# JavaScript for ImageJ: A User's Guide to the JavaScript_Editor Plug-in

**Kas Thomas <kthomas@acrojs.com>**

This document briefly describes the *JavaScript_Editor* plug-in for the Java-based freeware image editor, ImageJ. Information about ImageJ, and the latest downloads, can be obtained from **http://rsb.info.nih.gov/ij/**. The latest information about the *JavaScript_Editor* plug-in can be obtained there or from **http://www.acrojs.com** (the author's web site).

Information about the JavaScript language can be obtained from **http://developer.netscape.com/docs/manuals/**. The core language is standards-based: The standard can be seen at **http://www.ecma.ch/ecma1/stand/ecma-262.htm**.

The JavaScript_Editor plugin relies on the Mozilla Rhino package, which is a Java-based implementation of JavaScript 1.5. Information about that package (including the latest downloads) can be obtained at **http://www.mozilla.org/rhino/**. Note that you *must* obtain the Rhino package (specifically, the **js.jar** file) in order to use the JavaScript_Editor plug-in.

# 1.0 Introduction

## 1.1 Why JavaScript for Image Editing?

One of the most compelling features of ImageJ is its plug-in architecture, which makes it easy for a Java developer to create special-purpose image-processing executables that can operate within a robust, highly functional GUI framework. The ImageJ plug-in API is rich and easy to learn. A graphics programmer can be productive with it in minimal time. Even so, it is possible to see results in *less* time with JavaScript. With Java, a compilation cycle is required in order to see the onscreen behavior of a new plug-in. This cycle is eliminated with JavaScript. Also, when developing in a high-level language such as Java, one must spend a good deal of time specifying, declaring, and converting between data types. With JavaScript, this onus is removed. The interpreter takes care of data-type issues so that the programmer can spend less time trying to satisfy the needs of the Java compiler and more time manipulating pixels.

With JavaScript, it is possible to type one line of code, execute it by clicking a button, and see an image change appearance immediately in response to the code. No compilation cycle is needed; no file need be written to disk. The speed and interactivity of JavaScript make rapid development—and rapid learning—a real possibility. The same considerations apply to debugging: the debugging cycle, with JavaScript, is very short.

Another reason to use scripting in ImageJ is to use it as an *automation tool*, for automating batch operations or lengthy sequences of plug-in calls, etc. A script can call plug-ins, use any ImageJ API calls that a plug-in would use, and (in fact) instantiate any Java class. JavaScript thus provides the ultimate "macro language" for a Java-based image editor.

Yet another reason for having a JavaScript environment within ImageJ is to make it easier for beginning programmers and those with little Java experience (but perhaps some JavaScript experience) to access ImageJ features programmatically.

## 1.2 The Rhino Engine

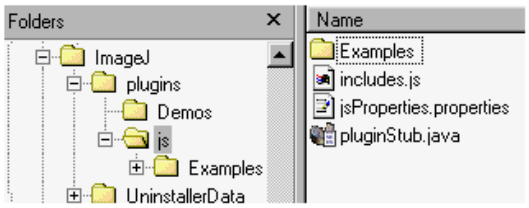
The JavaScript_Editor plug-in uses the Rhino JavaScript interpreter to execute scripts. Rhino is Netscape's open-source implementation of JavaScript 1.5 (part of the Mozilla project). It is a pure-Java implementation of the JavaScript language requiring Java 1.1 or higher at runtime.

Two open-source Java-based interpreters are available on the web: FESI (the Free EcmaScript Interpreter: see **http://home.worldcom.ch/jmlugrin/fesi/index.html**) and Rhino (URL given above). Rhino was chosen for its smaller footprint and more complete language implementation.

# 2.0 Installation

To use the JavaScript_Editor plug-in, you must have the Rhino binary file called **js.jar**. Go to **http://www.mozilla.org/rhino/** and download the latest version of Rhino before attempting to use the JavaScript_Editor plug-in. Be sure the **js.jar** file is in your classpath. If you are launching ImageJ using the *ImageJ.exe* file, you may want to modify your *ImageJ.lax* file to include appropriate path information.

To install the JavaScript_Editor plug-in, unzip the distribution file and make sure the **JavaScript_Editor.java** file is in your ImageJ plug-ins file. Also, make sure there is a folder called **\js** under your ImageJ **\plugins** folder. The **\js** folder should contain the jsProperties.properties file as well as the includes.js file and an Examples folder. *The properties file is required in order to run the plug-in.* The folder layout should look something like:




Here are the files you need and what they do:

- **js.jar**—This is the Rhino JavaScript interpreter jar. It is not part of the JavaScript_Editor distribution; you must download it separately (see URL above). This file is *required*. You can put it anywhere as long as it is visible to ImageJ in the classpath.
- **JavaScript_Editor.java**—This is the Java source code for the script editor. It comprises the ImageJ plug-in source. This file is *required* if you want to compile the plug-in. It should go in your **ImageJ\plugins** folder.
- **jsProperties.properties**—The JavaScript_Editor menus are built from this file. It should be in a folder called **\js** under your **ImageJ\plugins** folder. This file is *required*.
- **pluginStub.java**—This is a small Java file used by the Convert to Plugin command (see discussion below). This file is *optional*, but if you don't have it, the

Convert to Plugin functionality will not be available. It goes under **ImageJ\plugins\js**.

- **includes.js**—This is a small script file containing mostly nonessential convenience functions. (This file is *optional*, but if you don't have it, the Convert to Plugin functionality will not be available.) You can place your own code here, if you have script methods or library routines that you want to load automatically every time the plug-in runs. It goes under **ImageJ\plugins\js**.
- **Examples** folder (sample scripts; nonessential); can go anywhere.

When all files are in place, simply start ImageJ, then use the Compile and Run command to run **JavaScript_Editor.java** for the first time.

## 2.1 Version Information

This version of the plug-in was tested on Windows NT and XP, against ImageJ 1.27 and Rhino 1.5R3, under JDK1.3.0_02. Earlier JVMs will probably work (the plug-in does not rely on 1.3 Java features), but no guarantees are made.

For the latest version of this plug-in, check **http://www.acrojs.com**.

## 2.2 Copyright and License Terms



## 2.3 Disclaimer

You agree to use this plug-in entirely at your own risk and indemnify the author against any harmful outcome of using this software.

For purposes of UCC compliance, the following language must be made in a prominent manner (hence capitalized):

THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

## 2.4 Support

As this is a free product (with full source code), no support is offered.

# 3.0 Basic Operation

When you run the JavaScript_Editor plug-in, you will see a small script editor window with three buttons at the bottom: Clear, Evaluate, and Exit.

- **Clear** deletes all content from the editor window and resets the window name to Untitled.
- **Evaluate** causes the selected text in the editor window to be evaluated by the Rhino interpreter. If no text is selected, the entire window contents are passed to the interpreter.
- **Exit** causes the editor window to close, but if the window contents are "dirty" (i.e., the window contains text or changes that have not been saved to a file), you will be prompted with a "Save Changes?" dialog.

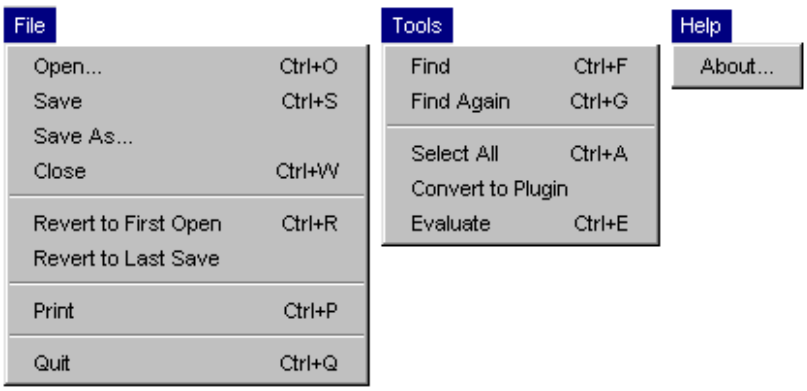
  **NOTE** If you click on the window-close icon in the window title bar (on Windows, this is the small 'X' in the upper right corner of the window frame), the window will close immediately, without prompting you to save your work. *Be careful when using this technique.*

You will use the script editor window to write and test scripts. You can also Save and load (Open) scripts from the script editor's File menu. The editor is a single-document interface (SDI) type of editor, meaning you can work on one document at a time. If you need to work on multiple documents at once, simply run multiple instances of the JavaScript_Editor plug-in (one instance per document).

Note that although you can save and load scripts, window contents do *not* have to be saved to disk first in order to be executed. You can execute the script editor window contents at any time by clicking the Evaluate button or using Control-E. You can also evaluate just a portion of the window content by highlighting (selecting) portions of text with the mouse before hitting Evaluate.

## 3.1 Menus

The script editor has a menubar. The menus are as follows:




The menu commands are self-explanatory, for the most part.

**Open** will allow you to open any file (not just *.js files). It is recommended that you use the "js" extension when saving your scripts, however, and use the script editor only for editing scripts.

There are two **Revert** options, so that you can do an emergency-restore of window contents as they were either at the last file-open event or at the last Save.

**Find** and **Find Again** will do a case-insensitive search from the current cursor position to the end of the window contents.

**Convert to Plugin** associates your script file with a Java file that will silently call the script at runtime. (The current window contents must first be saved to disk.) This command relies on the file called **pluginStub.java** in the **\js** folder. It modifies this stub and copies it to your ImageJ **\plugins** folder. If you compile and run that file, you will be able to call on it to run your script as an ImageJ plug-in.

**Evaluate** has the same effect as clicking on the Evaluate button at the bottom of the window frame.

**About** displays copyright and version information.

## 3.2 How Evaluation Works

When you click Evaluate, your script (the editor window contents, or the selected portion thereof) is collected into a single Java String and sent to the Rhino evaluator, along with the contents of the **includes.js** file.

> **NOTE** The **includes.js** file contains a few convenience routines, such as the constructor for a custom object called RGBImage. You can place your own routines in this file as well. Every time you evaluate a script, the contents of this file become part of the JavaScript runtime scope. Note, too, that you can edit this file in the JavaScript editor, then Save it and have the changes be reflected in every new click of the Evaluate button. You do not have to quit the editor and relaunch in order to see the effect of changes to **includes.js**.

The result of the evaluation will appear in the Results window of ImageJ. For example, if you evaluate the expression "1 + 2", the number 3 will appear in the Results window. On the other hand, if you evaluate an erroneous expression such as "1=2", you will see an error message in the Results window; in this case, "Invalid assignment left-hand side." If you've made a syntax mistake in your script, you will see a helpful diagnostic message in the Results window.

Before the evaluator is called, a custom method in the JavaScript_Editor object exposes certain Java objects in the JavaScript global scope so that they are directly available as JavaScript objects:

- `ImageProcessor`—This global object is the ImageJ ImageProcessor object for *the currently open, front-window image* (if one exists). Otherwise, if there is no image open, it is undefined.
- `ImagePlus`—This global object is the ImageJ ImagePlus object for *the currently open, frontmost image (if one exists).* Otherwise it is undefined.
- `IJ`—This is the ImageJ utility object. It is always available.
- `Editor`—This is a reference to the JavaScript_Editor object itself. Most of the instance variables and methods in this object are public and therefore visible to JavaScript. This means you can manipulate the editor environment at runtime (change the font, add buttons to the editor frame, etc.) with JavaScript, if desired.

Examples of using these objects are given in the scripts in your **\js\Examples** folder (and also in the tutorials below).

Once your evaluation is complete (that is, once a message appears in the Results window), your script—and indeed the entire interpreter—goes out of scope. No variables persist across Evaluate commands.

## 3.3 Asynchronous Operation

When you evalaute a script, the evaluator is executed in its own thread so that lengthy JavaScript operations will not cause ImageJ to block. It is up to you, however, to incorporate user-abort capability into your scripts, so that long-running scripts can be interrupted. (An example of how to do this is shown in the **includes.js** file.)

## 3.4 Calling Java from JavaScript

You can instantiate and/or call methods of any Java object using JavaScript. To do this merely requires that you prepend a Packages context string to the front of the object name. For example, to create a Java Vector object in JavaScript, you can do:

```
var vect = new Packages.java.util.Vector();
```

You can then call **addElement()** and other methods of the Vector class, on the JavaScript variable "vect."

Of course, you can also call ImageJ methods. For example, you can obtain the ImagePlus object for the current (frontmost) image window by executing:

```
frontWindow = Packages.ij.WindowManager.getCurrentImage();
```

You can then call `ImagePlus` methods, such as `getProcessor()`, on the `frontWindow` object.

**NOTE** You will normally not have to write the above line of code, because the ImagePlus object for the current (frontmost) image is available in the JavaScript global object called `ImagePlus`, which is created by the JavaScript_Editor plug-in at evaluation time.

In general, the Rhino interpreter handles the conversion of JavaScript objects to Java objects (and vice versa) seamlessly. Arrays, however, are not directly interconvertible. You can access a Java array directly, from JavaScript, but you cannot pass a JavaScript array to a Java method that requires a Java array. So there will be times when you will find it necessary to declare a Java array from JavaScript. This can be done in the following manner:

```
var row =
new java.lang.reflect.Array.newInstance(java.lang.Integer.TYPE,
320);
```

This line declares an array of “int”, of length 320. You can pass the “row” variable (in this example) to the `getRow()` method of the `ImageProcessor` class.

The Rhino implementation allows you to implement Java interfaces (such as listeners); an example of this is shown in the example script called **inverSelectedtRGBChannel.js**, where the Java Enumeration interface is implemented in order to create a pixel enumerator.

## 3.5 Performance Considerations

JavaScript is an interpreted language (as is Java), which means execution speed can be an issue at times, particularly where large images are concerned. The following “best practices” will help you achieve maximum performance.

- Utilize native ImageJ API methods whenever possible. For example, if you need to convolve an image with a 3x3 matrix, use the `convolve3x3()` method of the ImageProcessor class rather than write your own JavaScript convolver.
- Utilize native JavaScript methods whenever possible. For example, use the built-in `reverse()` method of the JavaScript Array object rather than loop over an array of pixels yourself. Core-language JavaScript methods are implemented in highly optimized Java. By using JavaScript Array methods like `slice(), concat(), reverse(), join(),` etc., and String methods `replace(), substring(),` etc. (plus RegExp methods where applicable), you can execute much of your script in optimized bytecode. For an example of this, see the **rotateImage.js** example file, which uses the JavaScript `reverse()` method to rotate an image 180 degrees.
- Consider moving “expensive” operations off to custom Java code that can be called from JavaScript.
- Use lookup-table strategies wherever possible.
- Avoid the use of the top-level core JavaScript method `eval()`, and do not put RegExp declarations inside of loops.

It’s important to incorporate visual feedback in lengthy operations. There are two ways to do this (and both techniques are illustrated in the example scripts that accompany this plug-in): One is to use the ImageJ progress bar to give a visual hint as to the current operation’s progress. The other is to use an incremental-redraw approach so that the image updates frequently.

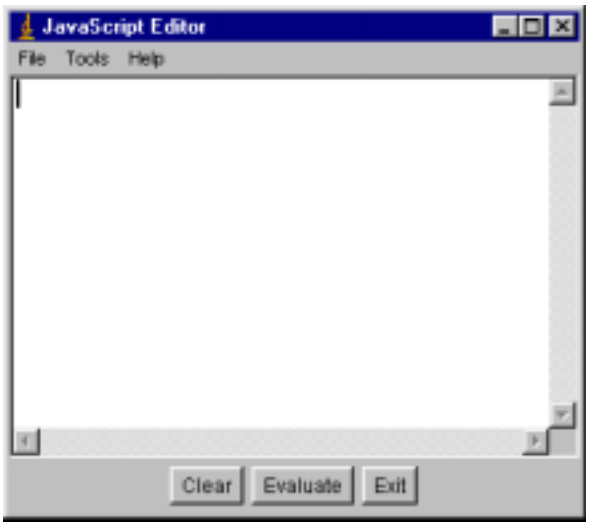
It’s also important to be able to “bail out of” long-running operations prematurely by letting the user type a special key or through some other gesture. One approach (involving the Alt key) is shown in the `RGBImage` routine in **includes.js**.

# 4.0 Tutorials

This section will lead you through a couple of brief scripting sessions with the JavaScript_Editor plug-in to acquaint you with its basic functionality.

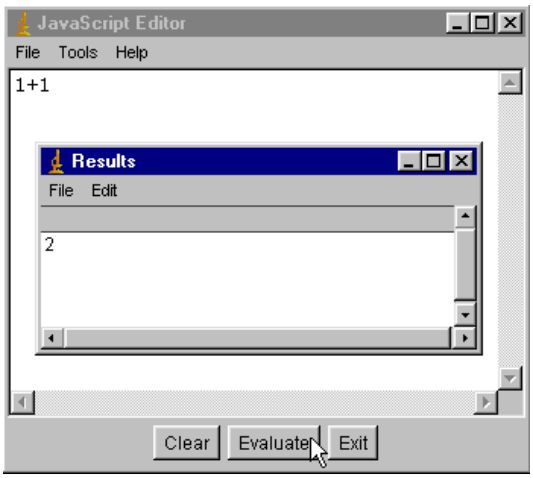
## 4.1 Using the Script Editor GUI

If this is the first time you are using the plug-in, launch ImageJ and select **Compile and Run** from the **Plugins** menu. Use the navigator dialog to find the *JavaScript_Editor.java* file and select it. After a few seconds, the editor window should appear.




If the plug-in does not compile, check your classpath and path environment variables to be sure the **js.jar** file is in your classpath. (This is the Mozilla Rhino jar, as explained earlier.) If you are launching ImageJ using the *ImageJ.exe* file on Windows, edit the *ImageJ.lax* file in a text editor as necessary to set the classpath. Be sure other ImageJ plug-ins can be compiled and run normally.

Type 1+1 in the editor window, then click the **Evaluate** button at the bottom of the window. The ImageJ *Results window* should appear (perhaps in front of the script editor window as shown here):



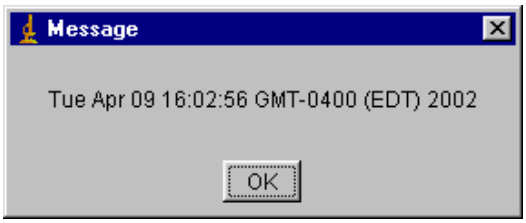

The result of the evaluation is shown in the Results window as 2.

Click the **Clear** button to clear the window contents.

Now type the following two lines of code:

```
today = new Date();
IJ.showMessage(today);
```

Execute the two lines by clicking **Evaluate** (or by typing **Control-E**). You should see a small dialog window appear, similar to:




The `IJ` object is a native ImageJ object. It is visible as a *global object* in JavaScript due to the fact that the **JavaScript_Editor.java** file contains a routine that *reflects* this object into the JavaScript runtime scope. Two other ImageJ objects are similarly reflected: `ImageProcessor` (representing the processor object for the current image) and `ImagePlus` (representing the associated ImagePlus object). To see how this works, first open an image. (For this exercise, it should be an 8-bit greyscale or RGB color image.) Then clear the editor window and type the following lines of code:

```
ImageProcessor.invert();
ImagePlus.updateAndDraw();
```

When you hit Evaluate, the image will invert colors and appear as a "negative image." Click the Evaluate button repeatedly and watch the image toggle between normal and inverted appearance.

Rather than Clear the window, go up to the **File** menu and choose **Save As**. A dialog will appear, allowing you to save your script window contents. Choose any folder you want, then type **myInverter.js** for the name of the file and click OK. Your script is saved to disk as a text file for use later.

**NOTE** You can save your script files with any extension you want, in any location you want. It is convenient and customary, however, to use the ".js" extension for JavaScript files.

With **myInverter.js** still open (use Open to find and open it, if need be), go to the script editor menubar and choose **Convert to Plugin** from the **Tools** menu. Within a second or two, you will see a dialog similar to this:

This means that your plugins folder now contains a small file called **myInverter_.java,** which, when compiled, will become an ImageJ plug-in. The compiled plug-in, in turn, will call your script for you whenever you run the plug-in from the ImageJ **Plugins** menu. (Note: Your script code does not get compiled into native Java: It remains JavaScript, and it continues to exist as **myInverter.js**.) What is happening here is that a small Java "trigger" class is being created. The trigger, which is named after your script file, is designed to run your script silently, without your needing to bring up the JavaScript_Editor window.

Use the **Compile and Run** command in ImageJ's **Plugins** menu to compile the file called **myInverter_.java** (which will be in your ImageJ plug-ins folder). The next time you launch ImageJ, the plug-in called **myInverter** will appear in the list of plug-ins in ImageJ's **Plugins** menu.

**NOTE** The **Convert to Plugin** command does not automatically compile your plug-in. To make it show up in the ImageJ **Plugins** menu, you must compile the Java file generated by **Convert to Plugin** and then relaunch ImageJ.

## 4.2 Working with Pixel Arrays

Direct, low-level pixel manipulations are, of course, at the heart of graphics programming, so it is vital that you know how to access pixels via JavaScript. The example scripts that come with the JavaScript_Editor package contain many examples of how to access pixel arrays in RGB images. But you can also access the pixel arrays of other image types. The following example shows how to work with `byte[]` arrays in 8-bit greyscale images.

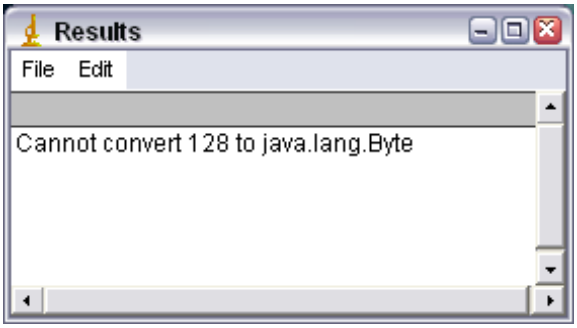
First, use ImageJ to open an 8-bit greyscale image (or open an RGB image and convert it to greyscale).

Next, launch the JavaScript_Editor window and type the following code:

```
pixels = ImageProcessor.getPixels(); // grab pixel array
for (var i = 0; i < pixels.length; i++)
{
   p = pixels[i];
   p &= 0xe0;
   pixels[i] = p;
}
ImagePlus.updateAndDraw();
```

The idea behind this code is that we want to AND each pixel against the hex value `0xe0`. In other words, we are masking the pixel against a hex value that represents binary `11100000`. When we do that, we are throwing away the bottom 5 bits of precision in the 8-bit pixel value.

There's only one problem. When we execute this code, we get an error message in the Results window and no change in the original image. The error is:




The error occurs because we are trying to assign an 8-bit *unsigned* value to a `java.lang.Byte`, whereas the `byte` type is *signed*. Java will not do the appropriate type cast for us. We must do it ourselves.

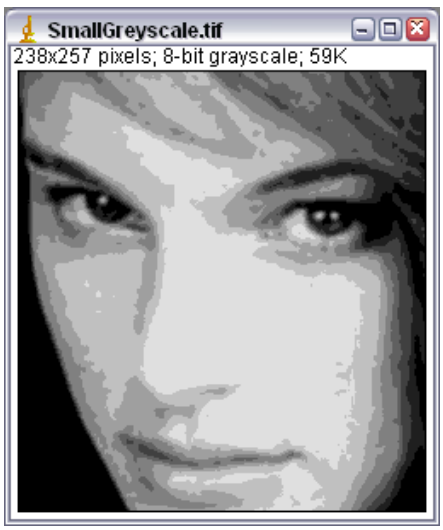
To correct the error, make the following change to the last line of code inside the loop:

```
pixels[i] = (new java.lang.Integer(p)).byteValue();
```

Notice that we are converting our unsigned pixel value to a `java.lang.Integer` object, then we obtain the byte value of that Integer by a call to `byteValue()`.

After making this change, click the Evaluate button once again so as to run the script against the open image. When you execute the code, you will see the image go from 256 shades of grey to 8 shades of grey:

256 shades of grey



8 shades of grey

On a 1GHz Pentium-III computer, this transformation occurs at a rather leisurely rate of 18,000 pixels per second. The bottleneck is the repeated instantiation of the `java.lang.Integer` object. The choke point, in other words, is Java, not JavaScript.

In the next example, we see how to do a much more elaborate image transformation, using an RGB image pixel array, with performance *more than twice as fast* as the above transformation.

## 4.3 Advanced RGB Image Transformation

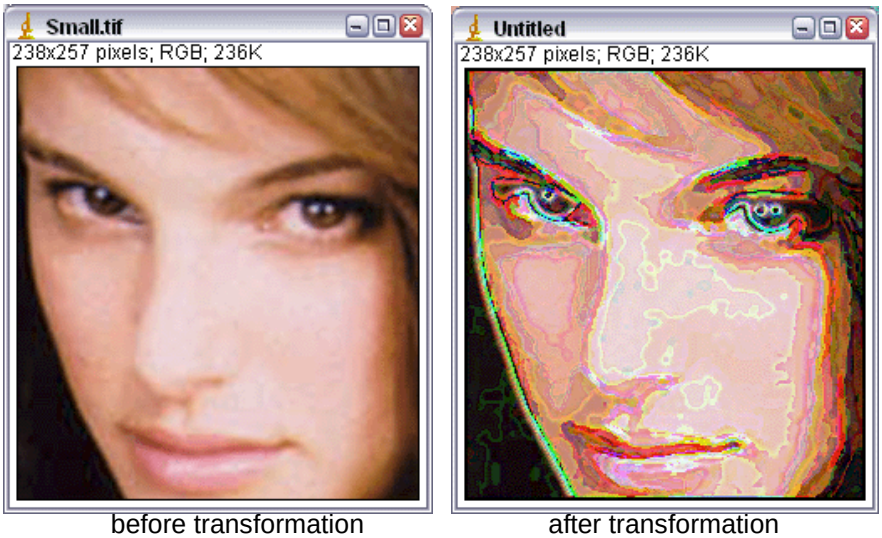
Using the script editor, open the example file called **posterStroke.js**. The code in that file is repeated here for your convenience:

```
// =============== posterStroke() ====================
function posterStroke(ip,imp,mask){

ip.snapshot();             // cache a copy of original
img = new RGBImage(ip);    // make a copy in a new window

pixels = ip.getPixels();   // grab pixel array
length = pixels.length;    // cache this for loop speed
ip.smooth();  // mild blur
ip.smooth();

for (var i = 0; i < length; i++)
{
    pixels[i] &= mask;     // quantize
    if (i%6000==0)         // show a progress bar
          IJ.showProgress(i/length);
}

ip.findEdges();            // get contours
ip.smooth();               // smooth them a bit

img.imageProcessor.copyBits(ip,0,0,11); // 11 == XOR mode
ip.reset();  // restore original
img.imageProcessor.copyBits(ip,0,0,7); // 7 == AVERAGE mode
img.imagePlus.updateAndDraw();

ip.reset();  // restore original
}

// ====================================================
// Exercise it

start = 1*new Date;
posterStroke(ImageProcessor,ImagePlus,0xf0f0ff);
interval = ((1*new Date)-start)/1000;
IJ.write( length/interval + " pix/sec");
```

The `posterStroke` method takes three arguments: an `ImageProcessor` object, an `ImagePlus` object, and a mask value.

**NOTE** Notice, incidentally, in all these examples, that the JavaScript interpreter takes care of casting numbers from double to int and vice versa. In JavaScript, a number, by default, is a IEEE-754 double-precision floating point number. But it will be cast to an integer when the occasion calls for it, usually without any special action by the programmer. On those rare occasions when you need to perform the cast yourself, simply use `Math.floor()` on your numeric variable.

There's a lot going on in this code, but the essence of it is that the original image is copied, blurred, and quantized (against the mask value), then the ImageJ `findEdges()` method is used to get the contours of the image. The contours are then blurred and XOR'd against the original image. Finally, the result of that operation is averaged with the original image. The result of all these machinations is a transformation of the original image into a highly stylized, somewhat abstract final image with unusual aesthetic qualities:




before transformation  after transformation

The interesting thing about this code is that it runs at about 40K pixels per second (on a 1GHz Pentium III), versus only 18K pixels/sec for the greyscale transformation discussed in the previous section. The bottleneck this time is our `for`-loop. JavaScript loops are, in general, expensive. We can speed the routine up about 20% by eliminating the progress-bar operation. (But it's usually worth having some sort of visual feedback of progress on a potentially lengthy JavaScript image transformation of this sort.)

It turns out that if we eliminate the `for` loop altogether (removing the quantization step), we can achieve a very similar visual end-result, but with *ten times greater speed* (600K pixels/sec). This is because the entire transformation occurs in bytecode; native ImageJ routines to do the heavy lifting.

The moral should be obvious: If you are interested in obtaining maximal performance, you should let native ImageJ API calls do as much of the work as possible. You should also be careful about doing a lot of work inside JavaScript for-loops, especially nested loops. And if you *do* need to do potentially slow operations, provide some visual feedback in a progress bar or by using an incremental-redraw strategy.

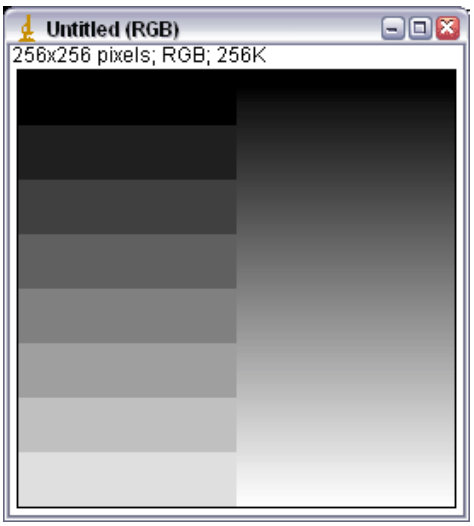
## 4.4 Programmatic Construction of Images

The example script called greyGradient.js shows how to use the `RGBImage` JavaScript object (a custom object defined in **includes.js**) to create a new, empty image on the fly and fill it with a greyscale ramp. We want the left half of the image to show eight levels of grey and the right half to show the full spectrum of greys (0..255). Here is the code:

```
img = new RGBImage(256,256); // create a blank image
k = 0;   // declare and initialize a counter

// use an anonymous callback in pixelIterate():
img.pixelIterate( function(p) {
  k = p.y;
  k = (k << 16) + (k << 8) + k;

  return (p.x < img.width/2) ?
       k & 0xe0e0e0 : k;
 }
);
```

Our convenience object, the `RGBImage` object defined in **includes.js**, has a method called `pixelIterate()`, which contains a for-loop that iterates over all the pixels in the image, calling a user-defined callback for each pixel. The `pixelIterate()` method takes exactly one argument, which is some arbitrary pixel-processing function that you define. The callback will be passed a custom object containing various fields preinitialized to values that would typically be of interest to such a callback: e.g., the red, green, and blue values of the current pixel, its x- and y-coordinates in the image, and so on. (See the includes.js source code for details.)

In this example, we pass `pixelIterate()` an anonymous function that generates pixel values on the fly based on pixel x- and y-positions. We need the y-coordinate of the current pixel so that we can set the RGB value to the y-value (across all color channels equally). Thus, pixels will go from pure white at the bottom of the image to pure black at the top. If the pixel's x-position is less than the image width divided by two, we quantize the result such that the pixel can take on one of only eight possible grey values. This makes for a quantized, 3-bit-bandwidth ramp on the left side of the image, and a full-bandwidth ramp on the right side. The final effect is:



## 4.5 Instantiating and Working with Java Objects

You can call Java methods from JavaScript quite readily. This simple exercise shows how.

Run the JavaScript_Editor plug-in, and in the script editor window, type the following code (or open the example file called javaAWTFrame.js):

```
Frame = java.awt.Frame;
Button = java.awt.Button;
Label = java.awt.Label;
Dimension = java.awt.Dimension;
Panel = java.awt.Panel;

frame = new Frame("My New Frame");
frame.setSize(new Dimension(240,100));
label = new Label("This is a message.",1);
button = new Button("OK");
panel = new Panel();

frame.add(label);
frame.add(button,java.awt.BorderLayout.SOUTH);

ok = function dismiss(e){
e.getSource().getParent().dispose(); }
act = { actionPerformed: ok }
listener = new java.awt.event.ActionListener( act );
button.addActionListener(listener);
frame.show();
```

If you are familiar with AWT programming, you will find this block of code very straightforward. We are simply constructing a Frame containing a label and a button. The button is associated with an ActionListener, with an `actionPerformed` method that points to the JavaScript function `ok()`. When you evaluate the above code, a new window will appear onscreen, looking like:


If you click the OK button, the ActionListener's `actionPerformed()` code will be invoked and the dialog will go away.

Several more examples of calling Java from JavaScript are shown in the sample scripts that come with the JavaScript_Editor distribution. For examples of how to use the advanced Java/JavaScript bridging facilities of Rhino, see the online documentation at the **www.mozilla.org/rhino** web site.

# 5.0 Source Code

The plug-in's source code (as of April 10, 2002) is reproduced further below. (Check **http://www.acrojs.com** for updates.) It comprises about 600 lines of Java, of which 100 or so are for the PrintUtility class, which implements the Print command on the File menu. Most of the rest of the code is devoted to implementing the editor and its GUI features. The ScriptRunner class, which actually implements the ImageJ-to-Rhino binding, is only 35 lines long.

The code is structured into four classes:

- **JavaScript_Editor**—This is the class that constructs and manages the script editor window. It extends `PlugInFrame`.
- **MenuBuilder**—This inner class (within JavaScript_Editor) is a helper class for building menus from property strings.
- **PrintUtility**—This outer class implements the Print command functionality of the editor.
- **ScriptRunner**—This inner class calls Rhino objects in order to run scripts. It extends `Thread`. It eventually needs to be an outer class.

## 5.1 To Do

The JavaScript_Editor plug-in is a labor of love and was written primarily for the author's own selfish graphics-programming needs, and the GUI reflects this. There are many features left to do. If spare time permits, I'll start going down the wish-list, but frankly, updates to the script editor GUI are not likely to get done any time soon.

Nevertheless, some features that I would like to incorporate (eventually) are enumerated below. Code contributions based on these or other useful features are invited!

- Caching of the names of the most recently visited files, and command based on those names at the bottom of the **File** menu or in an **Open Recent** submenu.
- A Java-native *pixel iterator class* (that can be hooked into from JavaScript) needs to be implemented, for better script performance. Loop overhead is very costly in JavaScript.
- Rhino, interestingly, comes with a true JavaScript-to-Java compiler class. It would be exciting and useful to offer the capability of compiling user scripts *directly to bytecode*.
- A **Replace** command is needed for basic editing.
- The **Find** command should allow regular expressions. This can be done fairly easily by harnessing JavaScript's powerful RegExp object.
- Typical programmers-editor capabilities like clip libraries, code completion, syntax coloring, etc., need to be incorporated in the script editor. Some of these features will be impossible without the use of Swing classes.
- A context-sensitive right-mouse menu is needed for clicks inside the editor's text area.
- Line and column numbers should be reported in real time in a status bar. A **Go to Line Number** command would then be appropriate and would help greatly with debugging.
- Some kind of online help would be good. (Not Sun's JavaHelp, though!)

- Additional Properties need to be used for persisting user preferences as to window size, typeface, font size, font style, etc.
- There should be a mode of operation in which the JavaScript global scope is maintained across script executions. The user should be able to toggle that mode from a menu command. This would allow user variables to maintain their values across calls to the evaluator (helpful for debugging and experimentation).
- `ScriptRunner` should be an external class visible to all plug-ins and all ImageJ objects.

```
// * * * * * JavaScript_Editor 1.0 by Kas Thomas * * * * *
//
// 10 April 2002
//
//
// LEGAL
//
// The following code is freely distributable but remains the
// intellectual property of the author and may not be used
// commercially nor in an enterprise setting without prior
// approval. This code is for personal, non-job-related use
// only. You agree to use it at your own risk.
//
//            Copyright 2002 by Kas Thomas.
//    Commercial or enterprise use prohibited without
//         the express permission of the author.
//
// See also the license terms of Rhino (at URL below),
// which apply separately.
//
//
// OBTAINING RHINO
//
// Please note that this ImageJ plugin WILL NOT WORK unless
// you have the Mozilla/Rhino js.jar file in your classpath.
// (Note the "import org.mozilla.javascript.*" statement below.)
// The only Rhino jar you need is the one called js.jar.
// That file comes with the Mozilla Rhino (JavaScript interpreter)
// package. You can download that package free from:
//
//           http://www.mozilla.org/rhino/
//
// See that URL also if you have Rhino support questions.
//
// No support, as such, is offered for this plugin.
// See www.acrojs.com for latest version, info, etc.
// Write to the author at: kthomas@acrojs.com.


import ij.*;
import ij.process.*;
import ij.gui.*;
import java.awt.*;
import java.io.*;
import java.util.*;
import ij.plugin.frame.*;
import java.awt.event.*;
import org.mozilla.javascript.*;

public class JavaScript_Editor extends PlugInFrame
{

// A lot of our instance variables are public so that they
// can be seen from JavaScript.

    public static final String pluginPath = Menus.getPlugInsPath();
    public String directory = Menus.getPlugInsPath(); // reasonable default
```

```
public String path = "";  // used by open() and saveAs()
private String originalFileContentsOnOpen = ""; // for Revert to Open
public String nameOfCurrentFile = "";
private boolean savedOnce = false;
public String targetString = "";
public TextArea ta=null;
String wholeWindowTextCache = "";  // for Revert to Save
public Properties props = new Properties();

// This is the one and only constructor...        * * * CONSTRUCTOR * * *
public JavaScript_Editor()  throws Exception
{
    super("JavaScript Editor");
    try
    {
        FileInputStream in = new FileInputStream(pluginPath +
                    "js/jsProperties.properties");
        props.load(in);
        in.close();
    }
    catch( FileNotFoundException e)
    {
        IJ.showMessage("Could not find " + e.getMessage() +
              ".\nPlease be sure this file exists.");
        throw new Exception();
    }

    buildAllMenus();
    setupEditWindow();
}

// ====================== buildAllMenus() ========================
// This method calls on an inner class (below) to help with menu-building.
public void buildAllMenus()
{
    String[] menus = { "File","Tools","Help" };
    MenuBuilder builder = new MenuBuilder(props); // see further below
    MenuBar mb = new MenuBar();
    for (int i = 0; i < menus.length; i++)
    {
        Menu menu = builder.buildMenu(menus[i]);
        mb.add(menu);
    }
    setMenuBar(mb);
}

// ====================== setupEditWindow() ========================
// Set up the editor environment.
private void setupEditWindow()
{
    int rows = Integer.parseInt(props.getProperty("WindowPrefs.rows") );
    int cols =  Integer.parseInt(props.getProperty("WindowPrefs.columns") );
    ta = new TextArea(rows,cols);

    ta.setFont(new Font("Monospaced",
  Integer.parseInt(props.getProperty("FontPrefs.fontStyle")),
        Integer.parseInt(props.getProperty("FontPrefs.fontSize")) )
```

```
        );
    setBackground(Color.lightGray );

    // create the 3 Buttons and their listeners...
    Button evalButton = new Button("Evaluate");
    Button clearButton = new Button("Clear");
    Button exitButton = new Button("Exit");

    evalButton.addActionListener(
        new ActionListener()  {
            public void actionPerformed (ActionEvent e)
            {
                String script =  ta.getSelectedText();
                if (script.equals(""))
                    script  = ta.getText();
                evaluate(script);
            }
        }
    );

    clearButton.addActionListener(
        new ActionListener() {
            public void actionPerformed (ActionEvent e)
            {
                ta.setText("");
                setTitle("Untitled");
            }
        }
    );

    exitButton.addActionListener(
        new ActionListener() {
            public void actionPerformed (ActionEvent e)
            {
                exitEditor();
            }
        }
    );

    // create button panel
    Panel buttonPanel = new Panel();
    buttonPanel.add(clearButton);
    buttonPanel.add(evalButton);
    buttonPanel.add(exitButton);

    // add panels to window
    add(ta);
    add(buttonPanel, BorderLayout.SOUTH);
    pack();

    GUI.center(this);
    setVisible(true);
}
```

```
// ====================== evalThisTextArea() ========================
public void evaluate( )
{
   String windowContent = ta.getText();
   ScriptRunner sr =  new ScriptRunner( windowContent );
}

// ========================== evaluate() ============================
// Evaluate (i.e., run) the script passed in 'str'.
public void evaluate(String str)
{
  ScriptRunner sr = new ScriptRunner(str);
}

// ======================== convert() ========================
// Create an ImageJ plugin (.java) that drives the current script
public void convert()
{
    evaluate("scriptToPlugin");
}

// ====================== revertToSave() ========================
public void revertToSave()
{
    ta.setText(wholeWindowTextCache);
}

// ====================== revertToOpen() ========================
public void revertToOpen()
{
    ta.setText(originalFileContentsOnOpen);
}

// ========================== open() ============================
public void open()
{
    if (checkForDirtyWindow() ==false) // meaning, it's not okay to continue
         return;
    try
    {
        savedOnce = false;
        FileDialog f = new FileDialog(this,
                                      "Open File", FileDialog.LOAD);
        f.setDirectory(directory);
        f.show();
        directory = f.getDirectory();  // remember dir
        String fileName  =nameOfCurrentFile = f.getFile();
        if (fileName == null) return;
        path = directory + nameOfCurrentFile;
        String contents = "";
        contents =
        originalFileContentsOnOpen =
        wholeWindowTextCache =
                     readAll(directory,fileName);
        ta.setText(contents);
        setTitle("JavaScript Editor: " + fileName);
    }
```

```
    catch (Exception e)
    {
        IJ.error("File Open Error. " + e.getMessage());
        return;
    }
}// open

// ====================== saveAs() ========================
// Save As uses a  dialog . . .
public boolean saveAs()
{
    FileDialog fd = new FileDialog(this, "Save As...", FileDialog.SAVE);
    String name1 = nameOfCurrentFile;
    fd.setFile(name1);
    fd.setDirectory(directory);
    fd.setVisible(true);
    String name2 = fd.getFile();
    String dir = fd.getDirectory();
    fd.dispose();
    savedOnce = true;
    if (name2!=null)
    {
        path = dir+name2;
        save(path);           // call save() now
        setTitle(name2);
        nameOfCurrentFile = name2;
        return true;
    }
    return false;
}

// ==================== saveToCurrentPath() ======================
public void saveToCurrentPath()
{
    if (path.equals("")) saveAs();
    else save(path);
}

// ====================== save() ========================
// Save current window contents without popping a  dialog . . .
public void save(String outpath)
{
    String s =wholeWindowTextCache =  ta.getText();
    try
    {
        BufferedWriter bw = new BufferedWriter(new FileWriter(outpath));
        if (bw != null)  bw.write(s, 0, s.length());
        bw.close();
    }
    catch
        (IOException e)
    {
        IJ.showMessage("Error on Save As! " +
              e.getMessage());e.printStackTrace();
    }
}
```

```
 // ====================== saveStringToFile() =========================
 public void saveStringToFile(String content,String outpath)
 {
     try
     {
         BufferedWriter bw = new BufferedWriter(new FileWriter(outpath));
         if (bw != null)  bw.write(content, 0, content.length());
         bw.close();
     }
     catch
         (IOException e)
     {
         IJ.showMessage("Error on Save As! " +
               e.getMessage());e.printStackTrace();
     }
 }

 // ====================== closeFile() =========================
 public void closeFile()
 {
 if (checkForDirtyWindow() == true) // meaning, we're going to close out
 {
     // zero out a few things
     ta.setText("");
     originalFileContentsOnOpen = nameOfCurrentFile = "";
     setTitle("Untitled");
  }
 }

// ================checkForDirtyWindow() ==================
// A return value of true means it is okay to close out.
public boolean checkForDirtyWindow()
{
      if (wholeWindowTextCache.equals(ta.getText())) // no changes
            return true;

         YesNoCancelDialog answer =
             new YesNoCancelDialog(this,
                                   "Save?",
                                   "Save File Before Closing?") ;

         if (answer.cancelPressed())
                return true;

         else   if (answer.yesPressed())
         {
             boolean cancel  =  !saveAs();
             if (cancel) return false;
         }
        return true;
}
```

```
// ======================== print() ============================
// This calls on a helper class further below.
public void print()
{
    Font font = new Font(ta.getFont().getName(),
  ta.getFont().getStyle(),
  ta.getFont().getSize()-4);
    new PrintUtility( this,ta.getText(), font );
}

// ====================== exitEditor() ========================
// Quit and go back to ImageJ.
public void exitEditor()
{
    if (checkForDirtyWindow() == true)
        dispose();
}

// ====================== readAll() ========================
// Read a file into a big string, and close the file.
public  String readAll(String  dir, String fname )
{
    File  file = new File(dir,fname);
    String bigString = new String();
    try
    {
        StringBuffer sb = new StringBuffer(5000);
        BufferedReader r = new BufferedReader(new FileReader(file));

        while (true)
        {
            String s=r.readLine();
            if (s==null)
                break;
            else bigString+=s + "\n";
        }
        r.close();
    }
    catch
        (IOException e)
    {
        IJ.showMessage("Could not read file. " + e.getMessage());return "";
    }
    return bigString;
}

// ====================== performReflections() ========================
// Expose some custom objects for use as top-level JavaScript objects.
void performReflections(Scriptable scope)
{
    Scriptable jsArgs;
    try
    {
        ImagePlus imp = WindowManager.getCurrentImage();
        ImageProcessor ip = imp.getProcessor();

        // expose ImagePlus for current image window
```

```
        jsArgs = Context.toObject(imp, scope);
        scope.put("ImagePlus",scope,jsArgs);

        // expose ImageProcessor for current image
        jsArgs = Context.toObject(ip, scope);
        scope.put("ImageProcessor", scope, jsArgs);
    }
    catch(Exception e)  {;}  // Don't bail just because an image wasn't open

    // create Editor object for easy script access to this class!
    jsArgs = Context.toObject(this, scope);
    scope.put("Editor", scope, jsArgs);

    // expose IJ utility class
    jsArgs = Context.toObject(new IJ(), scope);
    scope.put("IJ", scope, jsArgs);
}

// ====================== find() ========================
// Provide a decent Find facility.
public void find()
{
    targetString = IJ.getString("Search for:",
                                ta.getSelectedText().toLowerCase());
    if (targetString.equals("")) return; // sanity
    int cursorPos = ta.getCaretPosition();
    String theText = ta.getText().substring(cursorPos);

    int firstHit = theText.toLowerCase().indexOf(targetString);
    if (firstHit == -1)
        firstHit = theText.indexOf(targetString); // yes, this has a purpose
    if (firstHit != -1)
    {
        ta.setCaretPosition(cursorPos +  firstHit + targetString.length());
        ta.setSelectionStart(cursorPos +  firstHit );
        ta.setSelectionEnd(cursorPos +  firstHit + targetString.length());
    }
    else IJ.showMessage( "No matches found.");
    toFront();
}

// ====================== findAgain() ========================
// Provide a decent Find Again facility.
// (Todo: Manage menu item's enable state.)
public void findAgain()
{
    if (targetString.equals("")) return; // sanity

    int cursorPos = ta.getCaretPosition()+1;
    String theText = ta.getText().substring(cursorPos);

    int nextHit = theText.toLowerCase().indexOf(targetString);
    if (nextHit == -1)
        nextHit = theText.indexOf(targetString);
    if (nextHit != -1)
    {
        ta.setCaretPosition(cursorPos +  nextHit + targetString.length());
```

```
        ta.setSelectionStart(cursorPos +  nextHit );
        ta.setSelectionEnd(cursorPos +  nextHit + targetString.length());
    }
    else IJ.showMessage( "No more matches found.");
    toFront();
}

// ======================= showAbout() ========================
// Please don't remove this.
public void showAbout()
{
    IJ.showMessage("JavaScript_Editor 1.0.1 by Kas Thomas.\n"+
                   " \n "+
                   "A plugin to provide interactive image editing\n"+
                   "via JavaScript, using the Mozilla Rhino engine.\n"+
                   " \n"+
                   "Copyright 2002 by Kas Thomas.\n"+
                   "Contact: kthomas@acrojs.com");
}

// ================================================================
// * * * * * * * * * * INNER CLASS: MenuBuilder * * * * * * * * * *
//
// This builds menus from Property strings, which makes it easy
// to add or rearrange/modify menus, hotkey assignments, and
// command-to-action-function bindings just by editing a text file.
// ================================================================
public class MenuBuilder
{
    private Properties props = null;

    public MenuBuilder(Properties proplist) // * * * CONSTRUCTOR * * *
    {
        props = proplist; // get top-class instance variable 'proplist'
    }

    // Try to find the property 'menuName' and build a Menu
    // based on its value...

    public Menu buildMenu(String menuName)
    {
        Enumeration enum = props.propertyNames();
        Menu menu=null;

        while(enum.hasMoreElements() ) // check every property
        {
            String theProp = enum.nextElement().toString();
            String value = props.getProperty(theProp);

            if (theProp.indexOf("Menu")==0
                    && theProp.indexOf(menuName)!=-1)
             {
                menu = new Menu( theProp.substring(
                    theProp.lastIndexOf(".")+1) );

                // The '*' delimiter demarcataes
                // MenuItemName|hotkey|callback strings:
```

```
                StringTokenizer st = new StringTokenizer(value,"*");

                while (st.hasMoreTokens()) // parse menu commands
                {
                    MenuItem mi;

                    String theElement = st.nextElement().toString();

                    // The '|' delimiter separates name,hotkey,action
                    int index = theElement.indexOf("|");
                    if (index != -1)
                    {
                        String theCommandName = theElement.substring(0,
                                                                index);
                        String theShortcutName =
                                theElement.substring(index+1,index+2);
                        MenuShortcut hotkey =
                            theShortcutName.equals(" ") ? null :
                                 new MenuShortcut(theShortcutName.charAt(0));
                        mi = new MenuItem( theCommandName, hotkey );
                        int theCommandIndex = theElement.lastIndexOf("|");
                        final String theMethodName =
                              theElement.substring(theCommandIndex+1);

                        // The actionListener (below) will execute the appropriate
                        // action method for the menu item, using JavaScript
                        // reflection. In general, trampolining back and forth
                        // between JavaScript and Java this way is not a best
                        // practice. But doing it this way takes only one line of code.

                        mi.addActionListener(
                           new ActionListener() {
                                public void actionPerformed(ActionEvent event)
                                {
                                    evaluate(theMethodName); // trampoline into JS
                                }
                           }
                        );
                    }
                    else // No pipe character?
                    {
                        String theCommandName = theElement;
                        mi = new MenuItem( theCommandName );
                    }
                    menu.add(mi);  // add the MenuItem

                }// while st (get all commands)

            } // if Menu  (build this menu)

        }  //while props (search all properties)

        return menu;  // returns null menu if not found

    } //method

}  // inner class MenuBuilder
```

```
    // ================================================================
    // * * * * * * * * * * CLASS: ScriptRunner  * * * * * * * * * *
    //
    // This class allows us to run a script in its own thread, so that a lengthy
    // operation won't lock us out of the program.
    // ================================================================

      class ScriptRunner extends Thread
      {
       public String str;

       ScriptRunner( String s ) {
         super("Interpret");
        this.str = s;
       start();
        }

      public void run(){
       String resultString = "";
       try {
            Context cx = Context.enter();
            Scriptable scope = cx.initStandardObjects(null);
            String includes = readAll(pluginPath + "js/","includes.js");
            str = "\n" + includes + "\n" + str;
            performReflections(scope);
            try
            {
                Object result = cx.evaluateString(scope,
                                                  str,
                                                  "<cmd>",
                                                  1, null);
                resultString = cx.toString(result);
                IJ.write(resultString );
            }
            catch( JavaScriptException jse)
            {
                IJ.write("JavaScript exception: " + jse.getMessage()) ;
            }
       } catch( Exception e) { IJ.write(e.getMessage());}

       } // run()
      }

}// end class JavaScript_Editor
```

```
// ===================================================================
// * * * * * * * * * * CLASS: PrintUtility * * * * * * * * * *
//
// Out of haste, the code for this class was adapted from the
// ij.plugin.frame.Editor print() routines.
// ===================================================================

class PrintUtility {
  PrintJob pjob = null;
  Graphics pg =null;
  Font theFont;

    public PrintUtility(Frame frame, String theText, Font font) {
  pjob = frame.getToolkit().getPrintJob(frame, "JS Job",new Properties());
  pg = pjob.getGraphics( );
  theFont = font;
  printString(theText);
              pg.dispose( );
  pjob.end( );
    }

    void printString (String s) {
        int pageNum = 1;
        int linesForThisPage = 0;
        int linesForThisJob = 0;
        int topMargin = 50;
        int leftMargin = 50;
        int bottomMargin = 40;

        if (!(pg instanceof PrintGraphics))
            throw new IllegalArgumentException ("Graphics context not PrintGraphics");
        if (IJ.isMacintosh())
            topMargin = leftMargin = bottomMargin = 0;

        StringReader sr = new StringReader (s);
        LineNumberReader lnr = new LineNumberReader (sr);
        int pageHeight = pjob.getPageDimension().height - bottomMargin;
        pg.setFont (theFont );
        FontMetrics fm = pg.getFontMetrics(theFont );
        int fontHeight = fm.getHeight() - 1;
        int fontDescent = fm.getDescent();
        int curHeight = topMargin;
        try {
            String nextLine = "";
            do {
                nextLine = lnr.readLine();
                if (nextLine != null) {
                    nextLine = detabLine(nextLine);
                    if ((curHeight + fontHeight) > pageHeight) {   // page break
                        pageNum++;
                        linesForThisPage = 0;
                        pg.dispose();
                        pg = pjob.getGraphics();
                        if (pg != null)
                            pg.setFont (theFont );
                        curHeight = topMargin;
                    }
```

```
                    curHeight += fontHeight;
                    if (pg != null) {
                        pg.drawString (nextLine,
                                       leftMargin,curHeight - fontDescent);
                        linesForThisPage++;
                        linesForThisJob++;
                    }
                }
            } while (nextLine != null);
        } catch (EOFException eof) {
            // Fine, ignore
        } catch (Throwable t) { // Anything else
            t.printStackTrace();
        }
    }

    String detabLine(String s) {
        if (s.indexOf('\t')<0)
            return s;
        int tabSize = 4;
        StringBuffer sb = new StringBuffer((int)(s.length()*1.25));
        char c;
        for (int i=0; i<s.length(); i++) {
            c = s.charAt(i);
            if (c=='\t') {
                for (int j=0; j<tabSize; j++)
                    sb.append(' ');
            } else
                sb.append(c);
        }
        return sb.toString();
    }

} // end PrintUtility inner class
```