

Introduction to Scripting in Photoshop

Note: Programmers will often interchange the terms "function," "method," "Routine," "procedure," or "subprogram" for the same thing. For the sake of simplicity and consistency, we'll use the term "method" to describe commands/tasks that are built into the Photoshop DOM and we'll use the term "function" to describe custom commands/tasks that are outside of the Photoshop DOM.

Launching the Extend Script Toolkit 2.

The ExtendScript Toolkit is the text editor you will use to write ExtendScript for these lessons. The ESTK helps make writing scripts easier through syntax-highlighting, auto-complete, and built in debugging. The ESTK is installed when you install Photoshop or the Creative Suite:

Mac: <hard drive>/Applications/Utilities/Adobe Utilities/ExtendScript Toolkit 2/

Windows: C:\Program Files\Adobe\Adobe Utilities\ExtendScript Toolkit 2\

Writing your first script.

I rarely start writing a script from scratch. I'll usually start with a template or existing script and modify it to solve a new workflow problem. I've created several template scripts, which are available on your class CD and website, to help you get started. The most basic template is called [template.jsx](#).

Let's look at the first 4 lines of the script. These are what programmers call "comments." Comments are notes to yourself, or others you share your scripts with, detailing information about script, and what each piece of code does. Notice that the color of the comments is green. The ExtendScript Toolkit color-codes different pieces of text so you can tell comments from code, and make the script more readable.

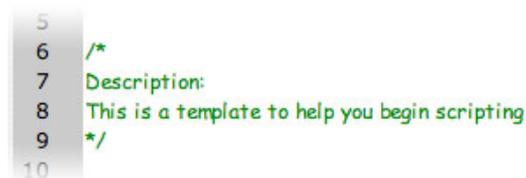


```
1 // template.jsx
2 // Copyright 2006-2008
3 // Written by Jeffrey Tranberry
4 // Photoshop for Geeks Version 2.0
5
```

You create single-line comments by beginning the line with two forward slashes `//` or you can create double-line comments by beginning with a forward slash and an asterisks `/*` and ending it will an asterisk and a forward slash `*/`.

In the case of these four lines, I'm giving some information about the name of the script; when it was written, who wrote it, and what version of the script is if I'm modifying and releasing new versions of the script.

In lines 6-9, I'm using a multi-line comment to give a detailed description of what my script does.



```
5
6 /*
7 Description:
8 This is a template to help you begin scripting
9 */
10
```

Up to this point, everything in this script has been a comment, and hopefully, this gives you a clue that comments are just as important, if not more important, as working code. This will be common theme as we work on scripts. Commenting will make working with or sharing scripts much easier.

The next chunk of code actually does something. Lines 11 and 12 describe what the code on line 13 does. In this case, `#target photoshop` tells the script that it's been created to run in Photoshop if it's been double-clicked in the finder or run from the ExtendScript Toolkit. Since scripts can run in other Creative Suite applications like Bridge, Illustrator, or InDesign, so it's always good form to include this code in every script you write.



```
10
11 // enable double clicking from the
12 // Macintosh Finder or the Windows Explorer
13 #target photoshop
14
```

Line 15 is a comment for the method in line 16 `app.bringToFront()`; which brings Photoshop to the front when

a script is running. I like to include this method in all my scripts because I like to see my scripts do their magic. There's something satisfying about watching Photoshop run under autopilot.

```
14
15 // Make Photoshop the frontmost application
16 app.bringToFront();
17
```

The next three chunks of comments are what I call section headers. I usually organize my scripts into 3 parts: **SETUP**, **MAIN**, and **FUNCTIONS**.

```
17
18 ///////////////////////////////////////////////////
19 // SETUP
20 ///////////////////////////////////////////////////
21
22 ///////////////////////////////////////////////////
23 // MAIN
24 ///////////////////////////////////////////////////
25
26 ///////////////////////////////////////////////////
27 // FUNCTIONS
28 ///////////////////////////////////////////////////
```

In the **SETUP** section, I'll usually declare variables for my script. The **MAIN** section is where I'll put the working code. The **FUNCTIONS** section is where I'll define any custom functions that I'll call in the **MAIN** section of the script. We'll talk more about variables and custom functions as we start to write more complex scripts later on.

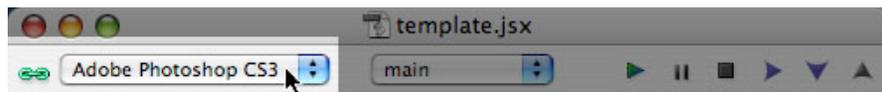
The alert(); method.

The first method we're going to learn is the `alert()` method. An alert simply pops a dialog and displays some information. I believe alerts are an invaluable tool for learning scripting and they will also come in handy later as you start debugging more complex scripts. We'll use alerts for several of our first exercises.

For our first alert, we want to display the text "Hello, World!"

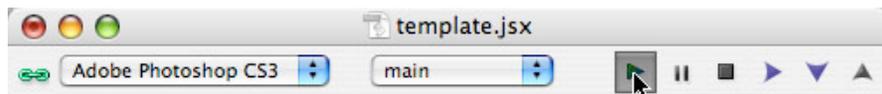
In our [template.jsx](#) script, let's first make a comment of what we want our alert to do. Then on the next line we'll type in our method `alert()`; and pass in the string "Hello, World!" as an argument in between the parenthesis:

Let's test our script. Because I have the command `#target photoshop` in my script, I don't need to specify a target in the ESTK. If I didn't have the command `#target photoshop` I'd have to manually select "Photoshop" from the pop-up in the upper left-hand corner of the script window.



If the link icon next to the pop-up is broken and red, that usually means that Photoshop isn't launched and running. You can either manually launch Photoshop or run your script and the ESTK will ask if you want to launch Photoshop or not. If Photoshop is launched and your scripts fail, make sure you are targeting the right program. Often times, I'll forget to include the command for `#target photoshop` and the script will try to run in the ESTK instead of Photoshop.

Click the play button in the upper right-hand corner of the script window to play your script.

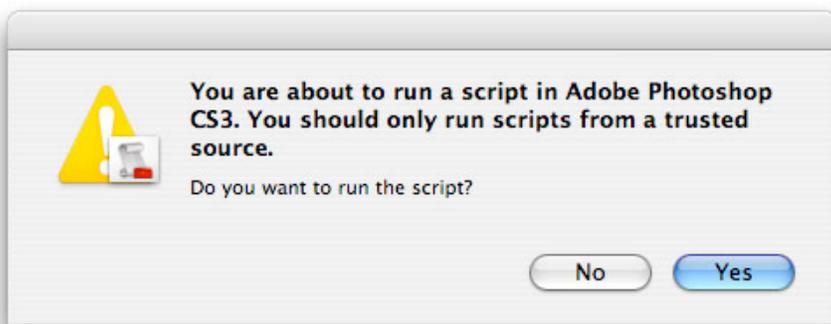


Photoshop should come to the front and display a dialog with our "Hello, World!" text in it.

Once we've tested our script and we know it works, let's finish commenting and save our script [helloWorld.jsx](#) to the desktop.

Running and Installing Scripts.

To run a script from the finder, simply double-click the script icon or drag the script onto the Photoshop application. If you run a script this way, you will be asked whether you want to run the script or not.



To get your script to show up in Photoshop's File>Scripts menu, navigate to the Presets>Scripts folder inside the Photoshop application folder on your hard drive and copy or place your script in the folder.

Mac: <hard drive>/Applications/Adobe Photoshop CS3/Presets/Scripts/
Windows: C:\Program Files\Adobe\Adobe Photoshop CS3\Presets\Scripts\

If Photoshop is already running, you'll need to quit and restart Photoshop for the script to show up in the menu.

When you run scripts from inside Photohop, you won't be asked whether you want the script to run or not, and the script will just execute.

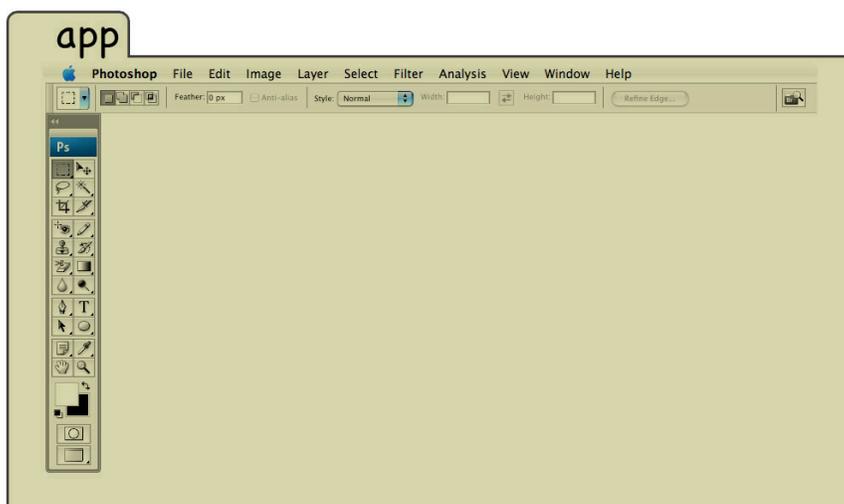
You can assign a keyboard shortcut to your script using the Edit>Edit Shortcuts... dialog. Tip: Don't use the option key for a keyboard shortcut for a script. The option key puts the script into "debug" mode and will launch the ESTK. You can also record an action that runs a script, and assign an F-key to the action to run it.

The Photoshop DOM.

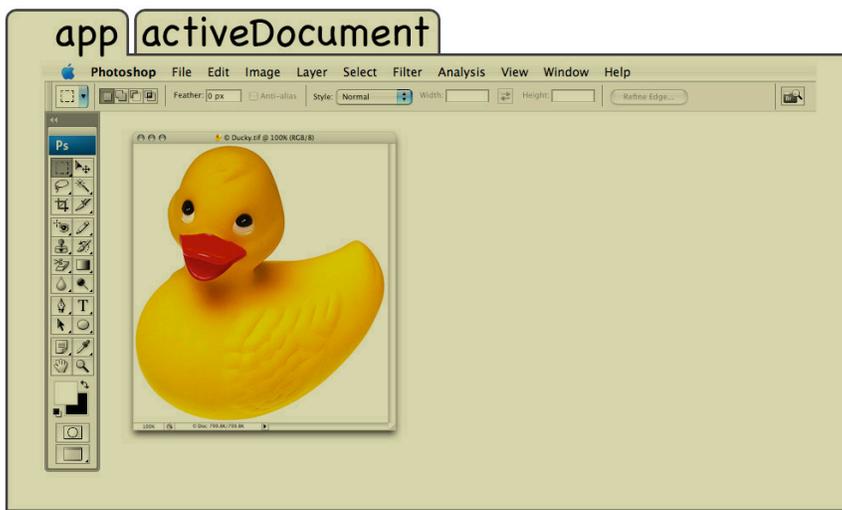
Let's start exploring the Photoshop DOM. What is a DOM? DOM stands for Document Object Model. Document Object Model is a programming interface that lets you interact with Photoshop to examine and manipulate the Photoshop application, Photoshop documents, and the objects contained within the document.

One way to think of the DOM is to think of a series of folders put inside one another. You can imagine either physical folders, or electronic folders on your hard drive or like a website path.

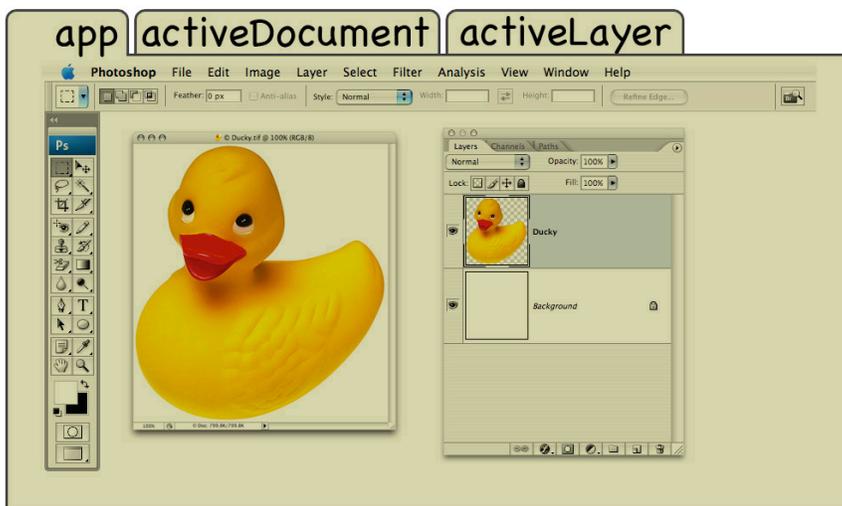
Each folder can contain various objects. Imagine the first, outside folder containing everything else in side it. That's Photoshop or app.



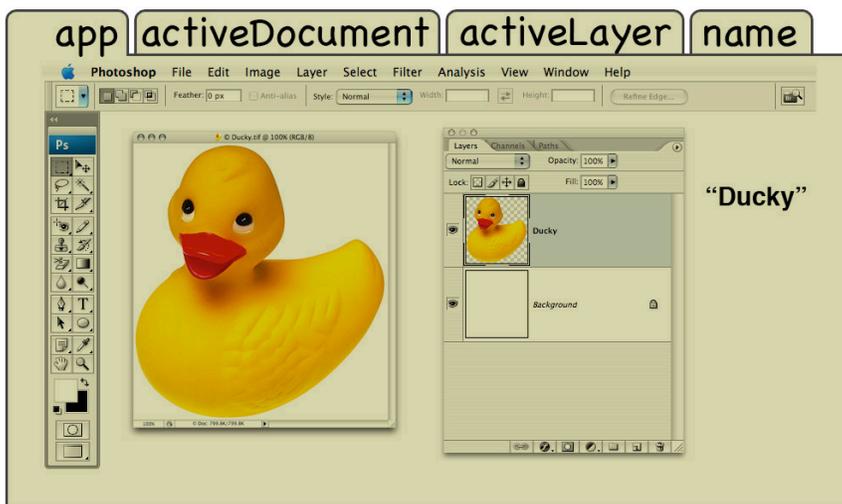
Now imagine there's another folder inside the Photoshop folder. That's the Photoshop document you are working on, or the `activeDocument`.



Now imagine there's another folder nested inside the document folder. That's the layer you are working on, or the activeLayer.



Now imagine the layer folder contains several sheets of paper in it. Each sheet of paper has a unique piece of information about the layer: it's name, it's visibility, it's opacity, it's blendmode, whether it's locked or not, etc.



So, if I wanted to look at the sheet of paper with the current layers name, I'd need to go into the Photoshop folder, then the document folder, then the layer folder, and look for the sheet with name on it. In the scripting world, the code `app.activeDocument.activeLayer.name` would be how you get the current layer's name.

Let's have Photoshop show an alert with the current layer's name. Start with the [template.jsx](#) script and add an alert, but this time, let get the current layer's name and display that in the alert dialog:

```
22
23 ///////////////////////////////////////////////////
24 // MAIN
25 ///////////////////////////////////////////////////
26
27 // Display a dialog with the current Layer's name
28 alert(app.activeDocument.activeLayer.name);
29
```

Make sure there is a document open in Photoshop that contains some layers and one of the layers is selected.

Press play to test our script. Photoshop displays a dialog with the current layer's name on it. ([layerName.jsx](#))

Using the Object Model Viewer to explore the Photoshop DOM.

Now that we know a little bit about how Photoshop Document Object Model works, let's take a look at Object Model Viewer.

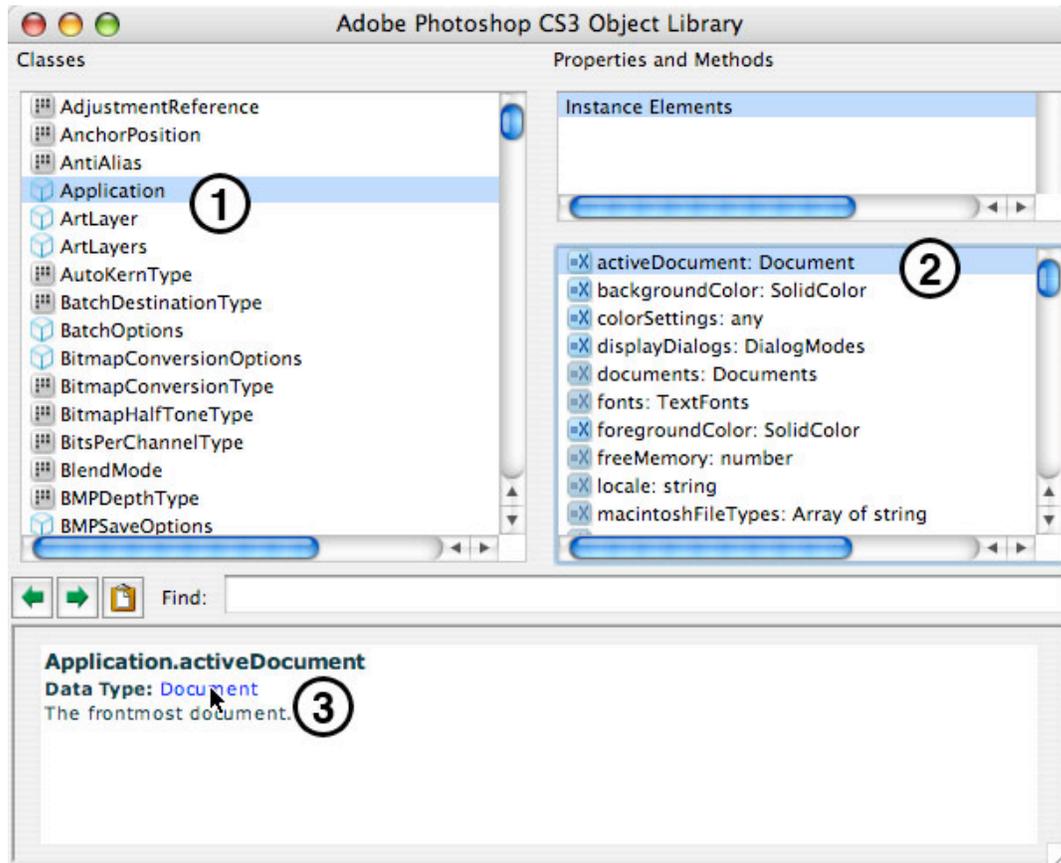
In the ExtendScript Toolkit, go to the Help menu and choose Adobe Photoshop CS3 Object Library. Let's navigate through the object classes and inspect the properties and methods associated with each class.

It usually takes 3 clicks to navigate to each class.

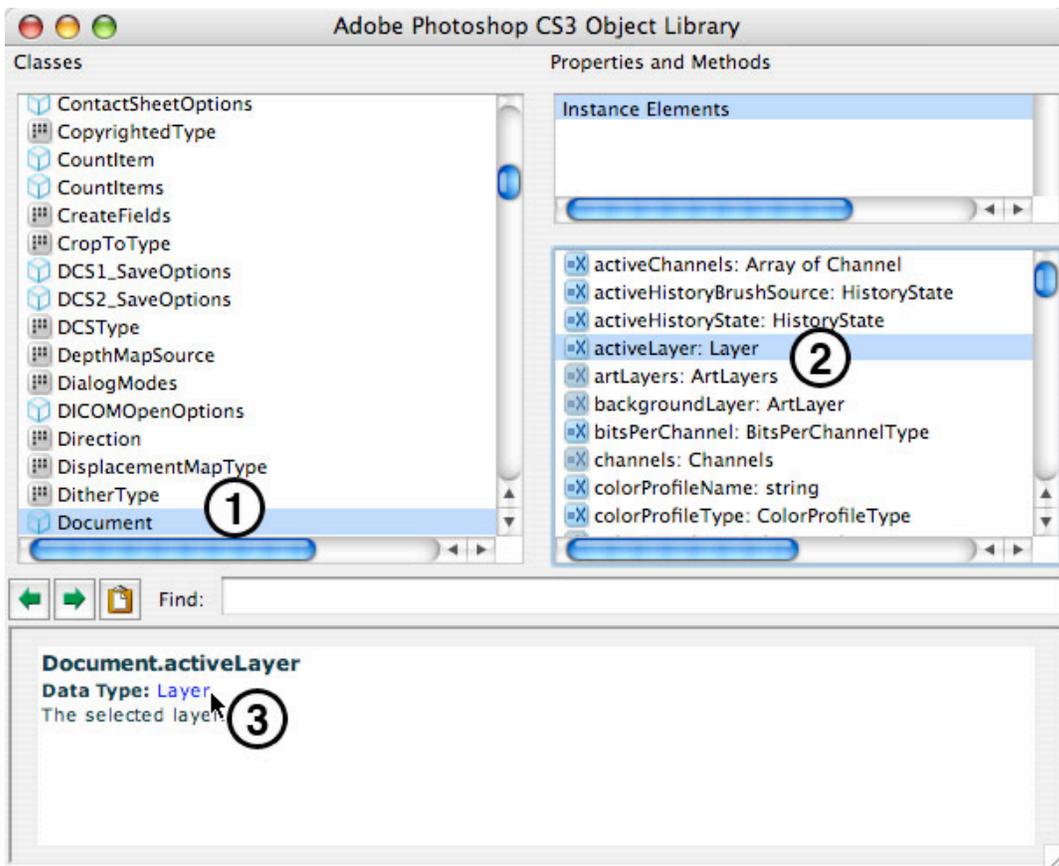
First, click on Application in the class panel of the Object Model Viewer to examine all the properties and methods of the Application or app.

Second, select activeDocument from the properties and methods panel.

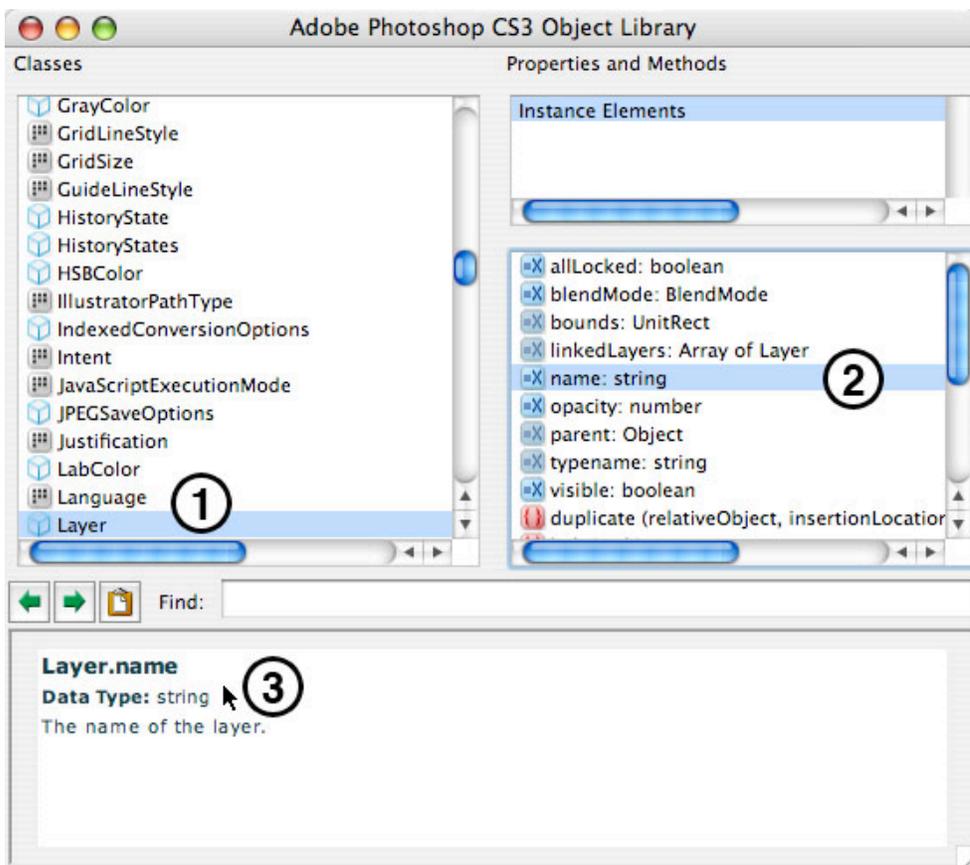
Third, click on the hyperlink Document in the status panel. This selects the Document class in the class panel:



We can now dig deeper into the DOM, by selecting activeLayer in the properties and methods panel and click on the Layer hyperlink in the status panel.



Notice how we can see all the properties and methods of the `activeLayer` in the properties and methods panel? Properties are preceded by a blue icon and methods are preceded by a red icon. Methods are always followed by `()`.



We can see the property for name that we used in the previous example. Experiment alerting and setting the value of some other properties such as `blendMode`, `opacity`, `allLocked`, and `visible`. Note that some properties like `bounds`, `typename`, and `parent` are "Read Only" meaning that you can get and alert their value, but you can't set

their value. ([layerProperties.jsx](#))

```
25
26 // Display a dialog with the current Layer's blendMode
27 alert(app.activeDocument.activeLayer.blendMode);
28
29 // Display a dialog with the current Layer's opacity
30 alert(app.activeDocument.activeLayer.opacity);
31
32 // Display a dialog with the current Layer's lock state
33 alert(app.activeDocument.activeLayer.allLocked);
34
35 // Display a dialog with the current Layer's visibility state
36 alert(app.activeDocument.activeLayer.visible);
37
38 // Set the current Layer's blendMode to MULTIPLY
39 app.activeDocument.activeLayer.blendMode = BlendMode.MULTIPLY;
40
41 // Set the current Layer's opacity to 50%
42 app.activeDocument.activeLayer.opacity = 50;
43
44 // Set the current Layer's locked state to false
45 app.activeDocument.activeLayer.opacity = false;
46
47 // Set the current Layer's visibility to false
48 app.activeDocument.activeLayer.opacity = false;
49
50 // Display a dialog with the current Layer's bounds (Read Only)
51 alert(app.activeDocument.activeLayer.bounds);
52
```

Let's try working with some methods. We can `remove()`; the layer, which does the same thing as dragging the layer to the trash button in the layers palette.

```
25
26 // Remove the current layer
27 app.activeDocument.activeLayer.remove();
28
```

Create a new document with a few layers and try running a script that uses the `remove()`; method. ([deleteLayer.jsx](#))

Solving a real-world problem with scripting.

A customer approached me and said that they recently discovered the command for File>Automate>Crop and Straighten Photos for automating part of their workflow. They had placed several 4x5 pictures on their scanner and created one image. Crop and Straighten Photos takes that image and automatically cuts out of the images and places them into their own documents. They were amazed at the time they would save using this command, but they wished there was a way to automatically name and save each of the resulting documents so they could process a folder of these multi-image scans.

I was able to write a script, [cropAndStraightenBatch.jsx](#), that solved this person's workflow in a matter of minutes. It's a good script to learn from because it combines many, smaller useful scripting techniques into one larger workflow script. As we build this script, you will learn about using variables, renaming & saving files, opening & closing files, using for & while loops, and using the ScriptingListener plug-in.

Designing your script.

Before I begin scripting, I'll write out what I want my script to do as comments in plain English, or what programmers call pseudo code. This is useful for laying out the groundwork of your script. It's also useful if you plan to hand off instructions if you work with a programmer to help write your script. Here is my script outline:

```
// Make sure there are no open documents
// Ask the user to choose the folder of documents to process
// Ask the user to choose an output folder
// Open each document in the folder, one at a time and then process it (for loop)
    // Store the name of the parent document we're going to crop and straighten from
```

```

// Crop and straighten the document, which will result in more than one open document, including the parent document
// Close the parent document without saving
// Process the remaining open documents until no documents are left open (while loop)
// Flatten the document in case the file type we want to save to requires a flat doc
// Save as a JPEG to the selected output folder
// Close the document

```

There's a good reason why I've formatted my comments with indentation the way I have and you'll see later in our lesson why I've done this.

The ScriptingListener plug-in.

The first piece of code we need is for Crop and Straighten Photos. If we look in the Object Model Viewer, you'll discover that there is no method for Crop and Straighten Photos in Photoshop's DOM. Don't distress. There are a lot of commands in Photoshop that aren't in the DOM. That's where the ScriptingListener plug-in comes in handy.

Installing the ScriptingListener plug-in:

- 1) Quit Photoshop
- 2) Locate the ScriptingListener plug-in inside Photoshop's application folder: Adobe Photoshop CS3>Scripting Guide>Utilities>ScriptingListener
- 3) Copy the ScriptingListener plug-in into Photoshop's Plug-Ins folder: Adobe Photoshop CS3>Plug-Ins
- 2) Launch Photoshop

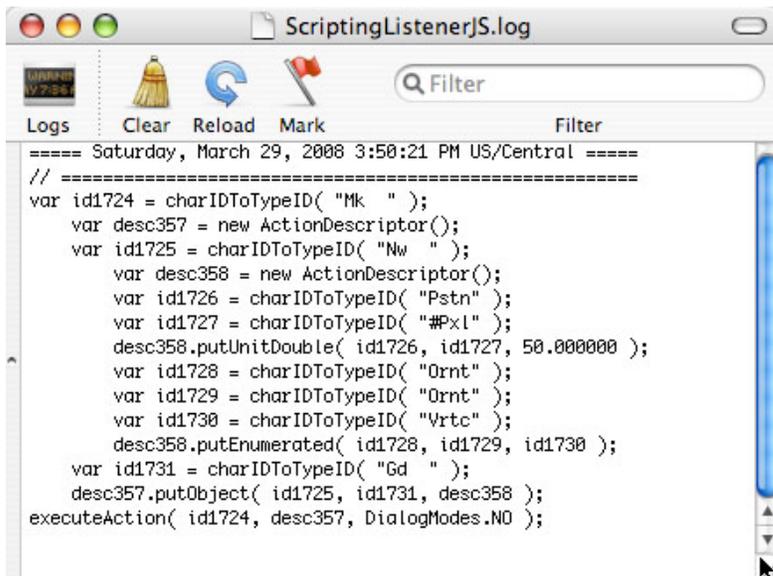
Using the ScriptingListener plug-in:

As you work in Photoshop, the ScriptingListener plug-in records Javascript for any operation which is Actionable to a log file named `ScriptingListenerJS.log`, which by default is saved to the desktop.

To determine if an operation is Actionable, double-click the `ScriptingListenerJS.log` file to launch the Console application. Keep the Console window view of the `ScriptingListenerJS.log` visible as you are working. As you do operations in Photoshop, you will see the log update if the operations you perform are Actionable. Painting, for example, is not an Actionable operation. You can click 'Clear' at any point to clear the entries in the log.

As you will notice, the Javascript recorded by the ScriptingListener plug-in isn't always easily read or clearly labeled.

For Example, if I create a guide in Photoshop, the ScriptingListener Javascript looks something like this:



```

===== Saturday, March 29, 2008 3:50:21 PM US/Central =====
// =====
var id1724 = charIDToTypeID( "Mk " );
var desc357 = new ActionDescriptor();
var id1725 = charIDToTypeID( "Nw " );
var desc358 = new ActionDescriptor();
var id1726 = charIDToTypeID( "Pstn" );
var id1727 = charIDToTypeID( "#Pxl" );
desc358.putUnitDouble( id1726, id1727, 50.000000 );
var id1728 = charIDToTypeID( "Ornt" );
var id1729 = charIDToTypeID( "Ornt" );
var id1730 = charIDToTypeID( "Vrtc" );
desc358.putEnumerated( id1728, id1729, id1730 );
var id1731 = charIDToTypeID( "Gd " );
desc357.putObject( id1725, id1731, desc358 );
executeAction( id1724, desc357, DialogModes.NO );

```

Try to get in the habit of recording one step at a time and commenting each operation so you know what it does. I generally record the operation a few times with different settings so I can see where the settings change in the code. In this case, I've determined that `72.000000` refers to the pixel position of the guide, and that `"Vrtc"` is used for vertical guides and `"Hrzn"` is used for horizontal guides.

You can also turn the Javascript that the ScriptingListener plug-in generates into your own functions. You can even pass in your own arguments. In this case, we pass in an integer for `pixelOffset` and the string `"Vrtc"`, `"Hrzn"` for orientation:

```

32
33 // this is a function to create a guide
34 // Arguments: integer for pixelOffset
35 // and a string of either "Vrtc", "Hrzn" for orientation
36 function makeGuide(pixelOffset, orientation) {
37     var id8 = charIDToTypeID( "Mk " );
38     var desc4 = new ActionDescriptor();
39     var id9 = charIDToTypeID( "Nw " );
40     var desc5 = new ActionDescriptor();
41     var id10 = charIDToTypeID( "Pstn" );
42     var id11 = charIDToTypeID( "#Rlt" );
43     desc5.putUnitDouble( id10, id11, pixelOffset ); // integer
44     var id12 = charIDToTypeID( "Ornt" );
45     var id13 = charIDToTypeID( "Ornt" );
46     var id14 = charIDToTypeID( orientation ); // "Vrtc", "Hrzn"
47     desc5.putEnumerated( id12, id13, id14 );
48     var id15 = charIDToTypeID( "Gd " );
49     desc4.putObject( id9, id15, desc5 );
50     executeAction( id8, desc4, DialogModes.NO );
51 }
52

```

Then you can simply call the function every time you want to use it:

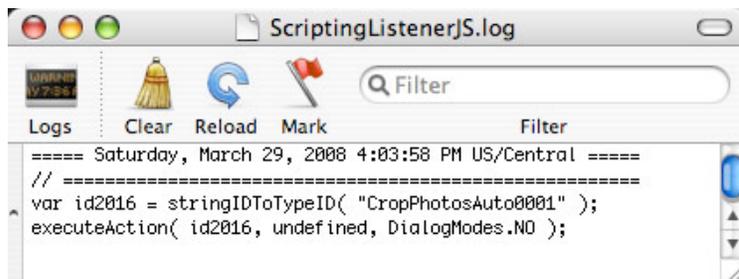
```

25
26 // create a vertical guide 72 pixels in from the left edge
27 makeGuide(72, "Vrtc");
28

```

([makeGuide.jsx](#))

To get the code for Crop and Straighten Photos, I'll open the sample file Vacation.tif inside Photoshop's application folder: Adobe Photoshop CS3>Samples>Vacation.tif. The the ScriptingListener Javascript looks like this:



Since there are no arguments, the function for Crop and Straighten Photos looks like this:

```

43 // Crop and Straighten function created
44 // using the ScriptingListener plug-in
45 function cropAndStraighten(){
46     var id333 = stringIDToTypeID( "CropPhotosAuto0001" );
47     executeAction( id333, undefined, DialogModes.NO );
48 }

```

Save the script as [cropAndStraighten.jsx](#).

Conditional scripts.

Conditionals are a very powerful feature of scripting. You can do an operation if certain conditions are met. A common one is to run different actions if the document is wider than it is tall, or if the document is taller than it is wide, using an if/else statement:

Other common conditionals check to make sure the document is in the right color mode or bit depth, or the current layer is a text or raster layer.

For our script, we want to make sure there aren't any open documents open before we run our script. It is a good idea to do this to avoid damaging or losing unsaved changes to an open document, especially if your script does any sort of batch

processing.

Here is the code to check for open documents:

```
25
26 //Make sure there are no open documents
27 if (app.documents.length > 0){
28     alert ("This script requires that there are no open documents to run.");
29 }else{
30     alert("OK to do a batch process.");
31 }
32
```

Save the script as [openDocumentConditional.jsx](#).

Renaming documents.

Another common use of scripting is to intelligently name/rename documents and save them.

For example, you might have a group of images of birds from your camera with a file name like "IMG_0918.tif" and you want to save it as a ".jpg" file. First, we'll need to do a little bit of coding to slice off the original ".tif" extension, in order to append on the new ".jpg" extension. You do this using the `slice()` method. The `slice` method expect two arguments, the `startSlice` and the `endSlice`. Imagine the string "IMG_0918.tif" is a loaf of bread, and each character is a slice of bread. If I want to get rid of the last four characters ".tif" I need to count back from the right edge of the word. So my `endSlice` number would be `-4`. Since I'm not slicing off anything at the start, I use `0` for the `startSlice` number. So the code would look like this:

```
27
28 // Have Photoshop display an alert box that
29 // displays the active document name as a string
30 // but slice off the last four characters
31 alert(app.activeDocument.name.slice(0,-4));
32
```

We can also slice off the useless prefix "IMG_" and add a new prefix, like "CraigsBirds_". For this, we're going to count forward the number of characters we want to remove, so the `startSlice` number would be `4`. So to slice off that first four and last four characters, the code would look like this:

```
32
33 // Have Photoshop display an alert box that
34 // displays the active document name as a string
35 // but slice off the last four characters
36 // and the first four characters
37 alert(app.activeDocument.name.slice(4,-4));
38
```

Now, we just need to add our new prefix and suffix. We're going to use the `+` sign operator to concatenate, or append, our new prefix and suffix onto our sliced up name. The `+` sign, is what is called an operator in JavaScript. Here's an example of using an operator:

```
38
39 // Have Photoshop display an alert box that
40 // displays the active document name as a string
41 // but slice off the last four characters
42 // and the first four characters
43 // and add "CraigsBirds_" to front of the file name
44 alert("CraigsBirds_" + app.activeDocument.name.slice(4,-4) + ".jpg");
45
```

Save your file as

[fileNaming.jsx](#).

Flattening documents.

Often times, when you are processing/exporting documents, you'll want to flatten the layers in your document in order to make your document smaller and save it as a file type that doesn't support layers, like JPEG or PNG. If you don't flatten before saving as a JPEG, it will pop a dialog and pause your script.

Flattening your documents takes up less memory, and makes subsequent steps in your script process faster. To do this, we'll use the `flatten()` method on the `activeDocument`:

```
25
26 // Flatten the document in case the file type we want to save to requires a flat doc
27 app.activeDocument.flatten();
28
```

[\(flatten.jsx\)](#)

Saving documents.

When saving files other than Photoshop native format (PSD), you sometimes need to specify the parameters from the dialog. For example the JPEG dialog has many settings. In JavaScript code you create a `JPEGSaveOptions` and use that to pass the parameters to the `saveAs` method.

The `saveAs` method takes the arguments for `saveIn`, `options`, `asCopy`, and `extensionType`. The argument `saveIn` is the path and file name where the file will be saved. `Folder.desktop` is the cross-platform, and language independent way to specify the users desktop.

The argument `options` are the parameters from the JPEG dialog. In this case, we set the quality to 10 and whether to `embedColorProfile` to `false`.

The argument `asCopy` determines whether to save the document as a copy, leaving the original open, in this case we say `false`.

The argument `extensionType` allows the user to specify `NONE`, `UPPERCASE` or `LOWERCASE`. Since we're passing in a string of the file name, we won't pass in this argument.

The final code to save a JPEG file to the desktop ([saveJPEG.jsx](#)) looks like this:

```
27
28 //Save as a JPEG to the users desktop
29 var jpegOptions = new JPEGSaveOptions();
30 jpegOptions.quality = 10;
31 jpegOptions.embedColorProfile = false;
32 app.activeDocument.saveAs( File( Folder.desktop + "/" + activeDocument.name + ".jpg"), jpegOptions, false);
33
```

Closing a document.

After we save our document, we'll want to close it and work on another document. To do this, we'll use the `close()` method on the `activeDocument`:

```
25
26 // Close without saving
27 app.activeDocument.close(SaveOptions.DONOTSAVECHANGES);
28
```

You can specify whether to `DONOTSAVECHANGES`, which closes the document without saving any changes. This is the option I use most often since I'm generally closing documents I know I just saved. You can also specify to `SAVECHANGES`, which saves and closes the document. The last option is `PROMPTTOSAVECHANGES`, which displays a dialog asking the user whether they want to save their document or not. Since I usually want our scripts to run without user interaction, I rarely use this option. ([closeDocument.jsx](#))

There's one last trick we want to do. In our final script, we want to keep track of the original document by capturing it's name as a variable before we run the for Crop and Straighten Photos command. This way we can close it after we run the for Crop and Straighten Photos command. Open the [cropAndStraighten.jsx](#) so we can add a variable `docRef` to store the parent docs name, then we'll the Crop and Straighten Photos command, and finally we'll close the parent document:

```

25
26 // Create a variable to store a reference to
27 // the currently active document, which in this
28 // case is the parent document we want to extract
29 // multiple scanned images from
30 var docRef = app.activeDocument;
31
32 // Run the cropAndStraighten function
33 // which will result in more than one open document
34 cropAndStraighten();
35
36 // Close the parent document we originally opened
37 docRef.close(SaveOptions.DONOTSAVECHANGES);
38

```

Save the changes in the [cropAndStraighten.jsx](#) script.

Using while loops.

We'll use a while loop to save each of the documents that are left open after we run the Crop and Straighten Photos command. A while loop basically iterates through something while a condition still exists. For example, if I go to McDonald's and order some fries, I will continue to eat a French fry while I still have French fries left on my plate. In Photoshop, we'll save and close open documents until they are all saved and closed.

If you aren't well versed in writing loops, I've included a template script for a while loop called [processAllOpenDocuments.jsx](#). Notice, there are two big comments telling you where to put your processing code. Also, notice how there are already two little bits of code you are familiar with? An `alert()`; for displaying the name of the current document and the `close()`; method for closing the document without saving.

```

27
28 // Process all open documents until no documents
29 // are left open.
30 while (app.documents.length >=1){
31
32 ///////////////////////////////////////////////////
33 // Put all your processing functions...
34 ///////////////////////////////////////////////////
35
36 // Have Photoshop display an alert box that
37 // displays the document name as a string
38 alert(app.activeDocument.name);
39
40 // Close without saving
41 app.activeDocument.close(SaveOptions.DONOTSAVECHANGES);
42
43 ///////////////////////////////////////////////////
44 // ...in the area between these two comments.
45 ///////////////////////////////////////////////////
46 }
47

```

Open some documents Photoshop making sure you don't have any documents open that need saving, and run the script. The script will alert the name of each document and close it until there are no documents open.

Let's modify this script. We'll delete the lines for alerting the name of the document, since that's just there to demonstrate how the script works. Save the file as [closeAllOpenDocumentsWithoutSaving.jsx](#). If you're like me, you'll have dozens documents open sometimes, and just want to close all of the files without being asked whether I want to save them or not. Now you have a neat little script to clean up your work area when you are ready to move onto the next project.

Let's move some of the code we've created in some of our previous scripts inside this while loop.

Take our code for flatten image and save as a JPEG and put it in before the step to close the document:

```

32 //////////////////////////////////////////////////
33 // Put all your processing functions...
34 //////////////////////////////////////////////////
35
36 // Flatten the document in case the file type we want to save to requires a flat doc
37 app.activeDocument.flatten();
38
39 // Save as a JPEG to the users desktop
40 var jpegOptions = new JPEGSaveOptions();
41 jpegOptions.quality = 10;
42 jpegOptions.embedColorProfile = false;
43 app.activeDocument.saveAs( File( Folder.desktop + "/" + activeDocument.name.slice(0, -4) + ".jpg"), jpegOptions, false);
44
45 // Close without saving
46 app.activeDocument.close(SaveOptions.DONOTSAVECHANGES);
47 |
48 //////////////////////////////////////////////////
49 // ...in the area between these two comments.
50 //////////////////////////////////////////////////

```

Make sure no other documents are open, and open the sample file `Vacation.tif` inside Photoshop's application folder: Adobe Photoshop CS3>Samples>Vacation.tif. Choose File>Automate>Crop and Straighten Photos. Close the original `Vacation.tif`. Now when we run the script, it saves out all the files created from the Crop and Straighten Photos step the desktop as JPEGs.

Save the script as [saveAllOpenDocuments.jsx](#) for future reference.

Using for loops.

Now that we have a script working to process a single document, we need to make it work on a folder of images we want to Crop and Straighten.

We'll use a for loop to open each document from a selected folder. A for loop basically gets the count of something, like the number of documents in a folder or number of layers in the layer stack and iterates through them until the count is zero.

Again, if you aren't well versed in writing loops, I've included a template script called [openFolder.jsx](#) which contains a for loop that iterates through a selected folder.

If you open the [openFolder.jsx](#), we can examine the template. In line 25, `gFilesToSkip` is an array, or list, of file types to skip. Often times, someone will point to a folder to process that may contain files that Photoshop doesn't open, like Word, or Excel files, which will stop the script with an error that the file can't be opened. To avoid this problem, this template maintains a list of common file types that should be skipped. If you find a file type that gives an error, you can simply add it to the list in line 25:

```

23
24 // A list of file extensions to skip, keep them lower case
25 gFilesToSkip = Array( "db", "xmp", "thm", "txt", "doc", "md0", "tb0", "adobebridgedb", "adobebridgedbt", "bc", "bct" );
26

```

Line 29, pops a dialog asking the user to choose an input folder and stores the path in a variable called `inputFolder`.

Line 33, pops a dialog asking the user to choose an output folder and stores the path in a variable called `outputFolder`. The output folder will be used later in place of the `Folder.desktop` path we used in our previous `saveJPEG` script to save the files to a user selected folder, rather than the user's desktop.

Under, the `MAIN` section, we call the `OpenFolder` function in line 40.

```

30
31 // Pops open a dialog for the user to
32 // choose the folder of documents to process
33 var inputFolder = Folder.selectDialog("Select a folder of documents to process");
34
35 // Pops open a dialog for the user to
36 // set the output folder
37 var outputFolder = Folder.selectDialog("Select a folder for the output files");
38
39 // Open Folder of Images
40 OpenFolder();
41

```

If we look at the `OpenFolder` function under the **FUNCTIONS** section, you'll notice there are two big comments telling you where to put your processing code. Also, notice how there are already two little bits of code you are familiar with? An `alert()`; for displaying the name of the current document, and the `close()`; method for closing the document without saving:

```

59 ////////////////////////////////////////////////////
60 // Put all your processing functions...
61 ////////////////////////////////////////////////////
62
63 // Alert and show the document name
64 alert(app.activeDocument.name);
65
66 // Closes the file without saving
67 app.activeDocument.close(SaveOptions.DONOTSAVECHANGES);
68
69 ////////////////////////////////////////////////////
70 // ...in the area between these two comments.
71 ////////////////////////////////////////////////////

```

If we go back to the **MAIN** section, there's an `alert()`; to show the path of the `outputFolder`.

If we run the script, select a folder of images when prompted for the `inputFolder`, select an `outputFolder`, the script will iterate through each of the images in the `inputFolder`, alerting the document's name and closing it. When it's done going through all the images in the folder, there's an alert with the path of the `outputFolder`.

Putting it all together.

We'll use the [openFolder.jsx](#) template as the starting point to putting all of our scripts together into one master script.

First, let's review our script outline:

```

// Make sure there are no open documents
// Ask the user to choose the folder of documents to process
// Ask the user to choose an output folder
// Open each document in the folder, one at a time and then process it (for loop)

```

The first thing we want to do is make sure there are no open documents. Take the code from our [openDocumentConditional.jsx](#) and past it into the **MAIN** section of the [openFolder.jsx](#) template.

The next thing our outline says we need to do is ask the user to choose the folder of documents to process. Then ask the user to choose an output folder and open each document in the folder, one at a time to process it.

Cut the code for choosing the `inputFolder`, the `outputFolder` and `OpenFolder` function so we can move it to another part of our script. We want to paste this code inside the `else` statement of our open document conditional, so this code is only run if there are no documents open. You can delete the `alert()`; for displaying the `outputFolder` since we don't need it anymore. The **MAIN** section of our script is complete.

```

30
31 //Make sure there are no open documents
32 if (app.documents.length > 0){
33     alert ("This script requires that there are no open documents to run.");
34 }else{
35
36     // Pops open a dialog for the user to choose the folder of documents to process
37     var inputFolder = Folder.selectDialog("Select a folder of documents to process");
38
39     // Pops open a dialog for the user to set the output folder
40     var outputFolder = Folder.selectDialog("Select a folder for the output files");
41
42     // Open and process a folder of Images
43     OpenFolder();
44
45 }
46

```

We'll save our file as [cropAndStraightenBatch.jsx](#), so we don't accidentally save over our [OpenFolder.jsx](#) template script.

Let's go back to our script outline:

```

// Store the name of the parent document we're going to crop and straighten from
// Crop and straighten the document, which will result in more than one open document, including the parent document
// Close the parent document without saving
// Process the remaining open documents until no documents are left open (while loop)

```

Note I'm taking the next four steps that are indented. You'll remember I indented these lines for a reason. That's because these steps will be nested inside our for loop of our `OpenFolder` function. Scroll down to the **FUNCTIONS** section of our [cropAndStraightenBatch.jsx](#) script and find the two big comments we need to put our processing functions.

Now open our [cropAndStraighten.jsx](#) script. Copy the all the code from the **MAIN** section from the [cropAndStraighten.jsx](#) and paste it in between the two big comments in our [cropAndStraightenBatch.jsx](#) script, replacing the two functions that are currently there.

```

64
65     // Create a variable to store a reference to
66     // the currently active document, which in this
67     // case is the parent document we want to extract
68     // multiple scanned images from
69     var docRef = app.activeDocument;
70
71     // Run the cropAndStraighten function
72     // which will result in more than one open document
73     cropAndStraighten ();
74
75     // Close the parent document we originally opened
76     docRef.close(SaveOptions.DONOTSAVECHANGES);
77

```

Lastly, copy the function for `cropAndStraighten()`; from the **FUNCTIONS** section of the [cropAndStraighten.jsx](#) and paste it at the bottom of the page in the **FUNCTIONS** section of our [cropAndStraightenBatch.jsx](#) script.

```

130 // Crop and Straighten function created
131 // using the ScriptingListener plug-in
132 function cropAndStraighten (){
133     var id333 = stringIDToTypeID( "CropPhotosAuto0001" );
134     executeAction( id333, undefined, DialogModes.NO );
135 }

```

That takes care of the first three steps for storing the name of the parent document, crop and straightening the document, and closing the parent document without saving.

Next, we need the code for our fourth step processing the remaining open documents until no documents are left open. Open our script [saveAllOpenDocuments.jsx](#) and copy all the code in the **MAIN** section of the script. We'll paste that code in our [cropAndStraightenBatch.jsx](#) script after the step to close the parent document, at line 78.

```
84 ////////////////////////////////////////////////////
85
86 // Flatten the document in case the file type we want to save to requires a flat doc
87 app.activeDocument.flatten();
88
89 // Save as a JPEG to the outputFolder
90 var jpegOptions = new JPEGSaveOptions();
91 jpegOptions.quality = 10;
92 jpegOptions.embedColorProfile = false;
93 app.activeDocument.saveAs( File( Folder.desktop + "/" + activeDocument.name + ".jpg"), jpegOptions );
94
95 // Close without saving
96 app.activeDocument.close( SaveOptions.DONOTSAVECHANGES );
97
98 ////////////////////////////////////////////////////
```

That takes care of the fourth step that processes the remaining open documents and most of the last three lines of our script outline:

```
// Flatten the document in case the file type we want to save to requires a flat doc
// Save as a JPEG to the selected output folder
// Close the document
```

There's still one thing to do. Instead of saving the JPEGs to the desktop, we want to save them to the `outputFolder` we defined in line 40 of our script. Change `Folder.desktop` in line 93 to `outputFolder`.

```
jpegOptions.quality = 10;
jpegOptions.embedColorProfile = false;
app.activeDocument.saveAs( File( outputFolder + "/" + activeDocument.name
```

Save the file and then run it to test our final script.