroutines provide a way to re-use parts of scripts. Typically, you send one or more values to a subroutine and it returns one or more values.

There's nothing special about the code used in subroutines — they are conveniences that save you from having to retype the same code lines in your script.

AS

```
set flag to DoConfirm("Are you sure?")
display dialog flag as string

on DoConfirm(prompt)
    set button to button returned of (display dialog prompt -
        buttons {"Yes", "No"} default button 1)
    return button = "Yes"
end DoConfirm
```

VB

```
Private Sub ScriptSample_Click(Index As Integer)
    result = DoConfirm("Are you sure?")
    MsgBox (result)
End Sub
```

```
Function DoConfirm(prompt) As Boolean
    buttonPressed = MsgBox(prompt, vbYesNo)
    DoConfirm = (buttonPressed = vbYes)
End Function
```

JS

```
var theResult = DoConfirm( "Are you sure?" );

alert(theResult);

function DoConfirm(message)
{
    var result = confirm(message);
    return result;
}
```

2.9 The Scripts menu

The Scripts menu item is located in the **File > Automate** menu. When the **Scripts...** item is selected, a dialog is presented from which you can select a JavaScript for execution. The scripts listed in the dialog are in the Scripts folder

2.9.1 Scripts folder

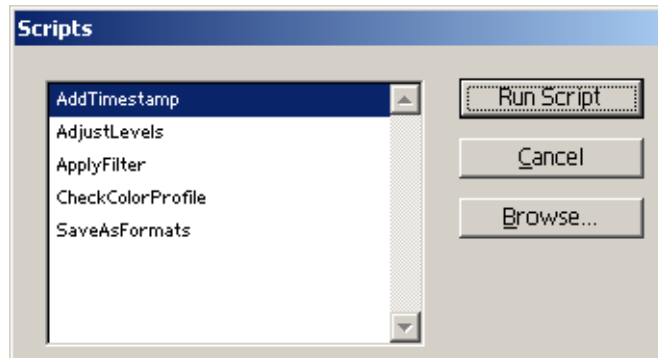
The Scripts folder is in the Photoshop/Presets folder. This folder is created during Scripting Support functionality installation. All JavaScript files placed in the Scripts folder will be available for execution from the Scripts Dialog. Only those script files found in the root of the folder will be listed. Any files in subfolders will be ignored and not listed in the dialog. For both Mac and Windows, a JavaScript file must be saved as a text file with a '.js' file name extension.

2.9.2 The scripts dialog box

To execute a script, select it from the list of scripts and click the "Run Script" button. If there is an error encountered during script execution, an error dialog will be displayed containing the error message returned by the script. If you hold down the option key (alt for Windows), the

“Run Script” button will change to “Debug Script.” Clicking this button will execute the JavaScript in a debug window. See 2.10.3, “JavaScript Debugging” on page 26 for more information.

There is also a “Browse...” button in the Scripts dialog. When this button is clicked, a file navigation dialog will be presented. Use this dialog to locate and execute scripts outside of the Scripts folder. If the option (or alt) key is held down when the “Browse...” button is clicked, the script chosen in the dialog will be executed in debug mode. Here’s the dialog:



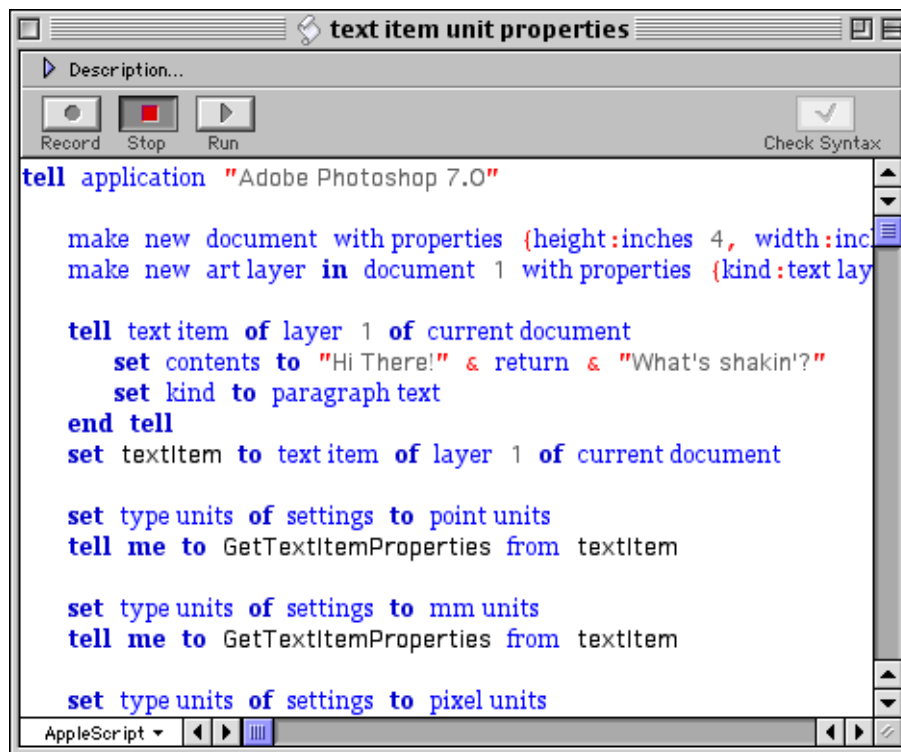
2.10 Testing and troubleshooting

The scripting environments provide tools for monitoring the progress of your script while it is running — which make it easier for you to track down any problems your script might be encountering or causing.

2.10.1 AppleScript debugging

While the basic syntax of your script will be checked when compiled, it is possible to create and compile scripts in AppleScript that will not run properly. The Script Editor Application doesn't have extensive debugging tools, but it does have the an Event Log window.

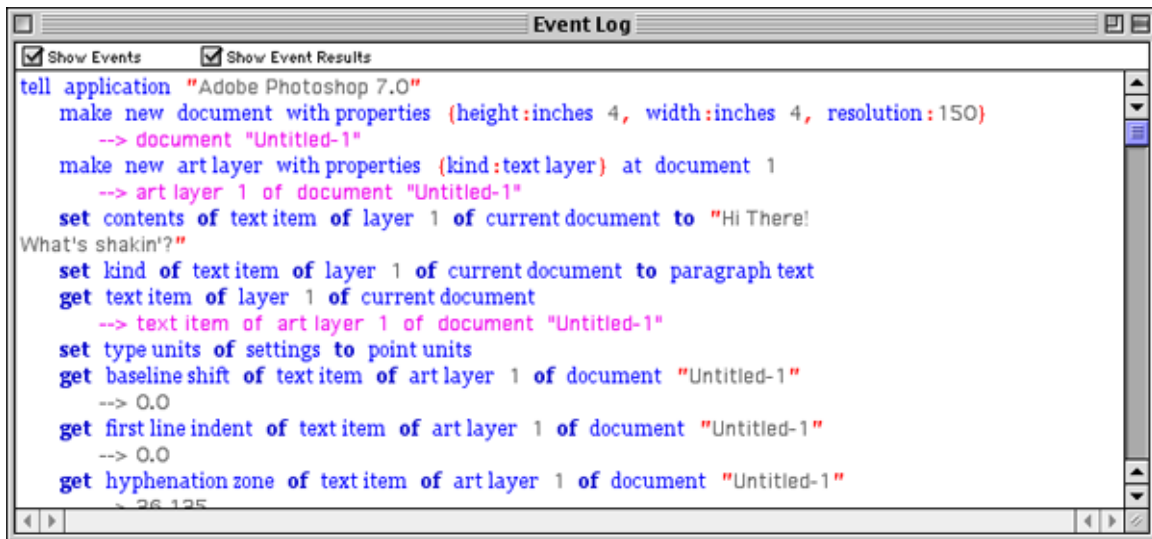
To watch the commands your script sends and the results it receives, choose “Controls > Open Event Log”. The Script Editor displays the Event Log window. Check the “Show Events” and “Show Events Results” options at the top of the “Event Log” window and run your script. As the script executes, you'll see the commands sent to Photoshop, and Photoshop's responses.



You can display the contents of one or more variables in the log window by using the log command.

```
log {myVariable, otherVariable}
```

In addition, the Result window (choose Controls > Show Result) will display the value from the last script statement evaluated. Third-party editors offer additional debugging features

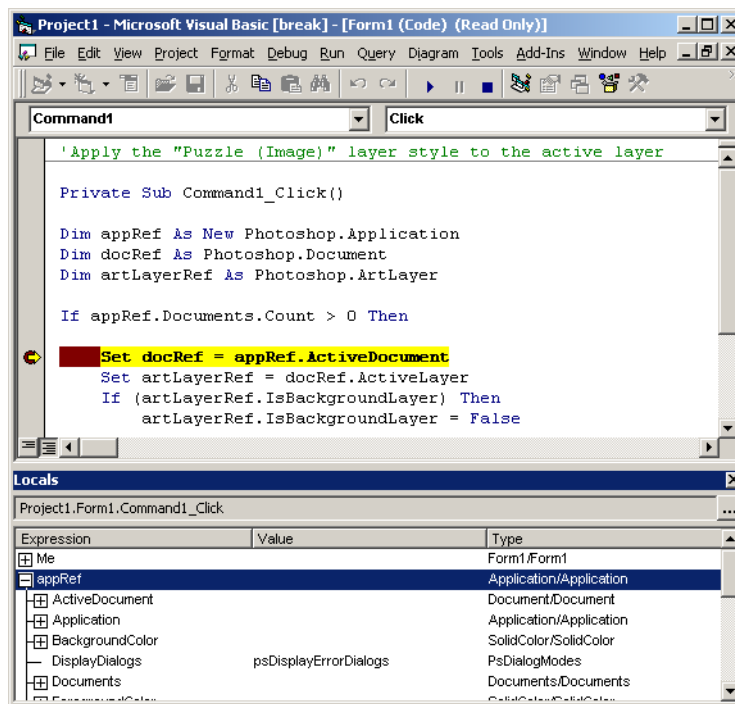


2.10.2 Visual Basic debugging

In Visual Basic, you can stop your script at any point, or step through your script one line at a time. To stop your script at a particular line, select that line in your script and choose “Debug > Toggle Breakpoint”.

When you run the script, Visual Basic will stop at the breakpoint you have set. Choose “Debug > Step Into” (or press F8) to execute the next line of your script, or choose “Run > Start” (or press F5) to continue normal execution of the script.

You can also observe the values of variables defined in your script using the “Watch” window — a very valuable tool for debugging your scripts. To view a variable in the “Watch” window, select the variable and choose “Debug > Quick Watch”. Visual Basic displays the “Quick Watch” dialog box. Click the “Add” button. Visual Basic displays the “Watch” window. If you have closed the “Watch” window, you can display it again by choosing “View > Watch Window.”



Check your Visual Basic documentation for more information. Windows Scripting Host also provides debugging information.

2.10.3 JavaScript Debugging

This section describes the information and controls that the main Script Debugger window provides.

In Photoshop you can use the JavaScript Debugger Window to step through your JavaScript code.

JavaScript can be executed in two different ways: from the UI via the “Scripts...” menu and from AppleScript or VisualBasic via the `do javascript` methods.

When running JavaScript from the UI you must hold down the `option` key on the Mac and `alt` on Windows to activate the debugger. When you hold down this modifier key the “Run Script” button changes to “Debug Script”.

When invoking JavaScript from AppleScript or VB you must set the `show debugger` argument appropriately.

If you set `show debugger` to `never` you will disable debugging. This corresponds to running the JavaScript from the UI without holding down the modifier key.

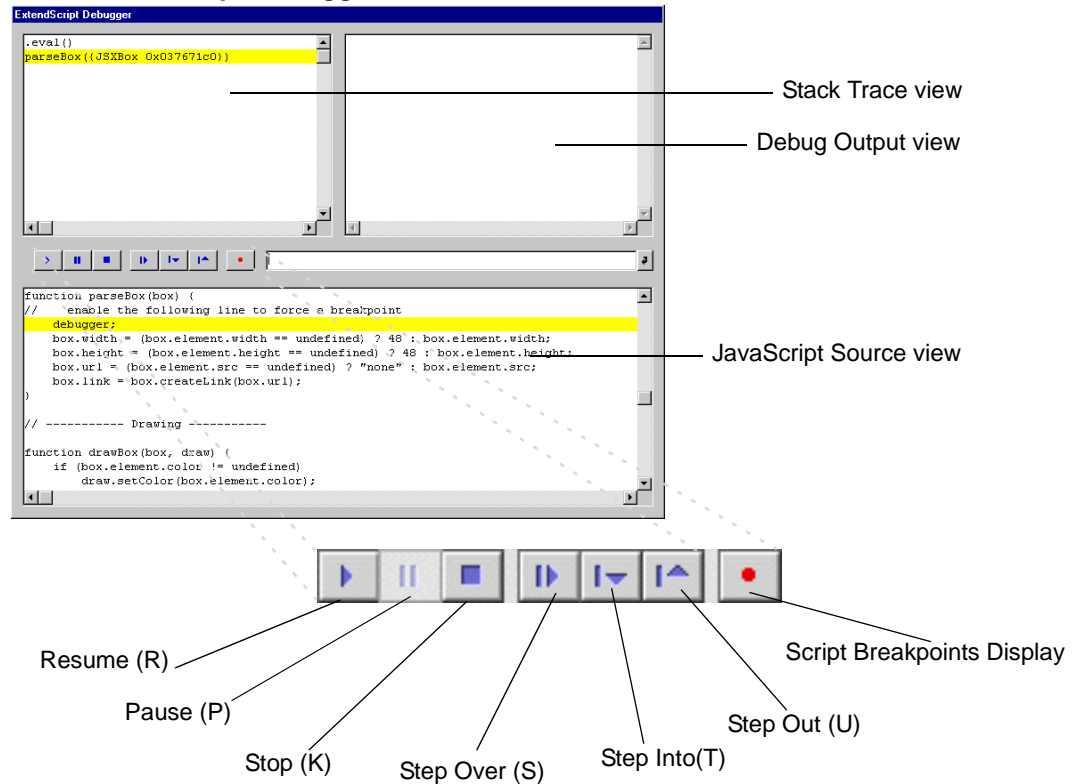
If you set `show debugger` to `on runtime error` your JavaScript will execute normally until it detects a run-time error or until it meets a “debugger;” call (see below for an example). When this happens execution will be stopped and the JavaScript debugger will be shown.

If you set `show debugger` to `at beginning` your JavaScript will halt before executing the first line of JavaScript code and the debugger will be shown. This mode corresponds to executing JavaScripts from the UI while holding down the modifier key.

Viewing Debug Information

The Photoshop Script Debugger window provides three informational views that Figure 2.1 depicts.

FIGURE 2.1 Script Debugger window



The current stack trace appears in the upper-left pane of the script debugger window. This **stack trace view** displays the calling hierarchy at the time of the breakpoint. Double-clicking a line in this view changes the current scope, enabling you to inspect and modify scope-specific data.

All debugging output appears in the upper-right pane of the script debugger window.

The currently-executing JavaScript source appears in the lower pane of the script debugger window. Double-clicking a line in this **JavaScript source view** sets or clears an unconditional breakpoint on that line; that is, if a breakpoint is in effect for that line, double-clicking it clears the breakpoint, and vice-versa.

Controlling Code Execution in the Script Debugger Window

This section describes the buttons that control the execution of code when the Script Debugger window is active. Most of these buttons also provide a keyboard shortcut available as a `Ctrl`-`key` combination on Windows platforms or a `Cmd`-`key` combination on Mac OS platforms.

**Resume**

`Cmd`-`R` (Mac OS)
`Ctrl`-`R` (Windows)

Resume execution of the script with the script debugger window open. When the script terminates, Photoshop closes the script debugger window automatically. Closing the debugger window manually also causes script execution to resume. This button is enabled when script execution is paused or stopped.

**Pause**

`Cmd`-`P` (Mac OS)
`Ctrl`-`P` (Windows)

Halt the currently-executing script temporarily and reactivate the script debugger window. This button is enabled when a script is running.

**Stop**

`Cmd`-`K` (Mac OS)
`Ctrl`-`K` (Windows)

Stop execution of the script and generate a runtime error. This button is enabled when a script is running.

**Step Into**

`Ctrl`-`T` (Mac OS)
`Cmd`-`T` (Windows)

Halt after executing a single JavaScript statement in the script or after executing a single statement in any JavaScript function the script calls.

**Step Over**

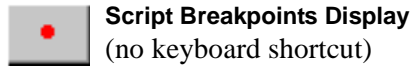
`Ctrl`-`S` (Mac OS)
`Cmd`-`S` (Windows)

Halt after executing a single JavaScript statement in the script; if the statement calls a JavaScript function, execute the function in its entirety before stopping.

**Step Out**

`Ctrl`-`U` (Mac OS)
`Cmd`-`U` (Windows)

When the debugger is paused within the body of a JavaScript function, clicking this button resumes script execution until the function returns. When paused outside the body of a function, clicking this button resumes script execution until the script terminates.



Clicking this button displays the [Script Breakpoints Window](#) shown in [Figure 2.2](#).

Using the JavaScript Command Line Entry Field

You can use the Script Debugger window's command line entry field to enter and execute Javascript code interactively within a specified stack scope. Commands entered in this field execute with a timeout of one second.



Command line entry field. Enter in this field a JavaScript statement to execute within the stack scope of the line highlighted in the **Stack Trace** view. When you've finished entering the JavaScript expression, you can execute it by clicking the command line entry button or pressing the **Enter** key.



Command line entry button. Click this button or press **Enter** to execute the JavaScript code in the command line entry field. Photoshop executes the contents of the command line entry field within the stack scope of the line highlighted in the **Stack Trace** view.

The command line entry field accepts any JavaScript code, making it very convenient to use for inspecting or changing the contents of variables.

NOTE: To list the contents of an *object* as if it were JavaScript source code, enter the `object.toString()` command.

Setting Breakpoints In the Script Debugger Window

When the Photoshop Script Debugger window is active, you can double-click a line in the source view to set or clear a breakpoint at that line. Alternatively, you can click the BP button to display the Script Breakpoints window and set or clear breakpoints in this window.

Setting Breakpoints in JavaScript Code

Adding the `debugger` statement to a script sets an unconditional breakpoint. For example, the following code causes Photoshop to halt and display the script debug window as soon as it enters the `parseBox` function.

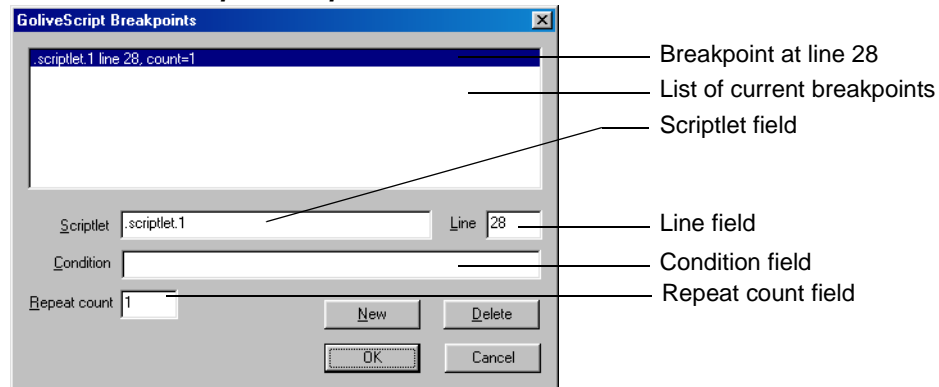
```
function getRatio( docRef )
{
    debugger;
    var theRatio = ( docRef.height ) / ( docRef.width );
    return theRatio;
}

alert( getRatio( activeDocument ) );
```

Script Breakpoints Window

This section describes the information and controls that the Script Breakpoints window provides. Display of the Script Breakpoints window is controlled by the **Script Breakpoints** button in the main script debugger window described on [page 27](#).

FIGURE 2.2 Script Breakpoints window



This dialog displays all defined breakpoints.

This dialog does not display:

- Breakpoints defined by the debugger statement in JavaScript code.
- Temporary breakpoints.

The Script Breakpoints window provides the following controls:

- The **Line** field contains the line number of the breakpoint within the scriptlet.
- The **Condition** field may contain a Javascript expression to evaluate when the breakpoint is reached. If the expression evaluates to `false`, the breakpoint is not executed.
- The **Repeat count** field contains the number of times that the breakpoint must be reached before Photoshop enters the debugger.

Breakpoints set in this window persist across multiple executions of a script. When Photoshop quits, it removes all breakpoints.

2.10.4 Error handling

The following examples show how to stop a script from executing when a specific file cannot be found.

AS

```
--Store a reference to the document with the name "My Document"
--If it does not exist, display an error message
tell application "Adobe Photoshop 7.0"
    try
        set docRef to document "My Document"
        display dialog "Found 'My Document' "

    on error
        display dialog "Couldn't locate document 'My Document'"
    end try
end tell
```

VB

```
Private Sub Command1_Click()
' Store a reference to the document with the name "My Document"
' If the document does not exist, display an error message.
    Dim appRef As New Photoshop.Application
    Dim docRef As Photoshop.Document
    Dim errorMessage As String
    Dim docName As String

    docName = "My Document"
    Set docRef = appRef.ActiveDocument
    On Error GoTo DisplayError
        Set docRef = appRef.Documents(docName)
        MsgBox "Document Found!"
    Exit Sub
DisplayError:
    errorMessage = "Couldn't locate document " & "" & docName & ""
    MsgBox errorMessage
End Sub
```

JS

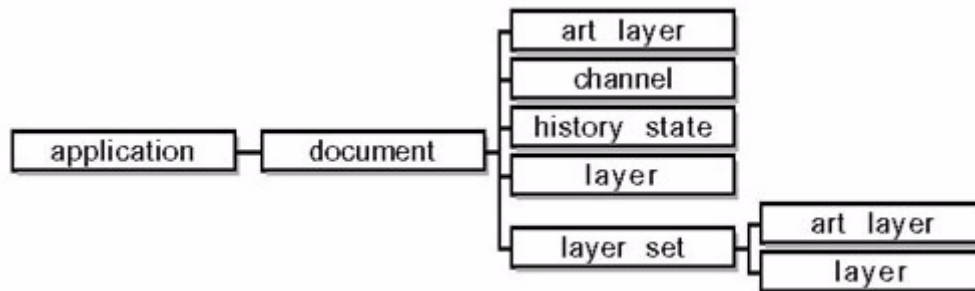
```
try
{
    for (i = 0; i < documents.length; ++i)
    {
        var myName = documents[i].name;
        alert(myName);
    }
}
catch(someError)
{
    alert( "JavaScript error occurred. Message = " + someError );
}
```

3

Scripting Photoshop

3.1 Photoshop scripting guidelines

Once you are used to thinking of Photoshop as an object oriented environment, as discussed in Chapter two, you are ready to move on to writing scripts for the application. It is important to think of the objects in Photoshop as part of an “object containment hierarchy”. The diagram below, read from left to right, illustrates Photoshop’s containment hierarchy.



The following guidelines will also help save debugging time when running Photoshop scripts.

- Before running scripts make sure Photoshop’s Text Tool is not selected and no dialog boxes are displayed to avoid script run-time errors.
- Select documents by name rather than numeric index and set the current document in your script before working on it. Document numbers do not represent their stacking order. See [3.4, “Object references” on page 42](#) for more information.
- In AppleScript always create your document with a name and later get that document by name.

```
-- get the front-most document
set docRef to make new document with properties -
    { height:pixels 144, width:pixels 144, resolution:50,-
      name:"My Document" }
```

- When working in VB or JavaScript, store the document reference to a newly-created document to reuse later.
- When running AppleScripts and two documents are open with the same name, both documents will be modified when the name is referenced. For example, the following script would modify the color profile of all open documents named “MyDocument.”

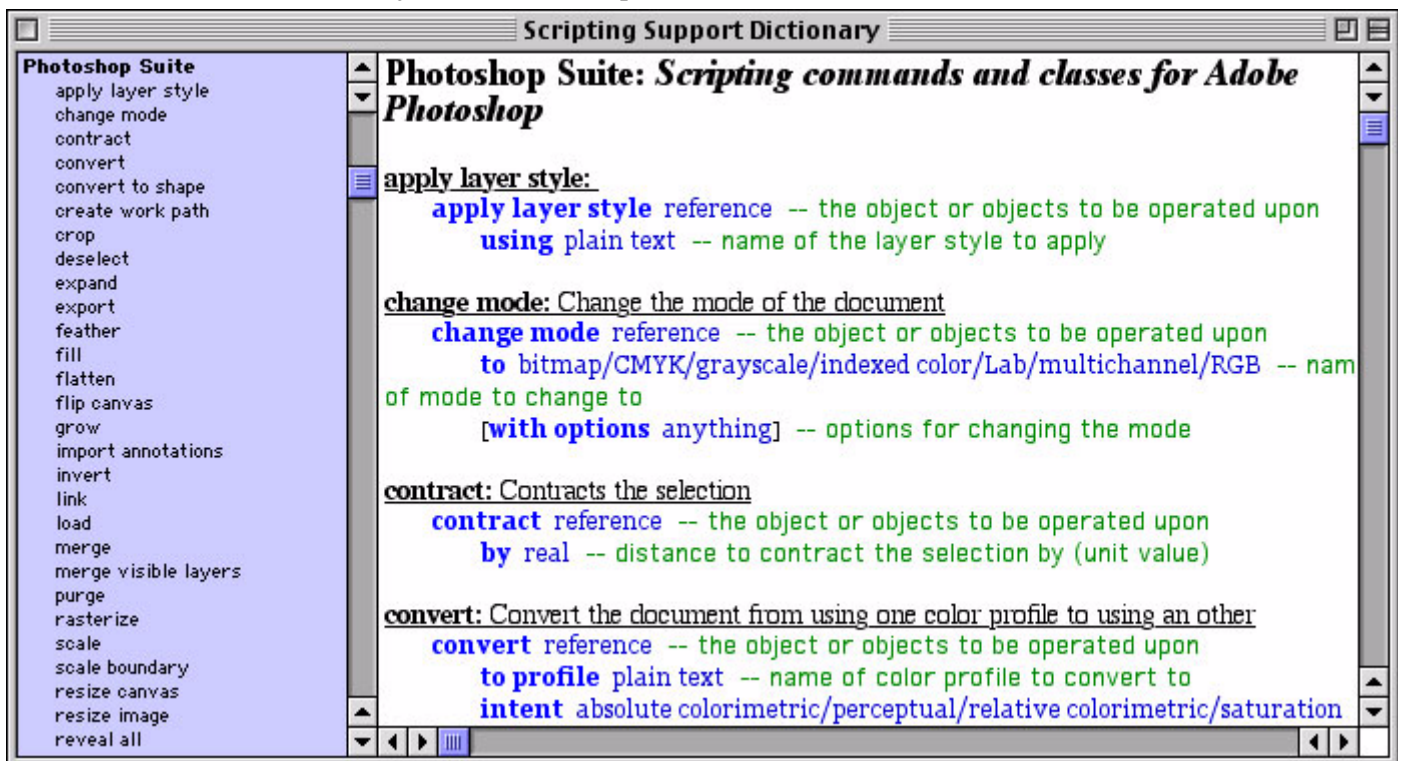
```
tell application "Adobe Photoshop 7.0"
    set color profile kind of document "MyDocument" to none
end tell
```

3.2 Viewing Photoshop objects, commands and methods

This section shows how to view Photoshop’s objects, commands and properties in AppleScript and Visual Basic editors. JavaScript does not include an object browser.

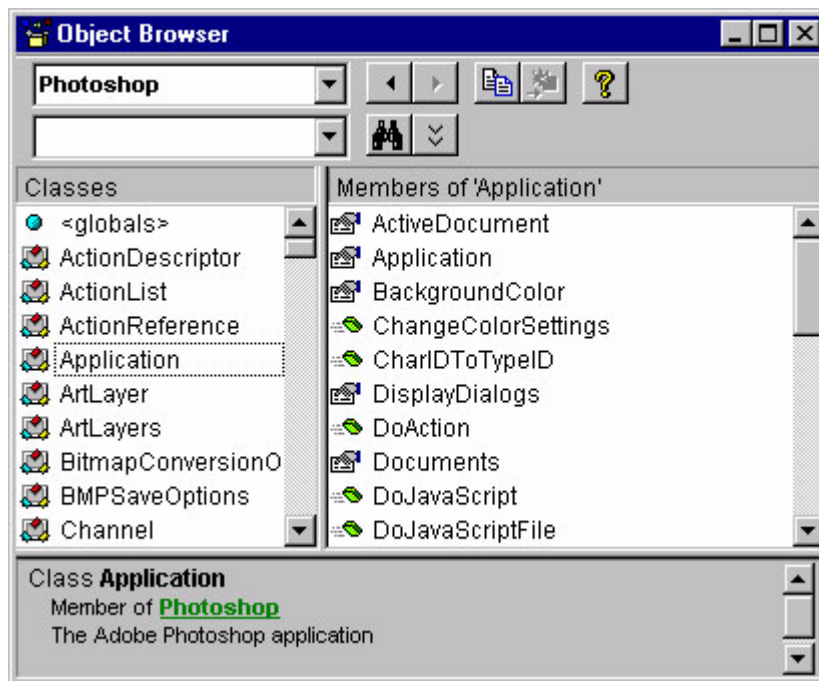
3.2.1 Viewing Photoshop’s AppleScript dictionary

1. Start Photoshop, then your “Script Editor.”
2. In Script Editor, choose “File > Open Dictionary”. Script Editor displays an “Open File” dialog.
3. Find and select the Photoshop application and click the “OK” button. Script Editor displays a list of Photoshop’s objects and commands and the properties and elements associated with each object, as well as the parameters for each command.



3.2.2 Viewing Photoshop's type library (VB)

1. In any Visual Basic project, choose “Project > References.” If you are using a built-in editor in a VBA application, choose “Tools > References.”
2. Turn on the “Adobe Photoshop 7.0 Object Library” option from the list of available references and click the “OK” button. If the library does not appear in the list of available references, then Scripting Support is not installed properly. Reinstall using the Scripting Support installer for Windows.
3. Choose “View > Object Browser.” Visual Basic displays the “Object Browser” window.
4. Choose “Photoshop” from the list of open libraries shown in the top-left pull-down menu.
5. Click an object class or class member to display more information about it.



3.3 Your first Photoshop script

The traditional first project in any programming language is to display the message “Hello World!” In this section, we’ll create a new Photoshop document, then add a text item containing this message with examples in AppleScript, Visual Basic, VBScript and JavaScript.

IMPORTANT: *Before attempting to run these sample scripts make sure you have properly installed the Scripting Support module for Photoshop.*

3.3.1 AppleScript

1. Locate and open Script Editor.
2. Enter the following script. The lines preceded by “--” are comments. They’re included to document the operation of the script and it’s good style to include them in your own scripts. As you look through the script, you’ll see how to create, then address, each object. The AppleScript command `tell` indicates the object that will receive the next message we send.

```
-- Sample script to create a new text item and change its
-- contents.
tell application "Adobe Photoshop 7.0"

    -- Create a new document and art layer.
        set docRef to make new document with properties -
            {width:3 as inches, height:2 as inches}
        set artLayerRef to make new art layer in docRef

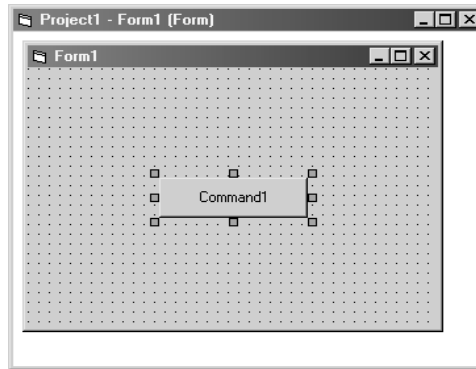
    -- Change the art layer to be a text layer.
        set kind of artLayerRef to text layer

    -- Get a reference to the text item and set its contents.
        set contents of text item of artLayerRef to "Hello, World!"
end tell
```

3. Run the script. Photoshop will create a new document, add a new art layer, change the art layer’s type to text and set the text to “Hello, World!”

3.3.2 Visual Basic

1. Start Visual Basic and create a new project. Add the “Adobe Photoshop 7.0 Object Library” reference to the project, as shown earlier. If you are using a built-in editor in a VBA application, skip to step 4.
2. Add a form to the project.
3. Create a new button on the form. Double-click the button to open the Code window.



4. Enter the following code. The lines preceded by ' (single quotes) are comments, and will be ignored by the scripting system. They're included to describe the operation of the script. As you look through the script, you'll see how to create, then address each object.

```
Private Sub Command1_Click()

    ' Hello World Script
    Dim appRef As New Photoshop.Application

    ' Remember current unit settings and then set units to
    ' the value expected by this script
    Dim originalRulerUnits As Photoshop.PsUnits
    originalRulerUnits = appRef.Preferences.RulerUnits
    appRef.Preferences.RulerUnits = psInches

    ' Create a new 4x4 inch document and assign it to a variable.
    Dim docRef As Photoshop.Document
    Dim artLayerRef As Photoshop.ArtLayer
    Dim textItemRef As Photoshop.TextItem
    Set docRef = appRef.Documents.Add(4, 4)

    ' Create a new art layer containing text
    Set artLayerRef = docRef.ArtLayers.Add
    artLayerRef.Kind = psTextLayer

    ' Set the contents of the text layer.
    Set textItemRef = artLayerRef.TextItem
    textItemRef.Contents = "Hello, World!"

    ' Restore unit setting
    appRef.Preferences.RulerUnits = originalRulerUnits

End Sub
```

5. Save the form.
6. Start Photoshop.
7. Return to Visual Basic and run the program. If you created a form, click the button you created earlier.
8. Run the script. Photoshop will create a new document, add a new art layer, change the art layer's type to text and set the text to "Hello, World!"

3.3.3 VBScript

You don't need to use Visual Basic to run scripts on Windows. Another way to script Photoshop is to use a VBA editor (such as the one that is included in Microsoft Word) or to use Windows Scripting Host.

Most Windows systems include Windows Scripting Host. If you do not have Windows Scripting Host or would like more information about Windows Scripting Host visit the Microsoft Windows Script Technologies Web site at <http://msdn.microsoft.com/scripting/>.

VBScript considerations

Both VBA and Windows Scripting Host use VBScript as their scripting language. The syntax for VBScript is very similar to the Visual Basic syntax. The three main differences relating to the scripts shown in this guide are:

- VBScript is not as strongly typed as Visual basic. In Visual Basic you say:

```
Dim aRef as Photoshop.ArtLayer
```

in VBScript you say:

```
Dim aRef
```

For VBScript simply omit the “as X” part

- VBScript does not support the “as New Photoshop.Application” form.

In Visual Basic you can retrieve the Application object as:

```
Dim appRef as New Photoshop.Application
```

In VBScript you write the following to retrieve the Application object:

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

- VBScript does not support enumerations. Here's an example of how to set the extension type that can later be used save a document.

```
Dim extType As Photoshop.PsExtensionType  
extType = psUppercase
```

In the Visual Basic reference the value of the various enumerated values are specified in a parenthesis after the enumeration name. For example “psConvertToCMYK (3)” means that from Visual Basic you can use the term “psConvertToCMYK” to refer to the CMYK document mode, scripting languages that do not use a typelibrary can use the value 3.

Here's an example VBScript:

```
' Hello World Script
Dim appRef
Set appRef = CreateObject( "Photoshop.Application" )

' Remember current unit settings and then set units to
' the value expected by this script
Dim originalRulerUnits
originalRulerUnits = appRef.Preferences.RulerUnits
appRef.Preferences.RulerUnits = 2

' Create a new 4x4 inch document and assign it to a variable.
Dim docRef
Dim artLayerRef
Dim textItemRef
Set docRef = appRef.Documents.Add(4, 4)

' Create a new art layer containing text
Set artLayerRef = docRef.ArtLayers.Add
artLayerRef.Kind = 2

' Set the contents of the text layer.
Set textItemRef = artLayerRef.TextItem
textItemRef.Contents = "Hello, World!"

' Restore unit setting
appRef.Preferences.RulerUnits = originalRulerUnits
```

To run this script create a text file and copy the script into it. Save the file with a “vbs” extension. Double-click the file to execute.

3.3.4 JavaScript

```
// Hello Word Script

// Remember current unit settings and then set units to
// the value expected by this script
var originalUnit = preferences.rulerUnits;
preferences.rulerUnits = Units.INCHES;

// Create a new 4x4 inch document and assign it to a variable
var docRef = documents.add( 4, 4 );

// Create a new art layer containing text
var artLayerRef = docRef.artLayers.add();
artLayerRef.kind = LayerKind.TEXT;

// Set the contents of the text layer.
var textItemRef = artLayerRef.textItem;
textItemRef.contents = "Hello, World!";

// Release references
docRef = null;
artLayerRef = null;
textItemRef = null;

// Restore original ruler unit setting
preferences.rulerUnits = originalUnit;
```

3.4 Object references

3.4.1 AppleScript

AppleScript uses object references to identify the target object for commands. When working with Photoshop you can identify each item in an object reference using either index or name form. For example, if you have a single document, named “My Document”, open, you could target the document’s first layer, named “Cloud Layer” with either line:

```
layer 1 of document 1
```

or

```
layer "cloud layer" of document "My Document"
```

NOTE: When scripting Photoshop a document's index is not always the same as its stacking order in the user interface. It is possible for document 1 to not be the front-most document. For this reason Photoshop will always return object references identifying documents by name. It is recommended that you always use the name form when identifying documents in your scripts.

An object's index or name also may change as a result of manipulating other objects. For example, when a new art layer is created in the document, it will become the first layer, and the layer that was previously the first layer is now the 2nd layer. Therefore, any references made to layer 1 of current document will now refer to the new layer.

Consider the following sample script:

```
1. tell application "Adobe Photoshop 7.0"
2.   activate
3.   set newDocument to make new document with properties ~
       { width: inches 2, height: inches 3}
4.   set layerRef to layer 1 of current document
5.   make new art layer in current document
6.   set name of layerRef to "My layer"
7. end tell
```

This script will not set the name of the layer referenced on the fourth line of the script. Instead it will set the name created on line five. Try referencing the objects by name as shown below:

```
1. tell application "Adobe Photoshop 7.0"
2.   activate
3.   set newDocument to make new document with properties ~
       { width: inches 2, height: inches 3}
4.   make new art layer in current document with properties {name: "L1" }
5.   make new art layer in current document with properties {name: "L2" }
6.   set name of art layer "L1" of current document to "New Layer 1"
7. end tell
```

3.4.2 Visual Basic and JavaScript

Object references in Visual Basic and JavaScript are fixed and remain valid until disposed or until the host object goes away. The following example shows how to create 2 layers and then rename the first one in Visual Basic.

```
Dim appRef As Photoshop.Application
Dim docRef As Photoshop.Document
Dim layer1Ref As Photoshop.ArtLayer
Dim layer2Ref As Photoshop.ArtLayer

' Set ruler units and create a new document.
Set appRef = New Photoshop.Application
originalRulerUnits = appRef.Preferences.RulerUnits
appRef.Preferences.RulerUnits = psInches
Set docRef = appRef.Documents.Add(4, 4, 72, "My New Document")

' Create 2 new layers and store their return references.
Set layer1Ref = docRef.ArtLayers.Add()
Set layer2Ref = docRef.ArtLayers.Add()

' Change the name of the first layer that was created.
layer1Ref.Name = "This layer was first"

'restore unit values
appRef.Preferences.RulerUnits = originalRulerUnits
```

To do a similar thing in JavaScript you could do:

```
// set ruler units and create new document
originalRulerUnits = preferences.rulerUnits
preferences.rulerUnits = Units.INCHES;
documents.add(4,4,72,"My New Document");
docRef = activeDocument;
layer1Ref = docRef.artLayers.add();
layer2Ref = docRef.artLayers.add();
layer1Ref.name = "This layer was first";

// restore unit setting
preferences.rulerUnits = originalRulerUnits;
```

3.5 Working with units

Photoshop provides two rulers for use when working on a document — a graphics ruler used for most graphical layout measurements and a type ruler which is active when using the type tool. The unit types for these two rulers are set using the `ruler units` (`RulerUnits/rulerUnits`) and `type units` (`TypeUnits/typeUnits`), respectively. These settings correspond to those found in the Photoshop preference dialog under “Edit > Preferences > Units & Rulers.”

The graphics ruler is used for most operations on a document where height, width, or position are specified. The type ruler is used when operating on text items, such as when setting leading or indent values. By changing the settings for each ruler you can work with documents in the measurement system that make the most sense for the project at hand.

3.5.1 Unit values

Photoshop Scripting Support uses unit values for certain properties and parameters. The comments for the Photoshop Scripting Support objects and properties will note where unit values are used.

Because of scripting languages differences, the way you provide a unit value in a script will depend on the language you are using. All languages support plain numbers for unit values. Scripting Support treats these values as being of the type currently specified for the appropriate ruler.

For example, if the ruler units are currently set to inches and the following Visual Basic statement is executed:

```
docRef.ResizeImage 3,3
```

the document's image will be resized to 3 inches by 3 inches. If the ruler units were set to pixels, the image would be 3 pixels by 3 pixels, which is probably not what was intended. To ensure that your scripts produce the expected results you should check and set the ruler units to the type appropriate for your script. After executing a script the original values of the rule settings should be restored if changed in the script. See section 3.5.3, “[Changing ruler and type units](#)” on page 48 for directions on setting unit values.

AppleScript unit considerations

AppleScript provides an additional way of working with unit values. You can provide values with an explicit unit type where unit values are used. When a typed value is provided its type overrides the ruler's current setting.

For example, to create a document which is 4 inches wide by 5 inches high you would write:

```
make new document with properties {width:inches 4, ↵  
height:inches 5}
```

The values returned for a Photoshop property which used units will be returned as a value of the current ruler type. Getting the height of the document created above:

```
set docHeight to height of current document
```

would return a value of 5.0, which represents 5 inches based on the current ruler settings.

In AppleScript, you can optionally ask for a property value as a particular type.

```
set docHeight to height of current document as points
```

This would return a value of 360 (5 inches x 72 points per inch).

IMPORTANT: *Because Photoshop is a pixel-oriented application you may not always get back the same value as you pass in when setting a value. For example, if Ruler Units is set to mm units, and you create a document that is 30 x 30, the value returned for the height or width will be 30.056 if your document resolution is set to 72 ppi. The scripting interface assumes settings are measured by ppi.*

The length unit value types available AppleScript use are listed below:

TABLE 3.1 *AppleScript Length Unit Values*

inches	millimeters
feet	centimeters
yards	meters
miles	kilometers
points	picas
traditional points	traditional picas
ciceros	

The `points` and `picas` unit value types are PostScript points, with 72 points per inch. The `traditional points` and `traditional picas` unit value types are based on classical type setting values, with 72.27 points per inch.

When working with unit values, it is possible to convert, or coerce, a unit value from one value type to another. For example, the following script will convert a point value to an inch value.

```
set pointValue to points 72
set inchValue to pointValue as inches
```

When this script is run the variable `inchValue` will contain `inches 1`, which is 72 points converted to inches. This conversion ability is built in to the AppleScript language.

To use a unit value in a calculation it is necessary to first convert the value to a number (unit value cannot be used directly in calculations). To multiply an inch value write:

```
set newValue to (inchValue as number) * someValue
```

Special unit value types

The unit values used by Photoshop Scripting Support are length units, representing values of linear measurement. Support is also included for pixel and percent unit values. These two unit value types are not, strictly speaking, length values but are included because they are used extensively by Photoshop for many operations and values.

NOTE: In AppleScript you can get and set values as pixels or percent as you would any other unit value type. You cannot, however, convert a pixel or percent value to another length unit value as you can with other length value types. Trying to run the following script will result in an error.

```
set pixelValue to pixels 72
-- Next line will result in a coercion error when run
set inchValue to pixelValue as inches
```

3.5.2 Unit value usage

The following two tables list the properties of the classes and parameters of commands that are defined to use unit values. Unit values for these properties and parameter, with the exception of some text item properties, are based the graphics ruler setting.

TABLE 3.2 *Object Properties*

Object	AppleScript Properties	Visual Basic Properties	JavaScript Properties
Document	height width	Height Width	height width
EPS open options	height width	Height Width	height width
PDF open options	height width	Height Width	height width
lens flare open options	height width	Height Width	height width
offset filter	horizontal offset vertical offset	HorizontalOffset VerticalOffset	horizontalOffset verticalOffset

TABLE 3.2 *Object Properties*

Object	AppleScript Properties	Visual Basic Properties	JavaScript Properties
Text Item	baseline shift*	BaselineShift*	baselineShift*
	first line indent*	FirstLineIndent*	firstLineIndent*
	height	Height	height
	hyphenation zone*	HyphenationZone*	hyphenationZone*
	leading*	Leading*	leading*
	left indent*	LeftIndent*	leftIndent*
	position	Position	position
	right indent*	RightIndent*	rightIndent*
	space before*	SpaceBefore*	spaceBefore*
	space after*	SpaceAfter*	spaceAfter*
	width	Width	width

* Unit values based on type ruler setting

TABLE 3.3 *Command Parameters*

AppleScript	Visual Basic	JavaScript
crop (bounds, height, width)	Document.Crop (Bounds, Height, Width)	document.crop (bounds, height, width)
resize canvas (height, width)	Document.ResizeCanvas (Height, Width)	document.resizeCanvas (height, width)
resize image (height, width)	Document.ResizeImage (Height, Width)	document.resizeImage (height, width)
contract (by)	Selection.Contract (By)	selection.contract (by)
expand (by)	Selection.Expand (By)	selection.expand (by)
feather (by)	Selection.Feather (By)	selection.feather (by)
select border (width)	Selection.SelectBorder (Width)	selection.selectBorder (width)
translate (delta x, delta y)	Selection.Translate (DeltaX, DeltaY)	selection.translate (deltaX, deltaY)

TABLE 3.3 Command Parameters

AppleScript	Visual Basic	JavaScript
translate boundary (delta x, delta y)	Selection.TranslateBoundary (DeltaX, DeltaY)	selection.translateBoundary (deltaX, deltaY)

3.5.3 Changing ruler and type units

The unit type settings of the two Photoshop rulers control how numbers are interpreted when dealing with properties and parameters that support unit values. Be sure to set the ruler units as needed at the beginning of your scripts and save and restore the original ruler settings when your script has completed.

In AppleScript `ruler units` and `type units` are properties of the `settings-object`, accessed through the `Application` object's `settings` property as shown below.

```
set ruler units of settings to inch units
set type units of settings to pixel units
set point size of settings to postscript size
```

In Visual Basic and JavaScript `ruler units` and `type units` are properties of the `Preferences`, accessed through the `application` object's `preferences` property as shown below.

VB:

```
appRef.Preferences.RulerUnits = PsInches
appRef.Preferences.TypeUnits = PsTypePixels
appRef.Preferences.PointSize = PsPostScriptPoints
```

JS:

```
preferences.rulerUnits = Units.INCHES;
preferences.typeUnits = TypeUnits.PIXELS;
preferences.pointSize = PointType.POSTSCRIPT;
```

IMPORTANT: Remember to reset the unit settings back to the original values at the end of a script.

3.6 Executing JavaScripts from AS or VB

With Photoshop's scripting support you can run JavaScripts from AppleScript or Visual Basic. For Applescript, use `do javascript`.

For Visual Basic, use either the Application's DoJavaScript or DoJavaScriptFile method. DoJavaScript takes a string, which is the JavaScript code to execute. DoJavaScriptFile opens a file that contains the JavaScript code. An example is below:

AS:

```
set scriptFile to "myscript" as alias
do javascript scriptFile
```

VB:

```
Dim appRef As Photoshop.Application
Set appRef = CreateObject("Photoshop.Application")
appRef.DoJavaScriptFile ("D:\\Scripts\\MosaicTiles.js")
```

3.6.1 Passing arguments to JavaScript

You can pass arguments to JavaScript from either AppleScript or Visual Basic by using the with arguments (Arguments) parameter. The parameter takes an array for you to pass any values.

For example, save the following JavaScript in a file somewhere on your machine:

```
alert( "You passed " + arguments.length + " arguments" );
for ( i = 0; i < arguments.length; ++i )
{
    alert( arguments[i].toString() )
}
```

To pass arguments from AppleScript try this:

```
tell application "Adobe Photoshop 7.0"
    make new document
    do javascript (alias <a path to the JavaScript shown above>) ↵
        with arguments {1, "test text", (file <a path to a file>), ↵
            current document}
end tell
```

To do the same thing in VB, write:

```
Dim appRef As Photoshop.Application
Set appRef = CreateObject("Photoshop.Application")
appRef.DoJavaScriptFile "C:\scripts-temp\test.js", _
    Array(1, "text text", appRef.ActiveDocument)
```

When running JavaScript from AppleScript or Visual Basic you can also control the debugging state. To do this you use the show debugger (ExecutionMode) argument. The values of this argument include:

- `never` (`NeverShowDebugger`): This option will disable debugging from the JavaScript. Any error that occurs in the JavaScript will result in a JavaScript exception being thrown. Note that you can catch JavaScript exceptions in your script; see 2.10.3, “[JavaScript Debugging](#)” on page 26 for more information on how to handle JavaScript exceptions. When you use this option the JavaScript command “`debugger();`” will be ignored.
- `on runtime error` (`DebuggerOnError`): This option will automatically stop the execution of your JavaScript when a runtime error occurs and show the JavaScript debugger. When you use this option the JavaScript command “`debugger();`” will stop the JavaScript and display the JavaScript debugger.
- `before running` (`BeforeRunning`): This option will show the JavaScript debugger at the beginning of your JavaScript. When you use this option the JavaScript command “`debugger();`” will stop the JavaScript and display the JavaScript debugger.

3.6.2 Executing one-line JavaScripts

You can also execute simple JavaScripts directly without passing a file as shown in the following examples.

AS:

```
do javascript "alert('alert text');"
```

VB:

```
objApp.DoJavaScript ("alert('alert text');")
```

3.7 The Application object

AppleScript and Visual Basic scripts can target multiple applications so the first thing you should do in your script is target Photoshop.

By using the properties and commands of the Application object, you can work with global Photoshop settings, open documents, execute actions, and exercise other Photoshop functionality.

Targeting the Application object

To target the Photoshop application in AppleScript, you must use a `tell...end tell` block. By enclosing your Photoshop commands in the following statement, AppleScript will understand you are targeting Photoshop.

```
tell application "Adobe Photoshop 7.0"
```

```
...
```

```
end tell
```

In Visual Basic or VBScript, you create and use a reference to the Application. Typically, you would write:

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

If using VB, this can also be done by writing:

```
Set appRef = New Photoshop.Application
```

In JavaScript, there is no application object and therefore, all properties and methods of the application are accessible without any qualification. To get the active Photoshop document, write:

```
var docRef = activeDocument;
```

Once you have targeted your application, you are ready to work with the properties and commands of the application object.

The active document

Because “document 1” does not always indicate the front-most document, it’s recommended that your scripts set the current or active document before executing any other commands. To do this, use the “current document (ActiveDocument/activeDocument)” property on the application object.

```
AS: set docRef to current document
```

```
VB: Set docRef = appRef.ActiveDocument
```

```
JS: docRef = activeDocument;
```

You can also switch back and forth between documents by setting the active document.

```
AS: set current document to document "My Document"
```

```
VB: appRef.ActiveDocument = appRef.Documents("My Document")
```

```
JS: activeDocument = documents["My Document"];
```

Application preferences

The application object contains a property for Photoshop preferences. The preferences property is itself an object and has many properties. The name of the preferences object for the three languages is the following:

```
AS: settings
```

```
VB: Preferences
```

```
JS: preferences
```

The properties in the preferences object correlate to the preferences found by displaying the Photoshop “Preferences” dialog in the user interface (select the “Edit > Preferences” menu in the Photoshop UI).

Display dialogs

It is important to be able to control dialogs properly from a script. If a dialog is shown your script stops until a user dismisses the dialog. This is normally fine in an interactive script that expects a user to be sitting at the machine. But if you have a script that runs in an unsupervised (batch) mode you do not want dialogs to be displayed and stop your script.

Using the `display dialogs` (`DisplayDialogs/displayDialogs`) property on the application object you can control whether or not dialogs are displayed.

If you set `display dialogs` to `always` (`psDisplayAllDialogs/ALL`), Photoshop will show all user related dialogs. This is typically not what you want.

If you set `display dialogs` to `error dialogs` (`DisplayErrorDialogs/ERROR`), then only dialogs related to errors are shown. You would typically use this setting when you are developing a script or if your script is an interactive one that expects a user to be sitting at the machine while running the script.

If you set `display dialogs` to `never` (`DisplayNoDialogs/NO`), then no dialogs are shown. If an error occurs it will be returned as an error to the script. See section 2.10.4, “Error handling” on page 31 for more information on catching errors.

Opening a document

When using the open command there are a number of specifiable options. These options are grouped by file type in the provided open options classes. Because the type and contents of the file you are working on affects how it is opened, some of the option values may not always be applicable. It also means that many of the option values do not have well defined default values.

The best way to determine what values can or should be used for open is to perform an open command from the user interface. You can then copy the value from the options dialog to your script. You should perform a complete open operation because there can be multiple dialogs presented before the document is actually opened. If you cancel one of the open dialogs without completing the operation you could miss seeing a dialog which contains values needed in your script.

Specifying file formats to open

Because Photoshop supports many different file formats, the Open command lets you specify the format of the document you are opening. If you do not specify the format, Photoshop will infer the type of file for you. Here’s how to open a document using its default type:

AS:

```
set theFile to alias "MyFile.psd"  
open theFile
```

VB:

```
fileName = "C:\MyFile.psd"  
Set docRef = appRef.Open(fileName)
```

JS:

```
var fileRef = new File("//MyFile.psd");  
var docRef = open (fileRef);
```

Notice that in JavaScript, you must create a File object, and it gets passed into the open command. See the JavaScript file documentation for more information.

Some formats require extra information when opening. When you open a Generic EPS, Generic PDF, Photo CD or Raw image you have to provide additional information to the open command.

Do this by using the various open options classes:

- EPS Open Options (EPSOpenOptions/EPSOpenOptions)
- PDF Open Options (PDFOpenOptions/PDFOpenOptions)
- Photo CD Open Options (PhotoCDOpenOptions/PhotoCDOpenOptions)
- raw format Options (RawFormatOpenOptions/RawFormatOpenOptions)

The following example shows how to open a generic PDF document.

AS:

```
tell application "Adobe Photoshop 7.0"  
    set myFilePath to alias < a file path >  
    open myFilePath as PDF with options ↵  
        {class:PDF open options, height:pixels 100, ↵  
          width:pixels 200, mode:RGB, resolution:72, ↵  
          use antialias:true, page:1, ↵  
          constrain proportions:false}  
end tell
```

VB:

```
Dim appRef As Photoshop.Application  
Set appRef = CreateObject("Photoshop.Application")
```

'Remember unit settings; and set to values expected by this script

```
Dim originalRulerUnits As Photoshop.PsUnits  
originalRulerUnits = appRef.Preferences.RulerUnits  
appRef.Preferences.RulerUnits = psPixels
```

'Create a PDF option object

```
Dim pdfOpenOptionsRef As Photoshop.PDFOpenOptions  
Set pdfOpenOptionsRef = CreateObject("Photoshop.PDFOpenOptions")  
pdfOpenOptionsRef.AntiAlias = True  
pdfOpenOptionsRef.Height = 100
```

```
pdfOpenOptionsRef.Width = 200
pdfOpenOptionsRef.mode = psOpenRGB
pdfOpenOptionsRef.Resolution = 72
pdfOpenOptionsRef.ConstrainProportions = False

'Now open the file
Dim docRef As Photoshop.Document
Set docRef = appRef.Open(< a file path>, pdfOpenOptionsRef)

'Restore unit setting
appRef.Preferences.RulerUnits = originalRulerUnits
```

JS:

```
// Set the ruler units to pixels
var originalRulerUnits = preferences.rulerUnits;
preferences.rulerUnits = Units.PIXELS;

// Get a reference to the file that we want to open
var fileRef = new File( < a file path > );

// Create a PDF option object
var pdfOpenOptions = new PDFOpenOptions;
pdfOpenOptions.antiAlias = true;
pdfOpenOptions.height = 100;
pdfOpenOptions.width = 200;
pdfOpenOptions.mode = OpenDocumentMode.RGB;
pdfOpenOptions.resolution = 72;
pdfOpenOptions.constrainProportions = false;

// Now open the file
open( fileRef, pdfOpenOptions );

// restore unit settings
preferences.rulerUnits = originalRulerUnits;
```

Because Photoshop cannot save all of the format types that it can open, the open document types may be different from the save document types.

3.8 Document object

After you target the Photoshop application, the next object you will likely target is the Document object. The Document object can represent any open document in Photoshop.

For example, you could use the `Document` object to get the active layer, save the current document, then copy and paste within the active document or between different documents.

3.8.1 Saving documents and save options

Photoshop scripting support lets you work with various file formats. It is important to note, however, that the `Open` and `Save` formats are not identical.

Also note that some formats available in scripting require you to install optional file formats. The optional formats are:

- Alias PIX
- Electric Image
- SGI RGB
- Wavefront RLA
- SoftImage

When using the `save` command there are a number of specifiable options. These options are grouped by file type in the provided `save options` classes. Because the type and contents of the file you are working on affects how it is saved, some of the option values may not always be applicable. It also means that many of the option values do not have well defined default values.

The best way to determine what values can or should be used for `save` is to perform a `save` command from the user interface and then copy the value from the options dialog to your script. You should perform a complete `save` operation because there can be multiple dialogs presented before the document is saved. If you cancel one of the `save` dialogs without completing the operation you could miss a dialog containing values needed in your script.

There are many objects that allow you to specify how you want to save your document. For example, to save a file as a JPEG file, you would use the `JPEG save options` (`JPEGSaveOptions/JPEGSaveOptions`) class as shown below.

AS:

```
tell application "Adobe Photoshop 7.0"
  make new document
  set myOptions to {class:JPEG save options, -
    embed color profile:false, format options: standard, -
    matte: background color matte,}
  save current document in file myFile as JPEG with options -
    myOptions appending no extension without copying
end tell
```

VB:

```
Dim appRef As New Photoshop.Application
Set jpgSaveOptions = CreateObject("Photoshop.JPEGSaveOptions")
jpgSaveOptions.EmbedColorProfile = True
jpgSaveOptions.FormatOptions = psStandardBaseline
jpgSaveOptions.Matte = psNoMatte
jpgSaveOptions.Quality = 1
appRef.ActiveDocument.SaveAs "c:\temp\myFile2", _
    Options:=jpgSaveOptions, _
    asCopy:=True, extensionType:=psLowercase
```

JS:

```
jpgFile = new File( "/Temp001.jpeg" );
jpgSaveOptions = new JPEGSaveOptions();
jpgSaveOptions.embedColorProfile = true;
jpgSaveOptions.formatOptions = FormatOptions.STANDARDBASELINE;
jpgSaveOptions.matte = MatteType.NONE;
jpgSaveOptions.quality = 1;
activeDocument.saveAs(jpgFile, jpgSaveOptions, true,
    Extension.LOWERCASE);
```

3.8.2 Document information

A Photoshop document can be associated with additional information such as the author via the “File > File Info” menu.

The information found in this menu-item is handled by the `info` (`DocumentInfo`) object. To change the information, reference to the `info` object and set its properties as shown below.

AS:

```
set docInfoRef to info of current document
set copyrighted of docInfoRef to copyrighted work
set owner url of docInfoRef to "http://www.adobe.com"
```

VB:

```
Set docInfoRef = docRef.Info
docInfoRef.Copyrighted = psCopyrightedWork
docInfoRef.OwnerUrl = "http://www.adobe.com"
```

JS:

```
docInfoRef = docRef.info;
docInfoRef.copyrighted = CopyrightedType.COPYRIGHTEDWORK;
docInfoRef.ownerUrl = "http://www.adobe.com";
```

3.8.3 Document manipulation

The `Document` object is used to make modifications to the document image. By using the `Document` object you can crop, rotate or flip the canvas, resize the image or canvas, and trim the `Image`.

Because unit values are passed in when resizing an image, it is recommended that you first set your ruler units prior to resizing. See section 3.5.3, “Changing ruler and type units” on page 48 for more information.

The examples in this section assume that the ruler units have been set to inches.

To resize the image so that it is four inches wide by four inches high, use the document's `resize` (`Resize/resize`) command.

```
AS: resize image current document width 4 height 4
```

```
VB: docRef.ResizeImage 4,4
```

```
JS: docRef.resizeImage( 4,4 );
```

Resizing the canvas is done similarly.

```
AS: resize canvas current document width 4 height 4
```

```
VB: docRef.ResizeCanvas 4,4
```

```
JS: docRef.resizeCanvas( 4,4 );
```

To trim the excess space from a document, use the `trim` (`Trim/trim`) command. The example below will trim the top and bottom of the document.

```
AS:
```

```
trim current document basing trim on top left pixel ↵
    with top trim and bottom trim without left trim and right trim
```

```
VB:
```

```
docRef.Trim Type:=psTopLeftPixel, Top:=True, Left:=False, _
    Bottom:=True, Right:=False
```

```
JS:
```

```
docRef.trim(TrimType.TOPLEFT, true, false, true, false);
```

NOTE: The crop command uses unit values. The examples below assume that the ruler unit is set to pixels.

```
AS:
```

```
crop current document bounds {10, 20, 40, 50} angle 45 ↵
    resolution 72 width 20 height 20
```

```
VB:
```

```
docRef.Crop Array(10,20,40,50), Angle:=45, Width:=20, _
    Height:=20, Resolution:=72
```

JS:

```
docRef.crop (new Array(10,20,40,50), 45, 20, 20, 72);
```

To flip the canvas horizontally:

AS: flip canvas current document direction horizontal

VB: docRef.FlipCanvas psHorizontal

JS: docRef.flipCanvas(Direction.HORIZONTAL);

3.9 Layer objects

Photoshop has 2 types of layers: an art layer that can contain image contents and a layer set that can contain zero or more art layers. Scripts, likewise, have two types of layers: art layer and layer set.

Both types of layers have common properties such as “visible.” The common attributes are placed in a general “layer” class that both “art layer” and “layer set” inherits from.

When you create a layer you must specify whether you are creating an art layer or a layer set. The following examples show how to create an art layer filled with red at the beginning of the current document

AS:

```
tell application "Adobe Photoshop 7.0"
  make new art layer at beginning of current document -
    with properties {name:"MyBlendLayer", blend mode:normal}
  select all current document
  fill selection of current document with contents -
    {class:RGB color, red:255, green:0, blue:0}
end tell
```

VB:

```
Dim appRef As Photoshop.Application
Set appRef = CreateObject("Photoshop.Application")

' Create a new art layer at the beginning of the current document
Dim docRef As Photoshop.Document
Dim layerObj As Photoshop.ArtLayer
Set docRef = appRef.ActiveDocument
Set layerObj = appRef.ActiveDocument.ArtLayers.Add
layerObj.Name = "MyBlendLayer"
layerObj.BlendMode = psNormalBlend

' Select all so we can apply a fill to the selection
appRef.ActiveDocument.Selection.SelectAll

' Create a color to be used with the fill command
Dim colorObj As Photoshop.SolidColor
Set colorObj = CreateObject("Photoshop.SolidColor")
colorObj.RGB.Red = 255
colorObj.RGB.Green = 100
colorObj.RGB.Blue = 0

' Now apply fill to the current selection
appRef.ActiveDocument.Selection.Fill colorObj
```

JS:

```
// Create a new art layer at the beginning of the current document
var layerRef = activeDocument.artLayers.add();
layerRef.name = "MyBlendLayer";
layerRef.blendMode = BlendMode.NORMAL;

// Select all so we can apply a fill to the selection
activeDocument.selection.selectAll();

// Create a color to be used with the fill command
var colorRef = new SolidColor();
colorRef.rgb.red = 255;
colorRef.rgb.green = 100;
colorRef.rgb.blue = 0;

// Now apply fill to the current selection
activeDocument.selection.fill(colorRef);
```

The following examples show how to create a layer set after the first layer in the current document:

AS:

```
tell application "Adobe Photoshop 7.0"  
    make new layer set after layer 1 of current document  
end tell
```

VB:

```
Dim appRef As Photoshop.Application  
Set appRef = CreateObject("Photoshop.Application")  
  
' Get a reference to the first layer in the document  
Dim layerRef As Photoshop.Layer  
Set layerRef = appRef.ActiveDocument.Layers(1)  
  
' Create a new LayerSet (it will be created at the beginning of the  
' document)  
Dim newLayerSetRef As Photoshop.LayerSet  
Set newLayerSetRef = appRef.ActiveDocument.LayerSets.Add  
  
' Move the new layer to after the first layer  
newLayerSetRef.MoveAfter layerRef
```

JS:

```
// Get a reference to the first layer in the document  
var layerRef = activeDocument.layers[0];  
  
// Create a new LayerSet (it will be created at the beginning of the  
// document)  
var newLayerSetRef = activeDocument.layerSets.add();  
  
// Move the new layer to after the first layer  
newLayerSetRef.moveAfter( layerRef );
```

An existing art layer can also be changed to a text layer if the existing layer is empty. Conversely you can change a text layer to a normal layer. When you do this the text in the layer is rasterized.

3.9.1 Setting the Active layer

Before attempting to manipulate a layer you must first select it. You can do this by setting the current layer (ActiveLayer/activeLayer) to the one you want to manipulate.

AS:

```
set current layer of current document to layer "Layer 1" of ↵  
current document
```

VB:

```
docRef.ActiveLayer = docRef.Layers("Layer 1")
```

JS:

```
docRef.activeLayer = docRef.layers["Layer 1"];
```

3.9.2 Layer sets

Existing layers can be moved into layer sets. The following examples show how to create a layer set, duplicate an existing layer, and move the duplicate layer into the layer set.

AS:

```
set current document to document "My Document"  
set layerSetRef to make new layer set at end of current document  
set newLayer to duplicate layer "Layer 1" of current document ↵  
to end of current document  
move newLayer to end of layerSetRef
```

In AppleScript, you can also duplicate a layer directly into the destination layer set.

```
set current document to document "My Document"  
set layerSetRef to make new layer set at end of current document  
duplicate layer "Layer 1" of current document to end of layerSetRef
```

In Visual Basic and JavaScript you'll have to duplicate the layer and then move it. Here's how:

VB:

```
Set layerSetRef = docRef.LayerSets.Add
Set layerRef = docRef.ArtLayers(1).Duplicate
layerRef.MoveToEnd layerSetRef
```

JS:

```
var layerSetRef = docRef.layerSets.add();
var layerRef = docRef.artLayers[0].duplicate();
layerRef.moveToEnd (layerSetRef);
```

3.9.3 Linking layers

Scripting also supports linking and unlinking layers. You may want to link layers together so that moving or transforming them can be done with one statement. To link layers together, do the following:

AS:

```
make new art layer in current document with properties {name:"L1"}
make new art layer in current document with properties {name:"L2"}
link art layer "L1" of current document with art layer "L2" of
current document
```

VB:

```
Set layer1Ref = docRef.ArtLayers.Add()
Set layer2Ref = docRef.ArtLayers.Add()
layer1Ref.Link layer2Ref.Layer
```

JS:

```
var layerRef1 = docRef.artLayers.add();
var layerRef2 = docRef.artLayers.add();
layerRef1.link(layerRef2);
```

3.9.4 Applying styles to layers

Styles can be applied to layers from your scripts. The styles correspond directly to the styles in the Photoshop Styles palette and are referenced by their literal string name. Here is an example of how to set a layer style to the layer named “L1.”

NOTE: The layer styles name is case sensitive.

AS:

```
apply layer style art layer "L1" of current document using -  
    "Puzzle (Image)"
```

VB:

```
docRef.ArtLayers("L1").ApplyStyle "Puzzle (Image)"
```

JS:

```
docRef.artLayers["L1"].applyStyle("Puzzle (Image)");
```

3.9.5 Rotating layers

Use the `rotate` (`Rotate/rotate`) command on the layer to rotate the entire layer. Positive integers rotate the layer clockwise. Negative integers rotate it counterclockwise.

AS:

```
rotate current layer of current document angle 45.0
```

VB:

```
docRef.ActiveLayer.Rotate 45.0
```

JS:

```
docRef.activeLayer.rotate(45.0);
```

3.10 Text item object

In Photoshop, the `Text` object is a property of the art layer. To create a new text layer, you must create a new art layer and then set the art layer's `kind` (`Kind/kind`) property to `text layer` (`psTextLayer/ LayerKind.TEXT`). By changing an art layer's `kind`, you can also convert an existing layer to text as long as the layer is empty. For example, to create a new text layer, write:

AS:

```
make new art layer in current document with properties ↵
    { kind: text layer }
```

VB:

```
set newLayerRef = docRef.ArtLayers.Add()
newLayerRef.Kind = psTextLayer
```

JS:

```
var newLayerRef = docRef.artLayers.add();
newLayerRef.kind = LayerKind.TEXT;
```

To check if an existing layer is a text layer, you must compare the layer's `kind` to `text layer` (`psTextLayer/LayerKind.TEXT`).

AS:

```
if (kind of layerRef is text layer) then
```

VB:

```
If layerRef.Kind = psTextLayer Then
```

JS:

```
if (newLayerRef.kind == LayerKind.TEXT)
```

The art layer class has a `text item` (`TextItem/textItem`) property which is only valid when the art layer's `kind` is `text layer`. You can use this property to make modifications to your `text layer` such as setting its contents, changing its size, and controlling the different effects that can be applied to text. For example, to set the justification of your text to right justification, you write:

AS:

```
set justification of text item of art layer "my text" of ↵
    current document to right
```

VB:

```
docRef.ArtLayers("my text").TextItem.Justification = psRight
```

JS:

```
docRef.artLayers["my text"].textItem.justification =  
Justification.RIGHT;
```

IMPORTANT: *The text item object has a kind property, which can be set to either point text (psPointText/TextType.POINTTEXT) or paragraph text (psParagraphText/TextType.PARAGRAPHTEXT). When a new text item is created, its kind property is automatically set to point text.*

The text item properties height, width and leading are only valid when the text item's kind property is set to paragraph text.

3.10.1 Setting the contents of the text item

To set the contents of a text item in AppleScript you would write:

```
set contents of text item of art layer "Layer 1" of ↵  
current document to "Hello"
```

If you use a text item object reference to set the contents you will need to write:

```
set contents of contents of textItemRef to "Hello"
```

The second “contents of” is needed because “contents” is a keyword which tells AppleScript to operate on the contents of the variable, rather than on the object to which it may refer. This means that AppleScript sees the above line as:

```
set text item of art layer 1 of document "Untitled-1" ↵  
to "Hello"
```

To set the contents using references in VB and JS, write the following:

VB:

```
textLayerRef.TextItem.Contents = "Hello"
```

JS:

```
textLayerRef.textItem.contents = "Hello";
```

3.10.2 Setting text stroke colors

Setting the stroke color in AppleScript is a bit different than setting it in Visual Basic or JavaScript. To set the stroke color in AppleScript, use one of the color classes: CMYK color, gray color, HSB color, Lab color, or RGB color.

To set it in Visual Basic or JavaScript, you must first create a `SolidColor` object and appropriately assign one of the color classes to it. The following examples show how to set the stroke color of a text item-object to a CMYK color.

See section 3.14, “Color objects” on page 77 for more information on working with colors.

AS:

```
set stroke color of textItemRef to {class:CMYK color, cyan:20, magenta:50, yellow:30, black:0}
```

VB:

```
Set newColor = CreateObject ("Photoshop.SolidColor")
newColor.CMYK.Cyan = 20
newColor.CMYK.Magenta = 100
newColor.CMYK.Yellow = 30
newColor.CMYK.Black = 0
textLayerRef.TextItem.Color = newColor
```

JS:

```
var newColor = new SolidColor();
newColor.cmyk.cyan = 20;
newColor.cmyk.magenta = 100;
newColor.cmyk.yellow = 30;
newColor.cmyk.black = 0;
textLayerRef.color = newColor;
```

3.10.3 Setting fonts

To set the font of your text item object, set the text item's `font` property. The font names that you can use are the PostScript® names for the fonts. The PostScript names are not the names that are displayed in Photoshop's character palette. The steps below show how to find a PostScript font name.

1. Using the Photoshop user interface, create a new Photoshop document.
2. Create a new text layer and add some text to it.
3. Select the text you created in step 2.
4. Select the desired font from the Font pull down menu (for example, "Arial")

5. Create a script to get the font name of the text. An example JavaScript is below:

```
var textLayer = activeDocument.artLayers[0];
if (textLayer.kind == LayerKind.TEXT)
{
    alert(textLayer.textItem.font);
}
```

6. The name that is displayed in the alert dialog is the PostScript name of the font. Use this name to set the font of your text. For example, the above script returned the name "ArialMT." The examples below show how to set this font:

AS: set font of textItemRef to "ArialMT"

VB: textLayer.TextItem.Font = "ArialMT"

JS: textLayer.textItem.font = "ArialMT";

3.10.4 Warping text

Warping is another common effect that can be applied to text. To warp a text item-object, set the object's `warp style` (`WarpStyle/warpStyle`) property. The style to set it to is an enumeration.

AS:

set warp style of textItemRef to flag

VB:

textLayerRef.TextItem.WarpStyle = psFlag

JS:

textLayerRef.textItem.warpStyle = WarpStyle.FLAG;

3.11 Selections

There are instances where you will want to write scripts that only act on the current selection. If you are writing a script that depends on a selection, be sure to set the selection yourself, as you cannot test for a non-existent selection. When creating new selections, you can add to, replace, or subtract from a selection.

For example, you may apply effects to a selection or copy the current selection to the clipboard. But remember that you may have to set the active layer before acting on the selection. Here's how:

AS:

```
set current layer of current document to layer "Layer 1" of ↵
current document
```

VB:

```
docRef.ActiveLayer = docRef.Layers("Layer 1")
```

JS:

```
docRef.activeLayer = docRef.layers["Layer 1"];
```

See section 3.9.1, “[Setting the Active layer](#)” on page 61 for more information.

3.11.1 Defining selections

To create a new selection, use the `select` method with a type of `replaced` (`psReplaceSelection/SelectionType.REPLACED`). The other selection types are `diminished`, `extended` and `intersected`.

The `diminished` type will shrink the current selection, the `extended` selection type will grow the current selection, and the `intersected` type will find the intersection of the current selection and the new selection and replace the current selection with the intersection of the two.

If there is no intersection between the selections, the new selection will be empty. If there is no current selection, the new selection will be the newly specified selection.

Here are examples of how to replace the current selection:

AS:

```
select current document region {{ 5, 5}, {5, 100}, ↵
{ 80, 100}, { 80, 5}} combination type replaced
```

VB:

```
Dim appRef As New Photoshop.Application

'remember unit settings; and set to values expected by this script
Dim originalRulerUnits As Photoshop.PsUnits
originalRulerUnits = appRef.Preferences.RulerUnits
appRef.Preferences.RulerUnits = psPixels

'get selection and replace it
Dim docRef As Photoshop.Document
Set docRef = appRef.ActiveDocument
docRef.Selection.Select Array(Array(50, 60), Array(150, 60), _
    Array(150, 120), Array(50, 120)), Type:=psReplaceSelection

'restore unit setting
appRef.Preferences.RulerUnits = originalRulerUnits
```

JS:

```
// remember unit settings; and set to values expected by this
// script
var originalRulerUnits = preferences.rulerUnits;
preferences.rulerUnits = Units.PIXELS;

//get selection and replace it;
activeDocument.selection.select (new Array(new Array(60, 10), new
    Array(100, 10), new Array(100, 100), new Array(60, 100)),
    SelectionType.REPLACE);

// restore unit setting
preferences.rulerUnits = originalRulerUnits;
```

3.11.2 Stroking the selection border

The following examples show how to stroke the boundaries around the current selection and set the stroke color and width.

AS:

```
stroke selection of current document using color ~
    {class:CMYK color,cyan:20, magenta:50, yellow:30, black:0}~
width 5 location inside blend mode vivid light opacity 75 ~
without preserving transparency
```

VB:

```
selRef.Stroke strokeColor, Width:=5, Location:=psInsideStroke, _  
    mode:=psVividLightBlend, Opacity:=75, _  
    PreserveTransparency:=False
```

JS:

```
activeDocument.selection.stroke (strokeColor, 2,  
    StrokeLocation.OUTSIDE, ColorBlendMode.VIVIDLIGHT, 75,  
    false);
```

IMPORTANT: *The transparency parameter cannot be used for background layers.*

3.11.3 Inverting selections

When you invert a selection, you are masking the selection so you can work on the rest of the document, layer or channel while protecting the selection. Here's how to invert the current selection:

AS: invert selection of current document

VB: selRef.Invert

JS: selRef.invert();

3.11.4 Expand, contract and feather selections

These three commands are used to change the size of the selection. The values are passed in ruler units, the value of which are stored in Photoshop preferences and can be changed by your scripts. Feathering a selection will smooth its corners by the specified number of units while expand and contract will grow and shrink the selection.

If your ruler units are set to pixels, then the following examples will expand, contract and feather by five pixels. See section [3.5.3, "Changing ruler and type units"](#) on page 48 for examples of how to change ruler units.

AS:

```
expand selection of current document by pixels 5  
contract selection of current document by pixels 5  
feather selection of current document by pixels 5
```

VB:

```
Dim appRef As Photoshop.Application  
Set appRef = CreateObject("Photoshop.Application")
```

```
'remember unit settings; and set to pixels
```

```
Dim originalRulerUnits As Photoshop.PsUnits
originalRulerUnits = appRef.Preferences.RulerUnits
appRef.Preferences.RulerUnits = psPixels

Dim selRef As Photoshop.Selection
Set selRef = appRef.ActiveDocument.Selection

selRef.Expand 5
selRef.Contract 5
selRef.Feather 5

'Rem restore unit setting
appRef.Preferences.RulerUnits = originalRulerUnits
```

JS:

```
// remember unit settings; and set to pixels
var originalRulerUnits = preferences.rulerUnits;
preferences.rulerUnits = Units.PIXELS;

var selRef = activeDocument.selection
selRef.expand( 5 );
selRef.contract( 5 );
selRef.feather( 5 );

// restore unit setting
preferences.rulerUnits = originalRulerUnits;
```

3.11.5 Filling a selection

You can fill a selection either with a color or a history state. To fill with a color:

AS:

```
fill selection of current document with contents -
    {class: RGB color, red:255, green:0, blue:0} blend mode -
    vivid light opacity 25 without preserving transparency
```

VB:

```
Set fillColor = CreateObject("Photoshop.SolidColor")
fillColor.RGB.Red = 255
fillColor.RGB.Green = 0
fillColor.RGB.Blue = 0
selRef.Fill fillColor mode:=psVividLightBlend, _
    Opacity:=25, PreserveTransparency:=False
```

JS:

```
var fillColor = new SolidColor();
fillColor.rgb.red = 255;
fillColor.rgb.green = 0;
fillColor.rgb.blue = 0;
activeDocument.selection.fill( fillColor, ColorBlendMode.VIVIDLIGHT,
    25, false);
```

To fill the current selection with the 10th item in the history state, you would write:

AS:

```
fill selection of current document with contents history state 10 ↵
of current document
```

VB:

```
selRef.Fill docRef.HistoryStates(10)
```

JS:

```
selRef.fill( activeDocument.historyStates[9]);
```

3.11.6 Rotating selections

You can rotate either the contents of a selection or the selection boundary itself. The first set of examples below show how to rotate an existing selection 45 degrees.

AS:

```
rotate selection of current document angle 45
```

VB:

```
docRef.Selection.Rotate(45);
```

JS:

```
active.Document.selection.rotate(45);
```

To rotate the boundary of an existing selection:

AS:

```
rotate boundary selection of current document angle 45
```

VB:

```
docRef.Selection.RotateBoundary(45);
```

JS:

```
activeDocument.selection.rotateBoundary(45);
```

3.11.7 Loading and storing selections

Scripting Support exposes the functionality to store and load selections. Selections get stored into channels and loaded from channels. The following examples will store the current selection into a channel named “My Channel” and extend the selection with any selection that is currently in that channel.

AS:

```
store selection of current document into channel "My Channel" of -  
current document combination type extended
```

VB:

```
selRef.Store docRef.Channels("My Channel"), psExtendSelection
```

JS:

```
selRef.store(docRef.channels["My Channel"], SelectionType.EXTEND);
```

To restore a selection that has been saved to a selection, use the load (Load/load) method as shown below.

AS:

```
load selection of current document from channel "My Channel" of -  
current document combination type extended
```

VB:

```
selRef.Load docRef.Channels("My Channel"), psExtendSelection
```

JS:

```
selRef.load (docRef.channels["My Channel"], SelectionType.EXTEND);
```

See section [3.16, “Clipboard interaction” on page 81](#) for examples on how to copy, cut and paste selections.

3.12 Working with Filters

To apply a filter, use the layer's `filter` command for AppleScript or the `ApplyXXX/applyXXX` methods for Visual Basic and JavaScript. The following examples apply the Gaussian blur filter to the active layer.

AS:

```
filter current layer of current document using Gaussian blur -  
    with options { radius: 5 }
```

VB:

```
docRef.ActiveLayer.ApplyGaussianBlur 5
```

JS:

```
docRef.activeLayer.applyGaussianBlur(5);
```

3.12.1 Selecting channel(s) to filter

When applying filters, keep in mind they affect the selected channels of a visible layer. This means that prior to running a filter, you may have to set the active channels. Since more than one channel can be active at a time, you must provide an array of channels when setting a channel. The code below demonstrates how to set the active channels to the channels named “Red” and “Blue.”

AS:

```
set current channels of current document to { channel "Red" of -  
    current document, channel "Blue" of current document }
```

VB:

```
Dim theChannels As Variant  
theChannels = Array(docRef.Channels("Red"), docRef.Channels("Blue"))  
docRef.ActiveChannels = theChannels
```

JS:

```
theChannels = new Array(docRef.channels["Red"],  
docRef.channels["Blue"]);  
docRef.activeChannels = theChannels;
```

Or you can easily select all component channels by using the “component channel” property on the document:

AS:

```
set current channels of current document to component channels -  
    of current document
```

VB:

```
appRef.ActiveDocument.ActiveChannels= _  
    appRef.ActiveDocument.ComponentChannels
```

JS:

```
activeDocument.activeChannels = activeDocument.componentChannels;
```

3.12.2 Other filters

If the filter type that you want to use on your layer is not part of the scripting interface, you can also use the Action Manager from a JavaScript to run a filter. If you are using AppleScript, Visual Basic or VBScript, you can still run a JavaScript from your script.

See section 3.17, “Action Manager scripting” on page 84 for more information.

3.13 Channel object

The Channel object (Channel/channel) gives you access to much of the available functionality on Photoshop channels. You can create, delete and duplicate channels or retrieve a channel's histogram and change its kind or change the current channel selection.

3.13.1 Channel types

In addition to the component channels, Photoshop lets you to create additional channels. You can create a “spot color channel”, a “masked area channel” and a “selected area channel.”

It's important to keep the different types of channels in mind when writing scripts that work on them.

If you have an RGB document you automatically get a red, blue and a green channel. These kinds of channels are related to the document mode and are called “component channels.”

In the Scripting model a Channel has a kind property you can use to get and set the type of the channel. Possible values are: component channel, masked area channel, selected area channel and spot color channel.

You cannot change the kind of a component channel. But you could change a “masked area channel” to be “selected area channel” by saying:

```
AS: set kind of myChannel to selected area channel
```

```
VB: channelRef.kind = psSelectedAreaAlphaChannel
```

```
JS: channelRef.kind = ChannelType.SELECTEDAREA;
```

3.13.2 Setting the active channel

Because more than one channel can be active at a time, when setting a channel, you must provide a channel array. The sample below demonstrates how to set the active channels to the first and third channel.

AS:

```
set current channels of current document to ↵
    { channel 1 of current document, channel 3 of current document }
```

VB:

```
Dim theChannels As Variant
theChannels = Array(docRef.Channels(1), docRef.Channels(3))
docRef.ActiveChannels = theChannels
```

JS:

```
theChannels = new Array(docRef.channels[0], docRef.channels[2]);
docRef.activeChannels = theChannels;
```

Deleting a component will change the document to a multi-channel document.

3.13.3 Creating new channels

You can create three different types of channels from a script. These types are:

- masked area channel (psMaskedAreaAlphaChannel, ChannelType.MASKEDAREA)
- selected area channel (psSelectedAreaAlphaChannel, ChannelType.SELECTEDAREA)
- spot color channel (psSpotColorChannel, ChannelType.SPOTCOLOR).

The examples below show how to create a new masked area channel.

AS:

```
make new channel in current document with properties ↵
    {kind:masked area channel }
```

VB:

```
Set channelRef = docRef.Channels.Add
channelRef.Kind = psMaskedAreaChannel
```

JS:

```
var channelRef = docRef.channels.add();
channelRef.kind = ChannelType.MASKEDAREA;
```

3.14 Color objects

From scripting you can use the same range of colors that are available from the Photoshop user interface. Each has its own set of properties, which are specific to the color. For example, the RGB color class contains three properties — red, blue and green.

3.14.1 Setting a Color

Here's how to set the foreground color to a CMYK color in AppleScript.

```
set foreground color to {class:CMYK color, cyan:20.0, ↵
    magenta:90.0, yellow:50.0, black:50.0}
```

Because you can use any color model, you could also write the following to set the foreground to an RGB color.

```
set foreground color to { class:RGB color, red:80.0, green:120.0, ↵
    blue:57.0 }
```

In Visual Basic and JavaScript, the `SolidColor` object handles all colors. To set the foreground color you should create a `SolidColor` object, set its color model by assigning the color model values and then set the foreground color to the solid color. Here's how:

VB:

```
solidColor = CreateObject("Photoshop.SolidColor")
appRef.ForegroundColor = solidColor
```

JS:

```
var solidColor = new SolidColor();
foregroundColor = solidColor;
```

SolidColor class

Visual Basic and JavaScript have an additional class called the `SolidColor` class. This class contains a property for each color model. To use this object, first create an instance of a `SolidColor` object, then set its appropriate color model properties. Once a color model has been assigned to a `SolidColor` object, the `SolidColor` object cannot be reassigned to a different color model. Below are examples on how to create a `SolidColor` object and set its CMYK property.

VB:

```
Dim solidColor As Photoshop.SolidColor
Set solidColor = CreateObject("Photoshop.SolidColor")
solidColor.CMYK.Cyan = 20
solidColor.CMYK.Magenta = 90
solidColor.CMYK.Yellow = 50
```

```
solidColor.CMYK.Black = 50
```

JS:

```
var solidColor = new SolidColor();
solidColor.cmyk.cyan = 20;
solidColor.cmyk.magenta = 90;
solidColor.cmyk.yellow = 50;
solidColor.cmyk.black = 50;
```

Hex values

An RGB color can also be represented as a hex value. The hexadecimal value is used to represent the three colors of the RGB model. The hexadecimal value contains three pairs of numbers which when read from left to right, represent the red, blue and green colors.

In AppleScript, the hex value is represented by the `hex` value string property in class `RGB hex color`, and you use the `convert color` command described below to retrieve the hex value.

In Visual Basic and JavaScript, the `RGBColor` object has a string property called `HexValue/hexValue`.

3.14.2 Getting and converting colors

Here's how to get the foreground color in AppleScript.

```
get foreground color
```

This may return an RGB color and in some cases you may want the CMYK equivalent. To convert an RGB color to CMYK in AppleScript you would write:

```
convert color foreground color to CMYK
```

VB/JS:

The foreground color returns a `SolidColor` object. You should use its `model` property to determine the color model.

```
If (someColor.model = ColorModel.RGB) Then
    alert("It's an RGB color")
End If
```

You can also ask the `SolidColor` object to convert its color to any of the supported models. For example, writing:

```
someColor.cmyk
```

will return a `CMYKColor` object representing the CMYK version of the color in `someColor` regardless of the color model of `someColor`.

The examples below show how to convert the foreground color to a Lab color.

AS:

```
-- Convert foreground application color to Lab
set myLabColor to convert color foreground color to Lab
```

VB:

```
' Get the foreground color as Lab
Dim myLabColor As Photoshop.LabColor
Set myLabColor = appRef.ForegroundColor.Lab
```

JS:

```
// Get the Lab color from the foreground color.
var myLabColor = foregroundColor.lab;
```

3.14.3 Comparing Colors

Using the `equal colors` (`IsEqual/isEqual`) commands, you can easily compare colors. These methods will return `true` if the colors are visually equal to each other and `false` otherwise. The examples below compare the foreground color to the background color.

AS: `if equal colors foreground color with background color then`

VB: `If (appRef.ForegroundColor.IsEqual(appRef.BackgroundColor)) Then`

JS: `if (foregroundColor.isEqual(backgroundColor))`

3.14.4 Getting a Web Safe Color

To convert a color to a web safe color use the `web safe color` command on AppleScript and the `NearestWebColor/nearestWebColor` property on the `SolidColor` object for Visual Basic and JavaScript. The web safe color returned is an RGB color.

AS:

```
set myWebSafeColor to web safe color for foreground color
```

VB:

```
Dim myWebSafeColor As Photoshop.RGBColor
Set myWebSafeColor = appRef.ForegroundColor.NearestWebColor
```

JS:

```
var webSafeColor = new RGBColor();
webSafeColor = foregroundColor.nearestWebColor;
```

3.15 History object

Photoshop keeps a history of the actions that affect the appearance of documents. Each entry in the Photoshop History palette is considered a “History State.” These states are accessible from document object and can be used to reset the document to a previous state. A history state can also be used to fill a selection.

To set your document back to a particular state, set the document's current history state:

AS:

```
set current history state of current document to history state 1 ↵  
of current document
```

VB:

```
docRef.ActiveHistoryState = docRef.HistoryStates(1)
```

JS:

```
docRef.activeHistoryState = docRef.historyStates[0];
```

The code above sets the current history state to the top history state that is in the History palette. Using history states in this fashion give you the ability to undo the actions that were taken to modify the document.

The example below saves the current state, applies a filter, and then reverts back to the saved history state.

AS:

```
set savedState to current history state of current document  
filter current document using motion blur with options ↵  
{angle:20, radius: 20}  
set current history state of current document to savedState
```

VB:

```
Set savedState = docRef.ActiveHistoryState  
docRef.ApplyMotionBlur 20, 20  
docRef.ActiveHistoryState = savedState
```

JS:

```
savedState = docRef.activeHistoryState;  
docRef.applyMotionBlur( 20, 20 );  
docRef.activeHistoryState = savedState;
```

IMPORTANT: *Reverting back to a previous history state does not remove any latter states from the history collection. Use the Purge command to remove latter states from the history collection as shown below:*

```
AS:  purge history caches
```

```
VB:  appRef.Purge( psHistoryCaches)
```

```
JS:  purge( PurgeTarget.HISTORYCACHES );
```

3.15.1 Filling a selection with a history state

A history state can also be used to fill a selection. See section 3.11, “Selections” on page 67 for more information on working with selections.

3.16 Clipboard interaction

The clipboard commands in Photoshop Scripting Support operate on layers and selections. The commands can be used to operate on a single document, or to move information between documents.

NOTE: On Mac OS, Photoshop must be the front-most application when executing these commands. You must activate the application before executing any clipboard commands.

3.16.1 Copy

The example below shows how to copy the contents of art layer 2 to the clipboard.

```
AS:
```

```
activate
```

```
select all of current document
```

```
copy art layer 2 of current document
```

NOTE: In AppleScript, you must select the entire layer before performing the copy.

```
VB:
```

```
docRef.ArtLayers(2).Copy
```

```
JS:
```

```
docRef.artLayers[1].copy();
```

3.16.2 Copy merged

You can also perform a merged copy. This will make a copy of all visible layers in the selected area.

In AppleScript, use the copy merged command.

AS:

```
activate  
select all of current document  
copy merged selection of current document
```

In VB and JS, pass true for the Merged parameter of the Copy methods.

VB:

```
docRef.Selection.Copy True
```

JS:

```
docRef.selection.copy(true);
```

3.16.3 Cut

The Cut command operates on both art layers and selections, so you can cut an entire art layer or only the selection of a visible layer. The Cut method follows the same rules as the copy command.

AS:

```
activate  
cut selection of current layer of current document
```

VB:

```
docRef.Selection.Cut
```

JS:

```
docRef.selection.cut();
```

3.16.4 Paste

The paste command can be used on any open document, and operates on the current document. You must make the paste command's target document the current document before using the command. A new layer is created by the paste command, and a reference to it is returned. If there is no selection in the target document, the contents of the clipboard are pasted into the geometric center of the document.

Here's how to paste into a document named "Doc2":

AS:

```
activate
set current document to document "Doc2"
set newLayerRef to paste
```

In Visual Basic and JavaScript the paste command is defined on the Document object.

VB:

```
appRef.ActiveDocument = appRef.Documents("Doc2")
Set newLayerRef = docRef.Paste
```

JS:

```
activeDocument = documents["Doc2"];
var newLayerRef = docRef.paste();
```

3.16.5 Paste into command

The paste into command allows you to paste the contents of the clipboard into the selection in a document. The destination selection border is then converted into a layer mask. As for the paste command, you must make the paste command's target document the current document before using the command.

AS:

```
activate
set newLayerRef to paste with clipping to selection
```

VB:

```
Set newLayerRef = docRef.Paste (True)
```

JS:

```
newLayerRef = docRef.paste( true );
```

3.17 Action Manager scripting

The Action Manager allows you to write scripts that target functionality that is not otherwise accessible. You are able to script third party plug-ins, filters, and other tasks that are not otherwise included in the scripting interface. The only requirement is that the task that you want to access from the Action Manager is recordable.

The classes “ActionDescriptor”, “ActionReference” and “ActionList” are all part of the Action Manager functionality.

When you write scripts that use the Action Manager, you should install the “ScriptingListener” plug-in. It is located inside the “utilities” folder that is part of the scripting support download.

To install the plug-in place it in the Photoshop 7.0\Plug-Ins\Scripting\ folder.

“ScriptingListener” records most of your actions to a file on your hard drive. To avoid slowing down Photoshop as well as not to create a big file on your drive, only install the plug-in when you are creating Action Manager scripts.

When “ScriptingListener” is installed it will record a file with scripting code corresponding to the actions that you perform from the UI.

The Windows version of “ScriptingListener” creates the following 2 files:

- **C:\ScriptingListenerJS.log**: contains JavaScript code corresponding to the actions that are performed from the UI.
- **C:\ScriptingListenerVB.log**: contains VBScript code corresponding to the actions that are performed from the UI.

The Macintosh version “ScriptingListener” creates the following file:

- **ScriptingListenerJS.log**: the file is created on the desktop, and contains JavaScript code corresponding to the actions that are performed from the UI.

Note: There is no AppleScript interface to the Action Manager, but you can execute JavaScripts from AppleScript, so you are able to access Action Manager functionality from AppleScripts. See section, [“Running JavaScript based Action Manager code from AppleScript” on page 86](#) for more information on how to call JavaScript code from AppleScript.

3.17.1 Using the Action Manager from JavaScript

As an example let’s say that you want to be able to use the Emboss filter. The Emboss filter is not part of the filters that are exposed to the various scripting languages, but using the Action Manager you are able to use this filter. First make sure that you have installed the “ScriptingListener”. Then from the UI, open a document and apply the Emboss filter using the settings: angle 135, height 3 and amount 100.

When the ScriptingListener is installed, running the Emboss filter is recorded to a file called “ScriptingListenerJS.log” (see above for location of this file on the various platforms).

Open the “ScriptingListenerJS.log” file. At the end of the file you will see something like the following. Note the numbers may vary:

```
var id19 = charIDToTypeID( "Embs" );
var desc4 = new ActionDescriptor();
var id20 = charIDToTypeID( "Angl" );
desc4.putInteger( id20, 135 );
var id21 = charIDToTypeID( "Hght" );
desc4.putInteger( id21, 3 );
var id22 = charIDToTypeID( "Amnt" );
desc4.putInteger( id22, 100 );
executeAction( id19, desc4 );
```

The ScriptingListener divides every command by a line, so it is easy to find the last command.

The next step in making Emboss scriptable is to identify the values that you entered (135, 3 and 100). Copy the JavaScript code from the “ScriptingListenerJS.log” file to another file and substitute the filter values with variable names. In the following we have wrapped the code in a JavaScript function and replaced 135 with angle, 3 with height, and 100 with amount.

```
function emboss( angle, height, amount )
{
    var id32 = charIDToTypeID( "Embs" );
    var desc7 = new ActionDescriptor();
    var id33 = charIDToTypeID( "Angl" );
    desc7.putInteger( id33, angle );
    var id34 = charIDToTypeID( "Hght" );
    desc7.putInteger( id34, height );
    var id35 = charIDToTypeID( "Amnt" );
    desc7.putInteger( id35, amount );
    executeAction( id32, desc7 );
}
```

You now have a JavaScript function that performs the Emboss filter on the current document. To activate the Emboss filter from JavaScript you must include the function definition shown above and then call the function with the desired parameters. To apply Emboss with angle 75, height 2 and amount 89, you say:

```
// First include the emboss function somewhere in your JavaScript
// file
function emboss( angle, height, amount )
{
    var id32 = charIDToTypeID( "Embs" );
    var desc7 = new ActionDescriptor();
    var id33 = charIDToTypeID( "Angl" );
```

```
desc7.putInteger( id33, angle );
var id34 = charIDToTypeID( "Hght" );
desc7.putInteger( id34, height );
var id35 = charIDToTypeID( "Amnt" );
desc7.putInteger( id35, amount );
executeAction( id32, desc7 );
}
// Then call emboss with desired parameters
emboss( 75, 2, 89 );
```

Running JavaScript based Action Manager code from AppleScript

As there is no Action Manager functionality in AppleScript you will have to use JavaScript to use the Action Manager on the Mac. To do this you use the AppleScript command: “do javascript.” Provide filter settings in the “arguments” parameter.

You need to re-write your JavaScript code slightly to work with the “do javascript” command to use the “arguments” collection to get access to the AppleScript values from JavaScript. For example change the Emboss JavaScript shown in the previous section to the following and save it in a file called “Emboss.js”:

```
function emboss( angle, height, amount )
{
    var id32 = charIDToTypeID( "Embs" );
    var desc7 = new ActionDescriptor();
    var id33 = charIDToTypeID( "Angl" );
    desc7.putInteger( id33, angle );
    var id34 = charIDToTypeID( "Hght" );
    desc7.putInteger( id34, height );
    var id35 = charIDToTypeID( "Amnt" );
    desc7.putInteger( id35, amount );
    executeAction( id32, desc7 );
}
// Call emboss with values provided in the "arguments" collection
emboss( arguments[0], arguments[1], arguments[2] );
```

From AppleScript you can then run the Emboss filter by saying:

```
tell application "Photoshop 7.0"
    do javascript (file <a path to Emboss.js>) ↵
        with arguments { 75,2,89 }
end tell
```

Running JavaScript based Action Manager code from VBScript

From VBScript you have a choice of either running JavaScript based Action Manager code or VBScript based Action Manager code. This section describes how to access JavaScript based

Action Manager code. The next section covers how to run VBScript based Action Manager code.

To access JavaScript code from VBScript, you must use the “DoJavaScriptFile” command and provide specific settings in the “arguments” parameter.

Save the following script in a file called “C:\Emboss.js”

```
function emboss( angle, height, amount )
{
    var id32 = charIDToTypeID( "Embs" );
    var desc7 = new ActionDescriptor();
    var id33 = charIDToTypeID( "Angl" );
    desc7.putInteger( id33, angle );
    var id34 = charIDToTypeID( "Hght" );
    desc7.putInteger( id34, height );
    var id35 = charIDToTypeID( "Amnt" );
    desc7.putInteger( id35, amount );
    executeAction( id32, desc7 );
}
// Call emboss with values provided in the "arguments" collection
emboss( arguments[0], arguments[1], arguments[2] );
```

From VBScript you can then run the Emboss filter by saying:

```
Set objApp = CreateObject("Photoshop.Application")
objApp.DoJavaScriptFile "C:\Emboss.js", Array(75, 2, 89)
```

3.17.2 Using the Action Manager from VBScript

Using the Action Manager from VBScript is very similar to using it from JavaScript. When you have “ScriptingListener” installed on Windows, your actions will be recorded as VBScript code in the file “C:\ScriptingListenerVB.log”.

If you run the Emboss filter from the UI and you have the “ScriptingListener” plug-in installed you will see code as the following text at the end of “C:\ScriptingListenerVB.log”; note the numbers may vary:

```
REM =====
DIM objApp
SET objApp = CreateObject("Photoshop.Application")
DIM id19
id19 = objApp.CharIDToTypeID( "Embs" )
DIM desc4
SET desc4 = CreateObject( "Photoshop.ActionDescriptor" )
DIM id20
id20 = objApp.CharIDToTypeID( "Angl" )
Call desc4.PutInteger( id20, 135 )
```

```
DIM id21
id21 = objApp.CharIDToTypeID( "Hght" )
Call desc4.PutInteger( id21, 3 )
DIM id22
id22 = objApp.CharIDToTypeID( "Amnt" )
Call desc4.PutInteger( id22, 100 )
Call objApp.ExecuteAction( id19, desc4 )
```

To make Emboss scriptable you must identify the filter values that you entered (135, 3 and 100). Copy the VBScript code from the “ScriptingListenerVB.log” file to an other file and substitute the filter values with variable names. In the following we have wrapped the code in a VBScript sub-routine and replaced 135 with Angle, 3 with Height, and 100 with Amount.

```
Sub Emboss( Angle, Height, Amount )
DIM objApp
SET objApp = CreateObject("Photoshop.Application")
DIM id19
id19 = objApp.CharIDToTypeID( "Embs" )
DIM desc4
SET desc4 = CreateObject( "Photoshop.ActionDescriptor" )
DIM id20
id20 = objApp.CharIDToTypeID( "Angl" )
Call desc4.PutInteger( id20, Angle )
DIM id21
id21 = objApp.CharIDToTypeID( "Hght" )
Call desc4.PutInteger( id21, Height )
DIM id22
id22 = objApp.CharIDToTypeID( "Amnt" )
Call desc4.PutInteger( id22, Amount )
Call objApp.ExecuteAction( id19, desc4 )
End Sub
REM Now run the Emboss filter by invoking the Emboss sub-routine
Call Emboss( 135, 3, 98 )
```

If you save the code shown above in a file with an extension of “vbs” you can activate the Emboss filter by double clicking on the file.

To include the Emboss functionality into an existing script, copy the sub-routine into your other VBScript and activate the Emboss functionality by invoking the “Emboss” sub-routine with the appropriate arguments.



Index

A

- Action Manager 84
- AppleScript dictionary 34
- AppleScript Values 12
- Application object 50
 - display dialogs 52
 - opening a document 52
 - preferences 51
 - targeting 50
- Array 13
- Array value type 13

B

- Boolean 12, 13
- breakpoint 27

C

- Channel object 75
 - Channel types 75
 - creating new channels 76
 - setting the active channel 76
- Choosing a scripting language 7
- Clipboard interaction 81
- collections 10
- Color object
 - comparing colors 79
- Color objects 77
 - getting and converting color 78
 - Hex values 78
 - Setting a Color 77
 - web safe colors 79
- COM 8
- Command and methods
 - JavaScript 17
- Commands and methods 17
 - AppleScript 17
 - Visual Basic 17

- Comments in scripts 11
 - AppleScript 11
 - JavaScript 11
 - Visual Basic 11
- Condition field 30
- Conditional statements 17
- Control structures 18
- Conventions in this guide 5
- cross-application capability 8

D

- Debugging
 - AppleScript 23
 - Visual Basic 25
- Defining selections 68
- Display dialogs 52
- Document information 56
- Document manipulation 57
- Document object 54
 - document information 56
 - manipulation 57
 - save options 55
- Documenting scripts 11
- Double 13

E

- Error handling 31
 - AppleScript 31
 - JavaScript 32
 - Visual Basic 31
- Executing JavaScripts from AS or VB 48

F

- Filters 74
- functions 20

H

handlers 20
History object 80

I

Integer 12
Inverting selections 70

J

JavaScript
 Scripts folder 21
 Scripts menu 21
JavaScript Debugging 26
JavaScript source view 27
JavaScript Values 13

L

Layer objects 58
 applying styles 63
 layer sets 61
 linking layer 62
 rotating layers 63
 setting the active layer 61
Layer sets 61
Line continuation characters 11
Line field 30
List 12
List value type 12
Long 13

N

Number 12, 13
Numeric value types 12, 13

O

Object classes 9
object containment hierarchy 33
Object elements or collections 10

Object inheritance 10
Object reference 10, 12, 13
Object references 42
 AppleScript 42
 Visual Basic and JavaScript 43
Opening a document 52
Operators 16
Other scripting languages 8

P

Photoshop actions 6
Photoshop object model 9
Photoshop scripting guidelines 33
Photoshop's type library 35
properties 12

R

Real 12
Record value type 12
Reference 12, 13
Reference value type 12, 13
Repeat count field 30

S

save options 55
Script Breakpoints button 30
Script Breakpoints window 30
Script Debugger window 26
Selections 67
 defining 68
 expand, contract and feather 70
 filling 71
 inverting 70
 loading and storing 73
 replacing 68
 rotating 72
 stroking the border 69
Setting fonts 66
Setting the Active layer 61
stack trace view 27
String 12, 13

- Stroking the selection border 69
- subroutines 20
- superclass 10
- System requirements 6
 - JavaScript 7
 - Mac 7
 - Windows 7

T

- Text item object 64
 - setting fonts 66
 - setting text stroke colors 66
 - setting the contents 65
 - warping text 67
- Text values 12, 13

U

- Units 44
 - AppleScript Length Unit Values 45
 - Changing ruler and type units 48
 - command parameters that take unit values 47
 - object properties that depend on unit values 46
 - special unit value types 46

V

- Value types
 - array 13
 - boolean 12, 13
 - double 13
 - integer 12
 - list 12
 - long 13
 - number 12
 - real 12
 - record 12
 - reference 12, 13
 - string 12, 13
 - text 12, 13
- Variable 14
- Variables
 - Assigning values to 14

- Naming variables 15
- Viewing Photoshop objects, commands and methods 34
 - AppleScript dictionary 34
- Viewing Photoshop objects, commands and methods
 - Visual Basic type library 35
- Visual Basic
 - Object Browser 35
- Visual Basic Values 13

W

- Warping text 67
- Web Safe Color 79
- Windows Scripting Host 7