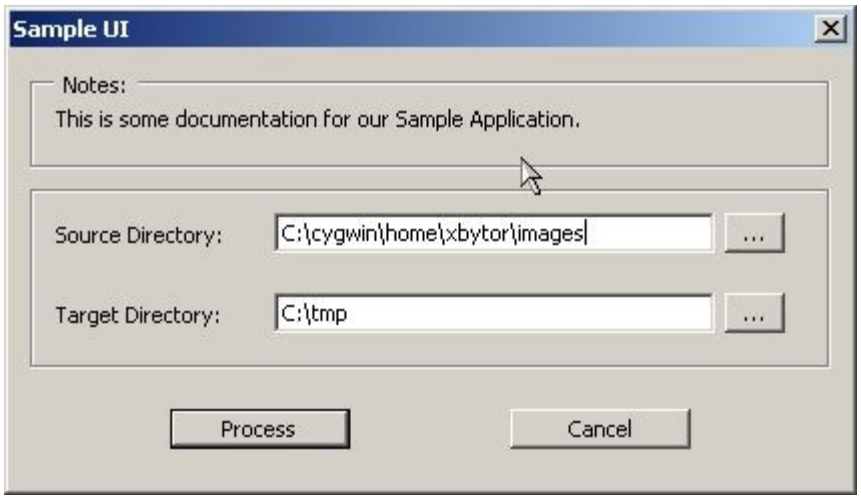
GenericUI

The GenericUI framework makes coding a user interface with ScriptUI easier. It does this by:

- implementing all of that common Window-level code (creation, display, handling of result values)
- automatically creating *Process* and *Cancel* buttons and their associated *onClick* methods.
- standardizing the way UI components are initialized and later validated
- providing an automated mechanism for reading and storing values from the UI in *.ini* files.

To better explain the framework, lets look at the UI created by the sample code presented later in this section.



This window contains two panels. The upper panel is the notes or *documentation panel*. It's intended to contain instructions for using the script. This helps in making the UI self-documenting. The lower panel is the *application panel*. This contains the UI components needed for a particular script.

The two buttons at the bottom are created automatically. Note that *Process* is used instead of *OK*. It turns out that ScriptUI has special features that are enabled if there is a button called *OK*. These features are not consistent between the CS and CS2 implementations. Naming the button *Process* gets around this problem. The return codes from 'show' (0, 1, 2) are hidden, being mapped to either a *process* or *cancel* method invocation.

There are three strings that the UI displays: the labels for the notes panel and the process and cancel buttons. These can be changed using the *notesTxt*, *processTxt*, and properties.

For purposes of this framework, the values collected from the UI that are to be used for processing are called *Options*. Options are typically simple values like strings and numbers. Complex objects and arrays can also be stored in an options object, but it is best to represent them as strings. The reason for this is that if the application can keep the values in the options

object simple, the framework can automatically load and store the options to *.ini* files. Option objects are also referred to as *ini* objects.

Ini objects have a magic property called *noUI*. This will allow the framework to run without actually show the user interface. At first this may seem contradictory or useless, but it turns out to be very helpful. If *noUI* is set to true, the framework will run but instead of gathering the option values from user interface, values are either specified manually in code and/or are read from a *.ini* file as we can see in this bit of code:

```
ui.exec( { noUI: true} );  // use values only from the ini file

var opts = { noUI:true, source: "/c/images/session-dawn", outf: "/c/temp"};
ui.exec(opts);   // use values specified manually. These values override any
                 // read from the .ini file.
```

One place that this is particularly useful is in the debugging of processing code. After the kinks in the user interface are worked out and some reasonable values are in the *.ini* file, the 'noUI: true' setting can be added in the code or in the *.ini* file. The script can then be executed repeatedly without the UI popping up during the writing and/or debugging of the processing code. Another use of this would be to aid in automatic testing of scripts.

From a programming point of view, to use the framework a script must write a UI class that:

- provides appropriate values for the properties that GenericUI specifies (e.g. *documentation*, *winRect*)
- implements the *createPanel*, *validatePanel*, and *process* methods

The sample script at the end of this section gets into the nuts and bolts of how to use this framework.

In the following tables, methods and properties that should normally be overridden by an application script's subclass of GenericUI are marked with a '*'.

Class Functions

Function	Parameter Type	Returns	What it does
getWidgetValue (widget)	ScriptUI component	boolean, number, or string	Retrieves the value of a ScriptUI component.
readIni	string or File	ini object	Read options from an ini file and return them as an object.
setWidgetValue (widget, value)	ScriptUI component boolean, number, or string		Set the value of a ScriptUI component.

Function	Parameter Type	Returns	What it does
validate		`boolean or ini object`	Called as part of the `Process onClick` handler. Returns: 1) ini object containing the gather options if the UI is valid 2) *true* if there was a problem with validation but keep the UI open 3) *false* if there was a problem with the validation and the UI should be closed. Calls the `GenericUI.validatePanel` method.
writeIni (iniFile, ini)	`string or File ini object`	`ini object`	Writes the values in the ini object to an ini file.

Properties

Property	Value Type	What it is
cancelTxt	`string`	The text in the *Cancel* button.
documentation*	`string`	Text that describes the script being run. Displayed in the top panel of the script's UI. Setting this text to " or `undefined` will remove the nots panel.
hasBorder*	`boolean`	Whether or not to display a border around the application panel. Default is *true*.
iniFile*	`string or File`	The name of the ini file used for this UI. If not defined (default) no ini file is used. The use of absolute paths is discouraged. Having this property set to *"~/myscript.ini"* or *File(app.preferencesFolder + "/myscript.ini")* will likely result in the most portable code.
notesSize*	`number`	The height of the documentation panel in pixels. Setting this value to 0 will remove the nots panel.
notesTxt	`string`	The text label for the documentation panel.
processTxt	`string`	The text in the *Process* button.
winRect*	`object numbers: x, y, w, h`	This object defines the rectangle to use for the script's UI. Ex. `{x: 100, y:100, x: 200, y: 125}`

Methods

Method	Parameter Type	Returns	What it does
cancel (doc)	`Document`		Called when the UI has been closed without processing taking place.

Method	Parameter Type	Returns	What it does
createPanel* (pnl, ini)	Panel ini object	Panel	Called by GenericUI.createWindow to allow the script to populate the application panel with required UI components. The ini object contains the default values for the UI components.
createWindow (ini, doc)	object Document		Called by GenericUI.exec to create the window and any needed panels. The ini object contains the default option values for the UI components. The Document (optional) is the document to be processed.
errorPrompt (str)	string	boolean	Called during validation if an input error is encountered. Returns *true* if the UI should remain open and *false* if it should be closed. The string is the error message that the user will see.
exec (arg1, arg2)	ini object and/or Document		Called by the application script. This creates and launches the UI, and subsequently the processing callback to run the actual underlying script.
process* (opts, doc)	ini object Document		Called by GenericUI.exec to perform the actual application processing for this script.
run (win)	Window	ini object	Called by GenericUI.exec to display the UI and return the options gather from it. *undefined* is returned if the UI was canceled.
validatePanel*		boolean or ini object	Called by GenericUI.validate to validate the UI components in the application panel. Returns: 1) ini object containing the gather options if the UI is valid 2) *true* if there was a problem with validation but keep the UI open 3) *false* if there was a problem with the validation and the UI should be closed.

Sample Script

From the file *SampleUI.jsx*, here is the description for this script:

```
Here is a sample usage of the GenericUI framework. The script prompts
for source and target folders. PSD files found in the source folder are
converted to B&W using a Lab conversion technique courtesy of some code
from Trevor Morris. As part of the conversion, the new files have a keyword
'B&W-Luminosity' added.

Files that are already grayscale or already have the 'B&W-Luminosity' keyword are
skipped. This is especially handy if the source and target
```

```
directories are the same.
```

SampleUI.jsx

Define a class for the options to be used by our script. This is not absolutley necessary. You can use a plain object. However, using a class like this helps to document the intent of the code.

```
//
// This is the class that contains our options for this script
// The default values for this class are specified here
//
SampleUIOptions = function(obj) {
  var self = this;

  self.source = File("~").fsName;     // the source folder
  self.target = File("~").fsName;     // the target/destination folder

  // values in obj can override the values set above
  if (obj) {
    for (var idx in obj) {
      self[idx] = obj[idx];
    }
  }
};
```

Now, define a class for the script's user interface. By convention, a *UI* postfix is used in the name. This class defines only properties needed for the framework. Others specific to the script could be added as well.

```
//
// SampleUI is our UI class
//
SampleUI = function() {
  var self = this;

  self.title = "Sample UI"; // our window title
  self.notesSize = 75;      // The height of our Notes panel
  self.winRect = {          // the size of our window
    x: 200,
    y: 200,
    w: 420,
    h: 250
  };
  self.documentation =
    "This script converts color images (.psd files) found in the source " +
    "folder using a Lab conversion technique. The new files are also tagged " +
    "with a new keyword: 'B&W-Luminosity'. Existing grayscale files or " +
    "files already tagged are skipped.";

  self.iniFile = "~/Sample.ini"; // our ini file name
  self.hasBorder = true;

  self.processTxt = 'Convert';    // use Convert as name of the Process button
};
```

In order for the framework to function properly, we need to make our sample UI class a *subclass* of

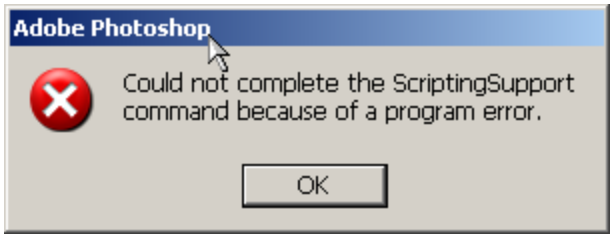
GenericUI. A *SampleUI* is a "kind-of" *GenericUI*. When we do this, any objects we create of class *SampleUI* not only have all of the propertiese that we've defined above, but they also have all of the properties and methods defined for *GenericUI*. The syntax for doing this in JavaScript looks like:

```
// make it a subclass of GenericUI
SampleUI.prototype = new GenericUI();
```

The code for the *target/output* widgets has been removed as it is nearly identical to that used for the *source* UI components.

```
// Here is where we create the components of our panel
SampleUI.prototype.createPanel = function(pnl, ini) {
  var xOfs = 10;
  var yy = 10;

  var opts = new SampleUIOptions();   // default values

  // for our panel, we have a source directory input
  var xx = xOfs;
  pnl.add('statictext', [xx,yy,xx+110,yy+20], 'Source Directory:');
  xx += 110;
  pnl.source = pnl.add('edittext', [xx,yy,xx+220,yy+20], opts.source);
  xx += 225;
  pnl.sourceBrowse = pnl.add('button', [xx,yy,xx+30,yy+20], '...');

  yy += 40;
  xx = xOfs;

  // target/output code removed....
```

At this point, any UI component callbacks have to be specified. In this example, we have the *sourceBrowse.onClick* callback function specified inline. It's nice and convenient doing it this way. However, there is a bug in PSCS2. In some cases, having mulitple inline callbacks will result in Photoshop issuing an error when the script completes. That error results in a message that looks like:



Although the script may have executed correctly and completely, the Photoshop scripting facilities become unstable and may fail unexpectedly either stopping Photoshop or requiring a shutdown/restart cycle. If this happens, you need to move the inlined functions outside of the createPanel method. This problem is not unique to GenericUI or ScriptUI but the way UI callbacks are sometimes written seems to be the most effective way of provoking this bug.

```
  // now specify the callbacks for our controls

  pnl.sourceBrowse.onClick = function() {
    try {
      var pnl = this.parent;
```

```
        var def = (pnl.source.text ? new Folder(pnl.source.text) : undefined);
        var f = Folder.selectDialog("Select a Source folder", def);

        if (f) {
          pnl.source.text = decodeURI(f.fsName);
          if (!pnl.target.text) {
            pnl.target.text = pnl.source.text;
          }
        }
      } catch (e) {
        alert(e);
      }
    }

    // target/output code removed....
```

In order for the *.ini* file support to function correctly, the values in the ini object are used to populate initial values in the appropriate UI components.

```
    if (ini) {   // if we have an ini object

      if (ini.source) {
        pnl.source.text = ini.source;  // get the source directory
      }
      if (ini.target) {
        pnl.target.text = ini.target;      // get the target directory
      }
    }
```

To finish off the method, the panel that we just filled with UI components needs to be returned.

```
    return pnl;          // return the panel object
  };
```

Now that we have the user interface constructed. we need our *validation* function which will retrieve the information that the user has set in the UI and put the contents into an *options* object. Any errors will result in a boolean being returned that indicates whether or not the user wants to try again or just end this script.

```
  //
  // code for validating our panel
  //
  SampleUI.prototype.validatePanel = function(pnl) {
    var self = this;
```

Define our options object

```
    var opts = new SampleUIOptions(); // our options object

    // A source directory must be specified and must exist
    var f;
    if (pnl.source.text) {
      f = new Folder(pnl.source.text);
    }
```

Check to see if the file was specified, and make sure it exists

```
    if (!f || !f.exists) {
        return self.errorPrompt("Source folder not found");
    }
```

Place the name of the file in the *opts.source* field.

```
    opts.source = f.fsName;

    // target/output code removed....
```

Reaching this point means that the all of the input data is present, validated, and ready for processing.

```
    // return our valid options (if we made it this far)
    return opts;
};
```

Now, it's time to write the *Process* method. The comments contained in the following code describe prescisely what this method does.

```
//
// The process callback function retrieves all of the .psd file from the
// source directory. Those that are not already grayscale and do not have
// the "B&W-Luminosity" keyword are converted to grayscale using Trevor
// Morris' luminousityChannel function. The new copy of the document has
// the "B&W-Luminosity" added and is written to the target directory.
//
SampleUI.prototype.process = function(opts) {

  var folder = new Folder(opts.source);
  var files = folder.getFiles("*.psd");     // get the .psd files from 'source'
  var fileNameSuffix = 'Lab (Luminousity)';
  var psdSaveOptions = new PhotoshopSaveOptions(); // create the save options

  psdSaveOptions.embedColorProfile = true;
  psdSaveOptions.maximizeCompatibility = true;

  for (var i = 0; i < files.length; i++) {   // for each file
    var file = files[i];
    var doc = app.open(file);
    var keywords = doc.info.keywords;

    // only do processing if the document is not grayscale already and
    // if it doesn't have 'B&W-Luminosity' as a keyword

    if (!/B&W-Luminosity/.test(keywords.toString()) &&
        doc.mode != DocumentMode.GRAYSCALE) {

      var name = doc.name.toString();
      var dupe = luminousityChannel();// convert to B&W a duplicate is returned

      var keywords = dupe.info.keywords;  // add the "B&W-Luminosity" keyword
      keywords.push("B&W-Luminosity");
      dupe.info.keywords = keywords;

      // insert fileNameSuffix between the filename and the extension
      // ex: "file.psd" becomes "file - Lab (Luminousity).psd"
```

```
        var fname = name.replace(/(\.[^\.]+)$/, " - " + fileNameSuffix + "$1");
        var file = new File(opts.target + '/' + fname);

        // save and close the B&W document
        dupe.saveAs(file, psdSaveOptions, true, Extension.LOWERCASE);
        dupe.close(SaveOptions.DONOTSAVECHANGES);
      }

      // close the original document
      doc.close(SaveOptions.DONOTSAVECHANGES);
    }
};
```

With the *Process* method completed, we need to provide the luminosityChannel function taken from Trevor's script. The primary change was to comment out his *saveFile* function call because we actually do the save (plus a couple of other things) in our *Process* method.

```
//
// The following function was pulled from BWVariations_0-3-5.jsx and slightly
// modified for use in this script.
//
////////////////////////////////////////////////////////////////////////////
// create Lightness channel variation (L*a*b* mode)
//
// Author: Trevor Morris (tmorris@fundy.net)
// Author Website: http://user.fundy.net/morris/
// Version: 0.3.5
// Source File: http://user.fundy.net/morris/downloads/scripts/BWVariations_0-3-
5.jsx
//
////////////////////////////////////////////////////////////////////////////
function luminousityChannel() {
  // duplicate and flatten the original document
  var duplicateDocument = activeDocument.duplicate();
  duplicateDocument.flatten();

  // convert document to L*a*b* mode
  if (duplicateDocument.mode != DocumentMode.LAB) {
    duplicateDocument.changeMode(ChangeMode.LAB);
  }

  // remove a* and b* channels
  duplicateDocument.channels[2].remove();
  duplicateDocument.channels[1].remove();

  // convert document to grayscale
  duplicateDocument.changeMode(ChangeMode.GRAYSCALE);

  // Modification by xbytor:
  // the call to the function saveFile has been replaced by code
  // in the caller which mimics its behavior.

  // save output
  //var fileNameSuffix = 'Lab (Luminousity)';

  //saveFile(fileNameSuffix);

  return duplicateDocument;
};
```

In the case of this script, the Cancel method does nothing. But we'll add an alert to illustrate that no matter how the window is closed without processing, we will be notified.

```
SampleUI.prototype.cancel = function() {
  alert("In Sample.cancel");                    // all we do is alert
};
```

Now that all of the above has been done, we just need to run the script.

```
SampleUI.main = function() {
  var ui = new SampleUI(); // create a new UI object
  ui.exec();               // run the UI
};

SampleUI.main();           // call our 'main' routine
```

Advanced Topics

Some of the more advanced things you can do with this framework are listed below.

- **Hardwired Options:** in some cases we may want to option values that override those that are read in from the ini file.

```
SampleUI.main = function() {
  var ui = new SampleUI(); // create a new UI object
  var opts = { source: "~/images };
  ui.exec(opts);               // run the UI
};

// or, more succinctly

SampleUI.main = function() {
  var ui = new SampleUI(); // create a new UI object
  ui.exec( { source: "~/images" } );

};
```

- **Headless Mode** is running a GenericUI script without an actual user interface opening. In order for this to work, you must first run the script with the UI turned on so that the ini file can be populated with the data that you are interested in. Then, you need to run it again with the magic 'noUI' option set to *true.* This can be done by either manually adding the line 'noUI: true' to the ini file or by manually setting the option in code like this

```
SampleUI.main = function() {
  var ui = new SampleUI(); // create a new UI object
  ui.exec( { noUI: true } );
};
```

If you change the ini file, the script will continue to run without a UI and with the same options until the *noUI* line is removed from the ini file.

- **Complex Objects** as options are possible as we see below:

```
WatermarkOptions = function(obj) {
  var self = this;

  self.fontColor = "0,0,0";

  self.getFontColor = function() {
    var c = this.fontColor;
    if (!(c instanceof SolidColor)) {
      if (c.constructor == String) {
        c = c.split(',');
      }
      if (c instanceof Array) {
        var rgbc = new SolidColor();
        rgbc.rgb.red = c[0];
        rgbc.rgb.green = c[1];
        rgbc.rgb.blue = c[2];
        c = rgbc;
      } else {
        c = undefined;
      }
    }

    return c;
  }

  self.setFontColor = function(c) {
    this.fontColor = c.rgb.red + "," + c.rgb.green + "," + c.rgb.blue;
  }
};
```

The fontColor property is what is used for reading and writing with an ini file. The get/setFontColor is what we use elsewhere in our script.

```
opts.setFontColor(app.foregroundColor);
//...
doc.activeLayer.textItem.color = opts.getFontColor();
```

It is also possible that *Object.toSource()* may offer a more general way of managing complex objects as strings.

- **Multipanel Interfaces.** One of the original intents of the framework was to have the ability to take a set of UI objects and use them to compose a larger multipanel user interfaces. This technique, while theoretically possible, has not yet been proven in an implementation.