Beginning ScriptUI

ScriptUI is a module in the Adobe CS/CC family of applications (starting in CS2 in Photoshop, in CS3 for InDesign and other CS and CC applications) with which dialogs can be added to scripts written in JavaScript. The module is included in each version of the ExtendScript Toolkit, and dialogs written in it can be used in scripts targeted at most CS/CC applications. This guide is for ScriptUI only: it assumes that you are more or less proficient in JavaScript.

To my knowledge, the only documentation available on ScriptUI is a chapter in JavaScript Tools Guide CC.pdf (in CS versions, the guide is called JavaScript Tools Guide CS*n*.pdf, where *n* stands for a CS version), which is included in every version of the ESTK and can be found in the Help menu. That chapter – referred to in this text as 'the Tools Guide' – is a complete reference for the ScriptUI environment, but it is a bit short on examples here and there. The present guide does not repeat the full reference; rather, it should be seen as a companion to it.

The present guide is InDesign-centric, which is just because InDesign is the only application that I write scripts for. When I read about ScriptUI problems in other applications or when readers alert me to such problems, I make a note of them, usually in sidenotes. You can trace these via the index.

A reference guide is available in the object-model viewer in the ESTK. Furthermore, there are alternative browsers which many people prefer over the ESTK browser. See the "Resources" section for details.

If you want to create your own ScriptUI dialogs but reading this guide is too much for you, head for Joonas Pääkkö's excellent dialog builder at https://scriptui.joonas.me/.

Peter Kahrel

Revision 2.16, June 2019 (revision details at the back)
Please send comments, suggestions, corrections, etc. to pkahrel@gmail.com

Contents

Hello world

Inevitably, the simplest window is one that prints a message on the screen. Here is an example that shows the bare basics of a ScriptUI window:

```
var myWindow = new Window ("dialog");
  var myMessage = myWindow.add ("statictext");
  myMessage.text = "Hello, world!";
myWindow.show ( );
```

The first line defines a new window of type **dialog**; the second line adds a **control** to the window, here a **statictext** item; the third line sets the content of the statictext control; the last line displays the window. To dismiss the dialog, press the exit cross in the top right of the dialog. Later we'll see how to dismiss dialogs more elegantly.

Types of window

There are two types of window: dialog and palette. You're already familiar with the difference between the two because they reflect the way that windows behave in InDesign. A **dialog** remains visible as long as you interact with it, and while it is visible you can't do anything else. Examples are the Text Frame Options and the Paragraph Rules dialogs: only after dismissing these dialogs can you continue to work in InDesign. On the other hand, if a **palette** is displayed on the screen you can continue to work in InDesign. For instance, in InDesign's UI you have the Paragraph and Character palettes (called 'panels' since CS4): you can work while these palettes are displayed. In other words, dialogs are modal, palettes are not.

In fact, scriptUI has a third window type, **window**, which appears to behave like a palette in many ways. Windows, unlike palettes, have minimise and maximise buttons. You don't see it used much and I won't deal with them in this guide.

InDesign's UI dialogs and palettes are visually distinct in that dialogs have OK and Cancel buttons, while palettes do not. But this difference doesn't necessarily hold in ScriptUI: you can do a palette with OK and Cancel buttons if you feel like it, and you could do a dialog without them (though that wouldn't be a very useful dialog).

Dialog

The Hello World script creates a dialog-type window.

Palette

To create a palette-style window you need to specify the window as a palette and you need to target a custom engine. The Hello World example can be turned into a palette as follows:

```
#targetengine "session"; // not needed in Illustrator/AfterEffects
var myWindow = new Window ("palette");
  var myMessage = myWindow.add ("statictext");
  myMessage.text = "Hello, world!";
myWindow.show ( );
```

The window displayed by this script looks exactly the same as the dialog-type window. When you try both scripts, you'll see that while the palette is displayed you can work in InDesign; but with the dialog displayed you can't do anything until you dismiss it.

There is a small cosmetic difference between dialogs and palettes (in Windows, anyway): dialogs have round corners and a larger close button, palettes have straight corners and a smaller close button. This reflects the distinction in InDesign's own windows: InDesign's dialog-type windows have rounded corners but have no close button (Textframe Options, Paragraph Style Options, etc.), while its palette-style windows have straight corners and do have a close button (Find/Change, Convert URLs to Hyperlinks, etc.). Windows 10 makes a similar difference though the close buttons look a bit different.

In CS3 and CS4 you can use palettes only if you run the script from the scripts panel. Once run from there, you can run the script from the ESTK. In CS5 and later, palettes can be run from the ESTK from the first run. These limitations are Windows only (I think).

The #targetengine directive is not needed in Illustrator and AfterEffects.

Palettes cannet be used in Photoshop.

Dialog Palette

Palettes inside functions

Palettes cannot be defined in a function. For example, the following code does nothing:

```
main();

function main () {
  var w = new Window ('palette');
  var m = w.add ('statictext');
  m.text = 'Hello, world!';
  w.show();
}
```

Instead, the window variable must be set at the script's top level. The rest of the window can be defined inside a function:

```
var win = createWindow();
win.show();

function createWindow () {
  var w = new Window ('palette');
  var m = w.add ('statictext');
  m.text = 'Hello, world!';
  return w;
}
```

It is possible to define a palette inside an anonymous function:

```
(function () {

  function createWindow () {
    var w = new Window ('palette');
    var m = w.add ('statictext');
    m.text = 'Hello, world!';
    return w;
  }

  var win = createWindow();
  win.show();
}());
```

Differences across applications

There are some differences in the behaviour and appearance of ScriptUI windows in the different CS/CC applications. A substantial difference is that palettes can't be used in Photoshop (but see Davide Barranca's blog for a workaround; for links, see the resources on p. 112). There are considerable cosmetic differences between the appearance of Photoshop CS6 dialogs and ScriptUI dialogs when run in Photoshop CS6. In earlier versions, ScriptUI dialogs were much closer to Photoshop's native dialogs. In contrast, ScriptUI dialogs in InDesign are look much the same as InDesign's native dialogs.

There are also considerable differences in appearance in applications of the same CS/CC version, but they are just cosmetic differences: the scripts behave the same in every respect.

Differences accross operating systems

There are some differences in behaviour and appearance between ScriptUI windows on Macs and Windows. These differences are relatively benign and are

pointed out throughout the text. And there are differences between Windows 7 and Windows 10 (I don't know about Windows 8, never used it.)

Adding controls

In the Hello World example we encountered our first control: **statictext**. This type of control, as we saw earlier, just adds some text to a window. Though the two-line method we used there (adding the control, then use a separate line to add the text) is fine, it could also have been stated as follows:

```
var myMessage = myWindow.add ("statictext", undefined, "Hello, world!");
```

that is, writing the text's contents as a parameter of the **.add( )** method. **undefined** in this line is a placeholder for a parameter that we won't deal with here, namely, the size and position of the text within the window or other type of container (later we'll return to containers within a window).

Another way of formulating this is as follows:

```
var myMessage = myWindow.add ("statictext {text: 'Hello, world!'}");
```

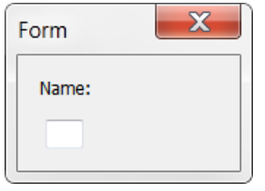
using a so-called resource string (resource strings are dealt with more extensively in the section "Resource string" on page 106). In what follows I'll use any of the three methods.

Getting started: An example

Before going into the details of all other controls, we'll first construct a simple script to illustrate the main elements of ScriptUI's window features. In discussing this simple example you also see how you often go about constructing a window, which in effect is like constructing most things: you first add the basic things, then refine these to make it more manageable and to make it look better. We'll create a script that asks a user for some input.

Text entry is done using the **edittext** control. To make sense of an edit field, you need to add a prompt separately using a **statictext** control. Here's the first attempt:

```
var myWindow = new Window ("dialog", "Form");
   myWindow.add ("statictext", undefined, "Name:");
   var myText = myWindow.add ("edittext");
myWindow.show ();
```

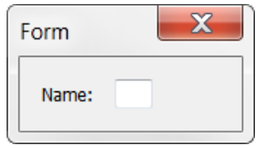


Note first that we added a title to the window ('Form'). (Titles are centred on the Mac.) Also note that the edit field appears below its title Name, which is not

For control titles, see also p. 71

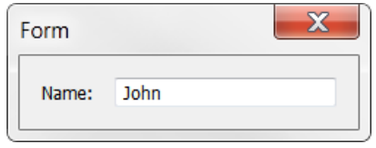
what we want. A window's default orientation is **column**. To change that to **row**, we need to include a statement to that effect:

```
var myWindow = new Window ("dialog", "Form");
  myWindow.orientation = "row";
  myWindow.add ("statictext", undefined, "Name:");
  var myText = myWindow.add ("edittext");
myWindow.show ();
```



The second line in the script sets the window's orientation. This looks a bit better, but the edit field is too small. In addition, we would like to add some default text:

```
var myWindow = new Window ("dialog", "Form");
  myWindow.orientation = "row";
  myWindow.add ("statictext", undefined, "Name:");
  var myText = myWindow.add ("edittext", undefined, "John");
    myText.characters = 30;
myWindow.show ();
```



You can set the width of an edittext control using its **characters** property. Later we'll see other ways to size controls.

The dialog looks better now, but it would be useful if the edit field were activated when the window is displayed so that the user needn't place the cursor there. This is done by including a line **myText.active = true;**

```
var myWindow = new Window ("dialog", "Form");
  myWindow.orientation = "row";
  myWindow.add ("statictext", undefined, "Name:");
  var myText = myWindow.add ("edittext", undefined, "John");
    myText.characters = 30;
    myText.active = true;
myWindow.show ();
```

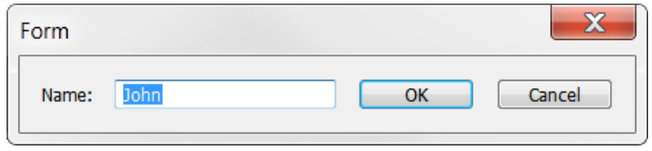
Note: If **active** appears not to work as expected, you try setting it in a so-called callback:

```
var myWindow = new Window ("dialog", "Form");
  myWindow.orientation = "row";
  myWindow.add ("statictext", undefined, "Name:");
  var myText = myWindow.add ("edittext", undefined, "John");
    myText.characters = 30;
  myWindow.onShow = function () {
    myText.active = true;
```

```
  }
myWindow.show ();
```

Now we want to add some buttons, in this case the usual OK and Cancel buttons. We do this using the **button** control:

```
var myWindow = new Window ("dialog", "Form");
  myWindow.orientation = "row";
  myWindow.add ("statictext", undefined, "Name:");
  var myText = myWindow.add ("edittext", undefined, "John");
    myText.characters = 20;
    myText.active = true;
  myWindow.add ("button", undefined, "OK");
  myWindow.add ("button", undefined, "Cancel");
myWindow.show ();
```



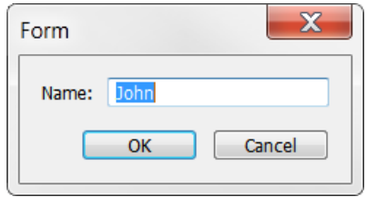
But this doesn't look right, we want the buttons laid out in a different way. We do this by grouping controls in groups and panels, to which we now turn.

Groups and panels

Because we set the window's orientation to row, all items we add are placed on the same row, but that's not what we want. To change this, we can **group** items together using the ScriptUI items **group** and **panel**. These two function the same in that they group items together, but differ in two ways: panels have a border, groups don't; and the default orientation of a group is row, that of a panel, column (panel borders can be changed in some applications; see "Panel border styles" on page 9).

So what we want to do now is to create two groups: one with the statictext and the edittext controls, the other with the two buttons. This can be done as follows:

```
var myWindow = new Window ("dialog", "Form");
  var myInputGroup = myWindow.add ("group");
    myInputGroup.add ("statictext", undefined, "Name:");
    var myText = myInputGroup.add ("edittext", undefined, "John");
      myText.characters = 20;
      myText.active = true;
  var myButtonGroup = myWindow.add ("group");
    myButtonGroup.alignment = "right";
    myButtonGroup.add ("button", undefined, "OK");
    myButtonGroup.add ("button", undefined, "Cancel");
myWindow.show ();
```



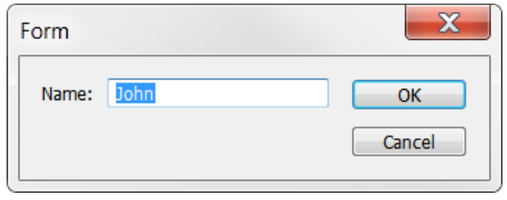
We defined two groups, **myInputGroup** and **myButtonGroup**. Note that we deleted the line that sets the window's orientation because it's not needed anymore: the window has just two items (the two groups), and since the Window's default orientation is column, there's no need to state it. Similarly, since the default orientation of groups is row, there's no need to set the orientation in the groups. Note that we aligned the button group to the right of the window using **alignment** – a small detail but I need it later on.

Groups (and panels) are good layout tools when you script windows. If used well, your windows are easily adaptable. For instance, if you want the buttons vertically aligned and to the right of the input group, all you need to do is add two orientation statements – these are marked green in the following example:

```
var myWindow = new Window ("dialog", "Form");
  myWindow.orientation = "row";
  var myInputGroup = myWindow.add ("group");
    myInputGroup.add ("statictext", undefined, "Name:");
    var myText = myInputGroup.add ("edittext", undefined, "John");
    myText.characters = 20;
    myText.active = true;
  var myButtonGroup = myWindow.add ("group");
    myButtonGroup.orientation = "column";
    myButtonGroup.add ("button", undefined, "OK");
    myButtonGroup.add ("button", undefined, "Cancel");
myWindow.show ();
```

A final tweak might be to align the two groups vertically. To do this, add this line to the script; just before or after the second line makes sense:

```
myWindow.alignChildren = "top";
```



The script's window is now displayed as shown on the right.

We'll decide that the window is good enough for our purposes, and now we turn to the question of how to deal with the user's input and how to use that input in the rest of the script. In this example, two things can happen: the user clicks OK (which in this script corresponds to pressing Enter) or they can click Cancel (the equivalent of pressing Escape). The window's behaviour is this: if the user presses OK, the line **myWindow.show( )** returns 1, if they press Esc, that line returns 2. We capture this as follows:

```
if (myWindow.show () == 1) {
  var myName = myText.text;
} else {
  exit ();
}
```

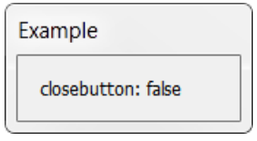
In this case we needn't check for Escape because there are just two options, namely, OK and Cancel. So if the user didn't press OK they must have pressed Cancel. Anyway, if OK was pressed we want to return to the script the contents of the edittext control, which is **myText.text**. In conclusion, here is the whole script, packed in a function as you would probably do:

```
var myName = myInput ();
// rest of the script

function myInput () {
  var myWindow = new Window ("dialog", "Form");
    var myInputGroup = myWindow.add ("group");
      myInputGroup.add ("statictext", undefined, "Name:");
      var myText = myInputGroup.add ("edittext", undefined, "John");
        myText.characters = 20;
        myText.active = true;
    var myButtonGroup = myWindow.add ("group");
      myButtonGroup.alignment = "right";
      myButtonGroup.add ("button", undefined, "OK");
      myButtonGroup.add ("button", undefined, "Cancel");
  if (myWindow.show () == 1) {
    return myText.text;
  }
  exit ();
}
```

Formatting the window frame

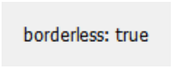
There are some properties that determine the look of palettes and dialogs. You can opt to suppress the close button on the frame (from now on I'll use w for the Window variable instead of myWindow):

```
w = new Window ("dialog", "Example", undefined, {closeButton: false});
  w.add ("statictext", undefined, "closebutton: false");
w.show ();
```



You can do borderless frames, too:

```
w = new Window ("dialog", undefined, undefined, {borderless: true});
  w.add ("statictext", undefined, "borderless: true");
w.show ();
```



But these borderless frames are pretty minimalistic in that they are in fact just grey panels. You can make them look better by adding a thin frame to them, as follows:

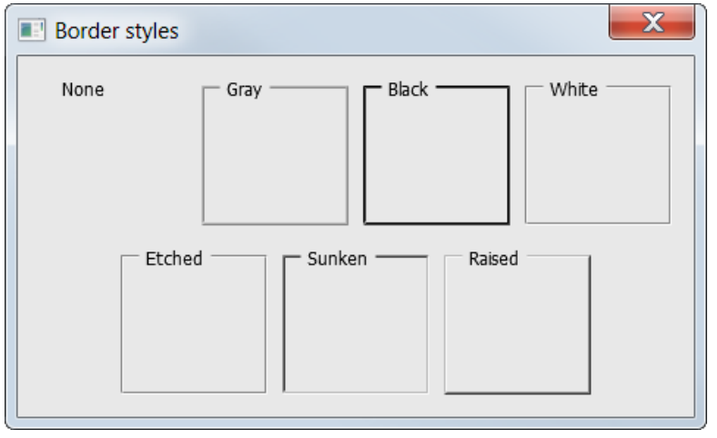
```
w = new Window ("dialog", undefined, undefined, {borderless: true});
  w.margins = [0,0,0,0];
  myPanel = w.add ("panel");
  myPanel.add ("statictext", undefined, "borderless: not quite true");
w.show ();
```

You'll notice, naturally, that the frame is not a border on the window but of the panel.

Panel border styles

Panels are drawn with a black frame rule by default. In Photoshop and After Effects (not in InDesign and Illustrator) the appearance of the frame rule can be changed using the creation property **borderStyle**:

```
w = new Window ('dialog');
  w.grp1 = w.add ('group');
    w.grp1.add('panel', [0,0,100,100], 'None', {borderStyle:'none'});
    w.grp1.add('panel', [0,0,100,100], 'Gray', {borderStyle:'gray'});
    w.grp1.add('panel', [0,0,100,100], 'Black', {borderStyle:'black'});
    w.grp1.add('panel', [0,0,100,100], 'White', {borderStyle:'white'});
  w.grp2 = w.add ('group');
    w.grp2.add('panel', [0,0,100,100], 'Etched', {borderStyle:'etched'});
    w.grp2.add('panel', [0,0,100,100], 'Sunken', {borderStyle:'sunken'});
    w.grp2.add('panel', [0,0,100,100], 'Raised', {borderStyle:'raised'});
w.show();
```



Now that we've seen the basics of windows, panels, and groups, we'll turn to the building blocks in some detail.

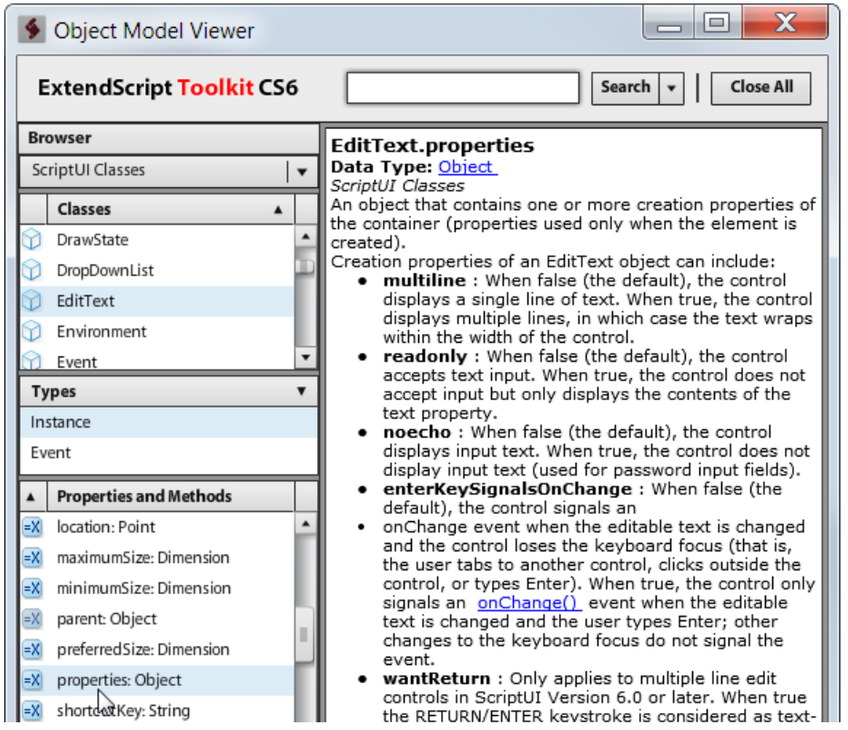
Creation properties and other properties

Windows and their controls can be modified by two types of property: creation properties and normal properties. Creation properties are so called because they must be specified when the control is created; normal properties can be set after a control was created.

There are numerous examples in this guide of both types. For instance, in the example that prints "borderless: not quite true", above, **{borderless: true}** is a creation property: the window's frame is set as borderless in the same line that creates the window. It's not possible to set that property later on in the script.

In contrast, **margins** is not a creation property. A control's margins can be set at any time. In the script, **w.margins = [0,0,0,0]** immediately follows the line at which the window is created, but that's not necessary: the margins can be set anywhere in the script after the line that creates the control.

Creation properties are listed separately in the Tools Guide, but can be found in the object-model viewer as well. A control's creation properties are shown if you click its properties property. The screenshot shows the creation properties of the edittext control.



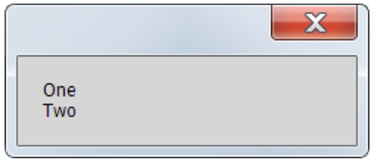
Displaying creation properties

Controls

statictext

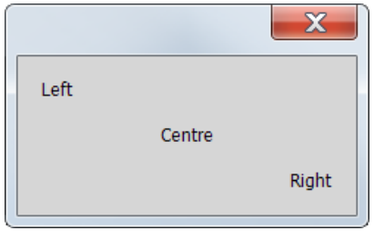
We've already seen several examples of statictext and there will be many more because this control is used a lot in ScriptUI windows. **statictext** controls are 1-line controls by default, but multi-line texts are possible when the control is created using the **multiline** creation property:

```
var w = new Window('dialog');
   w.add ('statictext', [0,0,200,50], 'One\rTwo', {multiline: true});
w.show();
```



Static text can be aligned left, right, or centre. This too must be set as a creation property, but there has always been something wrong with that. The only way you can get that to work (on Windows, anyway) is to use a so-called resource string for the control:

```
var w = new Window('dialog');
  w.add ('statictext {text: "Left", characters: 20, justify: "left"}');
  w.add ('statictext {text: "Centre", characters: 20, justify: "center"}');
  w.add ('statictext {text: "Right", characters: 20, justify: "right"}');
w.show();
```



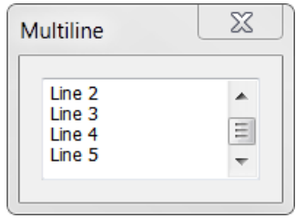
The **characters** property is used to set the control's width, and **justify** to set its alignment.

The contents of statictext controls can be changed dynamically; see pp. 15 and 15 for an example.

edittext

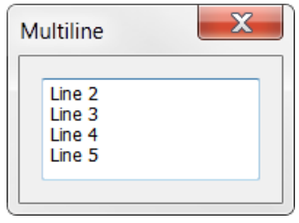
This control, too, we've seen in the example given above. It is used to get input from the user. By default, the control is just one line high and you can enter just one line of text. A useful creation property is **multiline**, which allows you to add more than one line:

```
var w = new Window ("dialog", "Multiline");
  var myText = w.add ("edittext", [0, 0, 150, 70], "", {multiline: true});
  myText.text = "Line 1\rLine 2\rLine 3\rLine 4\rLine 5\rLine 6\r";
  myText.active = true;
w.show ();
```



In the screenshot we've added a few lines. As you can see, we specified the size of the control in the second argument position ([0, 0, 150, 70]) – these dimensions are left, top, width, and height. (Note that these coordinates differ from those of InDesign's geometricBounds, which are top, left, bottom, right.) When more text is entered than fits the control, a scrollbar is added by default. To disable the scrollbar, use the creation property **scrolling**:

```
var myText = w.add ("edittext", [0, 0, 150, 70], "", {multiline: true, scrolling: false});
```



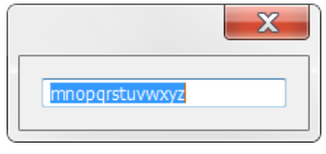
Although the control is called **edittext**, its edit possibilities are limited when you target InDesign (Photoshop and the ESTK itself don't have this problem). You can't cut and paste using keyboard shortcuts (Ctrl+C and Ctrl+V on Windows) though you can right-click in the edit window and use Copy and Paste from the context menu.

The Enter/Return key works as expected only in CS6 and later but you have to enable it using the creation property **wantReturn**, as in the following example:

```
var myText = w.add ("edittext", [0, 0, 150, 70], "", {multiline: true, wantReturn: true})
```

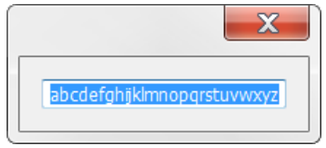
A **problem in Windows** is that if the script preselects some text, sometimes that text is not displayed correctly. Take this script:

```
var w = new Window ("dialog");
  var e = w.add ("edittext", undefined, "abcdefghijklmnopqrstuvwxyz");
  e.active = true;
w.show();
```



As you can see, the text field is sized correctly but the text is not fitted to the box. To remedy this, add the line highlighted in green:

```
w = new Window ("dialog");
   e = w.add ("edittext", undefined, "abcdefghijklmnopqrstuvwxyz");
   w.layout.layout();
   e.active = true; // this line must follow w.layout.layout()
w.show();
```



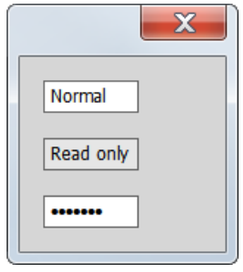
We'll not go into the details of **layout()** here, but will return to it later. Another way to get the text to display correctly is to use an onShow call-back:

```
var w = new Window ("dialog");
   var e = w.add ("edittext", undefined, "abcdefghijklmnopqrstuvwxyz");
   w.onShow = function () {
      e.active = true;
   }
w.show();
```

The alignment/justification of edittext controls can be set in the same way (and with the same quirks) as that of statictext controls.

Read-only

An edittext's **readonly** property can be set so that you can't enter text in it. You can however copy the text and the control's content is accessible to the rest of the script. This is a creation property and must applied as shown in the script below.



No-echo

Setting an edittext's noecho property makes the text unreadable by replacing it with dots. This type of field is often used to mask what a user enters, for example, when asking for a password. Though you can't read the text, it is accessible to the script via the control's **text** property.

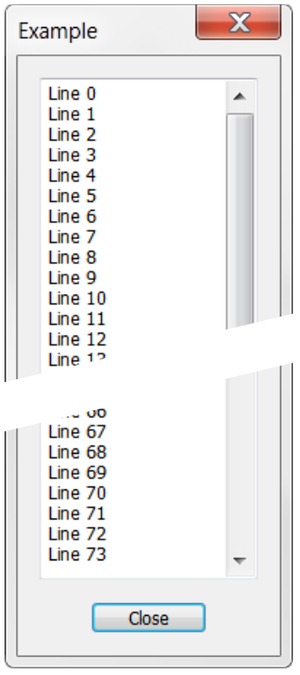
```
var w = new Window ('dialog {alignChildren: "fill"}');
   w.add ('edittext', undefined, 'Normal');
   w.add ('edittext', undefined, 'Read only', {readonly: true});
   w.add ('edittext', undefined, 'No echo', {noecho: true});
w.show();
```

Example: scrollable alert

You can have a script display text, but InDesign's native **alert()** is limited in that it's not good at handling large amounts of text. It's not so difficult, however,

to write your own display method so that you get a scrollable window. An additional advantage is that you can copy text out of that window. Here is the script, which I use a lot myself.

```
// create an example array
lines = [];
for (i = 0; i < 150; i++)
  lines.push ("Line " + String (i));
alert_scroll ("Example", lines);

function alert_scroll (title, input) {       // string, string/list
  // if input is an array, convert it to a string
  if (input instanceof Array)
    input = input.join ("\r");
  var w = new Window ("dialog", title);
  var list = w.add ("edittext", undefined, input, {multiline: true, scrolling: true});
  // the list should not be taller than the maximum possible height of the window
  list.maximumSize.height = w.maximumSize.height - 100;
  list.minimumSize.width = 150;
  w.add ("button", undefined, "Close", {name: "ok"});
  w.show ();
}
```



This function has one drawback: when you input large amounts of text or a big array, it takes a while to display it.

Controlling edit fields

Controls of type edittext are the only type of input in ScriptUI. There are no special controls for numbers, measurement units, etc. If you want input controls for numbers and measurements, you'll have to write them yourself (see p. 67 for an example of a script for a measurement control). But though the edit field is text only, ScriptUI provides several tools to make the edittext control more flexible so that you can create your own numerical and measurement controls.

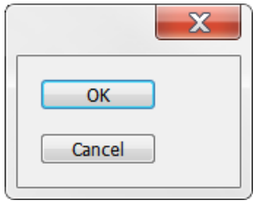
The section on event listeners gives an example of how to increment and decrement numerical values using the up and down arrow keys. Bob Stucky's scripts (see http://www.creativescripting.net/freeStuff/ScriptUI_Validation.zip) provide more examples of measurement input and field validation. See also the section on validation (p. 92), which borrows from those scripts.

button

Push buttons

There are various types of button. Earlier we saw the standard push buttons OK and Cancel, which are often used in windows:

```
var w = new Window ("dialog");
  w.add ("button", undefined, "OK");
  w.add ("button", undefined, "Cancel");
w.show ();
```



By default, an OK button responds to pressing the Enter key, a Cancel button to the Escape key. But these buttons show this behaviour only when the button's text is OK or Cancel. The Tools Guide therefore recommends to include the creation property **name: "ok"** to ensure that OK buttons behave appropriately in localised versions of InDesign or when you want to use some text for a button other than OK:

```
var w = new Window ("dialog");
  w.add ("button", undefined, "Yes", {name: "ok"});
  w.add ("button", undefined, "No", {name: "cancel"});
w.show ();
```

In this example, the Yes button behaves like an OK button and the No button like a Cancel button. (The **name** property can also be used for finding buttons and other controls in windows; see "Finding controls" on page 76 for examples.)

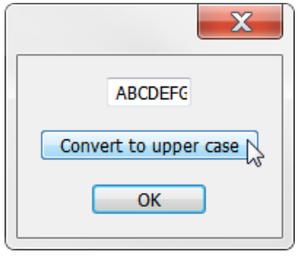
To create more than one line of text in a button, use \n as a separator (\r doesn't work here on Windows):

```
w.add ("button", undefined, "Line 1\nLine 2");
```

Responding to button presses

The behaviour of buttons other than OK and Cancel must be coded explicitely. You do this with a so-called **callback**, in this case **onClick**. Here is an example:

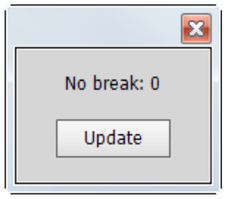
```
var w = new Window ("dialog");
  var e = w.add ("edittext", undefined, "Abcdefg");
  var convert_button = w.add ("button", undefined, "Convert to upper case");
  w.add ("button", undefined, "OK");
    convert_button.onClick = function () {e.text = e.text.toUpperCase();}
w.show ();
```

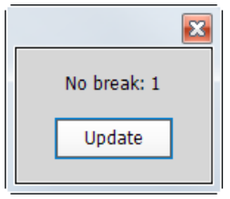


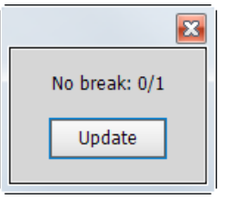
Clicking **Convert to upper case** converts the text in the edit field to upper case. This is a simple example, but the functions called by onClick can be of any complexity. (There are several other types of callback, which we will deal with later.)

Callbacks such as onClick can be combined with other callbacks in one statement. The following script displays a small panel that we can use to check the status of the state of the No break attribute.

```
#targetengine miscellaneous;
var w = new Window ('palette')
  w.nobreak = w.add ('statictext {text: "No break: ", characters: 10, justify: "center"}');
  w.button = w.add ('button {text: "Update"}');

  function updateStatus() {
    var state = '';
    if (app.selection.length > 0) {
      var str = app.selection[0].insertionPoints.everyItem().noBreak.join();
      if (str.indexOf ('false') > -1 && str.indexOf ('true') > -1) {
        state = '0/1';
      } else if (str.indexOf ('false') > -1) {
        state = '0';
      } else {
        state = '1';
      }
    }
    w.nobreak.text = 'No break: ' + state;
  }

  w.onShow = w.button.onClick = updateStatus;

w.show();
```







The state of **No break** is checked by the function updateStatus(), which is invoked when the window is first drawn (w.onShow) and when the Update button is pressed (w.button.onClick). Note how the onShow and onClick callbacks are stacked.

The script uses just one statictext control which is updated every time the Update button is clicked. (Later, in the section on finding windows, we'll redo this script as a version that updates automatically; see pp. 75 and 93.)

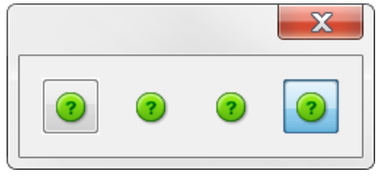
Icon buttons

Apart from these standard push buttons you can use icon buttons. These display not text such as OK but an image. The image should be in PNG, IDRC, or JPEG format. The following format is used:

```
w.add ("iconbutton", undefined, File (myFile));
```

This adds a button which is like a pushbutton with an image instead of text (button **a** in the screenshot). Buttons can be added as toolbuttons too, so that only the image is shown, not the button itself (button **b**). Finally, buttons can be made to toggle; buttons **c** and **d** in the screenshot are toggling toolbuttons: **c** is not pressed, **d** is. Here is the code:

```
var w = new Window ("dialog");
  w.orientation = "row";
  var f = File ("/d/test/icon.png")
  w.add ("iconbutton", undefined, f );    // a
  w.add ("iconbutton", undefined, f, {style: "toolbutton"});            // b
  var t1 = w.add ("iconbutton", undefined, f, {style: "toolbutton", toggle: true});       // c
  var t2 = w.add ("iconbutton", undefined, f, {style: "toolbutton", toggle: true});       // d
  t2.value = true;
w.show ();
```



a b c d

Examples of toggling toolbuttons in InDesign's interface can be found in the Text and GREP tabs of the Find/Change dialog: the buttons to determine whether or not to include footnotes, master pages, etc. in the search. The state of a toggle button is off by default, but can be enabled by setting its **value** property to true, as in the example, above. This property is used when you test the state of a toggling icon button: if (t1.value == true).

The creation properties **style** and **toggle** are meaningful only if used for icon buttons; they have no effect when included with normal buttons.

It's good practice to allow for the possibility that an icon file can't be found, e.g by adding a fallback:

```
var f = File ("/d/test/icon.idrc");
try {
  var b = w.add ("iconbutton", undefined, f )
} catch (_) {
  var b = w.add ("button", undefined, "@")
}
```

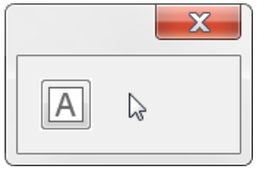
This code tries to create an icon button using "icon.idrc", and if the icon can't be found, a normal button is added with the text @.

The best way to deal with icons is to embed them in the script. Appendix 1 outlines this method.

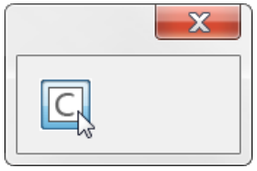
State-sensitive icon buttons

Icon buttons can be made mouse-sensitive by defining a list of images rather than a single one. They must be defined as a parameter of ScriptUI's image object. Here is an example:

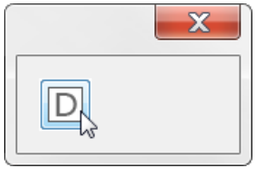
```
var dir = "/d/scriptui/fig/";
var icons = {a: File(dir+"icon-a.png"), b: File(dir+"icon-b.png"),
             c: File(dir+"icon-c.png"), d: File(dir+"icon-d.png")}

var w = new Window("dialog");
  b = w.add ("iconbutton", undefined, ScriptUI.newImage (icons.a, icons.b, icons.c, icons.d));
w.show();
```



Default

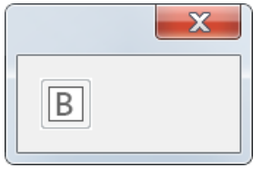


Click



Roll over

The first icon in the list (here, **icon.a**) is the default and is shown when the window is drawn. The third one, (c), becomes visible when you click the button; the last one, (d), is activated on mouse over. The second button, (b), is displayed when you disable it with a line such as b.enabled = false.



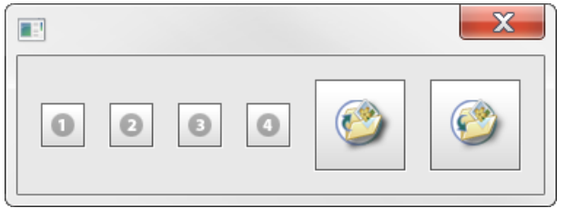
b.enabled = false;

Using application icons

A tantalising feature of ScriptUI is its ability to use application icons in its windows. I say 'tantalising' because this feature is hopelessly undocumented: just one remark in the Tools Guide, no further examples or overviews of the resource names. This is a shame, because access to system resources means that you don't have to worry about the presence of icons: they must be there since the application is there.

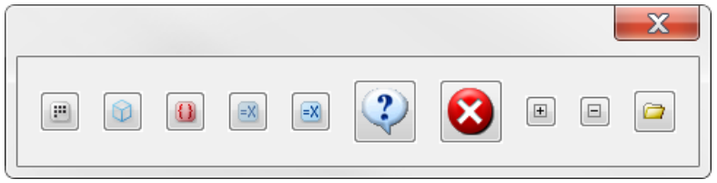
Here is an example from Photoshop:

```
#target photoshop;
var w = new Window ("dialog");
  w.orientation = "row";
  w.add ("iconbutton", undefined, "Step1Icon");
  w.add ("iconbutton", undefined, "Step2Icon");
  w.add ("iconbutton", undefined, "Step3Icon");
  w.add ("iconbutton", undefined, "Step4Icon");
  w.add ("iconbutton", undefined, "SourceFolderIcon");
  w.add ("iconbutton", undefined, "DestinationFolderIcon");
w.show();
```



There must be many more, but thus far the names of the resource icons remain a well-guarded secret. More luck with the ESTK's icons. The following script shows a few icons (a longer list is in the appendix):

```
#target estoolkit;
w = new Window ("dialog"); w.orientation = "row";
  w.add("iconbutton", undefined, "#Enumeration");
  w.add("iconbutton", undefined, "#Class");
  w.add("iconbutton", undefined, "#Method");
  w.add("iconbutton", undefined, "#PropertyRO");
  w.add("iconbutton", undefined, "#PropertyRW");
  w.add("iconbutton", undefined, "SystemQueryIcon");
  w.add("iconbutton", undefined, "SystemStopIcon");
  w.add("iconbutton", undefined, "SystemExpand");
  w.add("iconbutton", undefined, "SystemCollapse");
  w.add("iconbutton", undefined, "#FolderOpened");
w.show();
```



Using InDesign's icons

From CS5, InDesign takes a different approach in that it stores its icons as graphic files. They are stored in InDesign's subfolders under **Adobe InDesign CS*n*/Plug-Ins** (or CC, CC 2014, etc.) (PC) and **Applications/Adobe InDesign CS*n*/Adobe InDesign CS6.app** (Mac), which contains several subfolders, which in turn contain subfolders. The subfolders called **idrc_PNGA** and **idrc_PNGR** contain several icons. (Thanks to Dirk Becker for pointing this out.) These two folders contain the same icons but their colour is a bit different: InDesign uses these two versions of each icon to show the difference when you mouse over them.

The icon files have the file extension **.idrc** but they're standard PNG files which ScriptUI has no difficulty reading (but Photoshop can't read them). You could in principle write a script that uses InDesign's icons from InDesign's icon folders, but unfortunately InDesign's folder structures on Mac and PC are different and it's a bit of a hassle reliably to find the icons. It's easiest and safest to find the icon, create a resource folder on your (or your client's) computer, and keep the icons there; better yet, embed the icons in the scripts (see "Appendix 1: Embedding graphic files in a script"). You can create a catalogue of InDesign's icons using this script.

But a simpler method for using InDesign's icons is simply to make a screenshot of that an icon and cut it to the correct size. This is possible if you're not interested in the transparency that InDesign's icons offers or the different

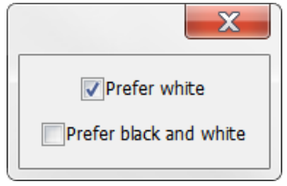
colours that you can use for the different states of icons (default, mouse over, click). And naturally, from CC, you can set the interface colour of CC applications.

Finally, it may be useful to add here that a window, group, or panel can be populated with buttons in a loop. See "Adding callbacks in loops" on page 84 for an example.

checkbox

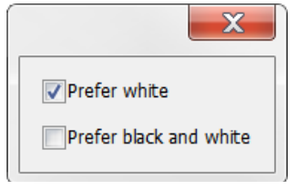
Here is an example of a window with some checkboxes. Optionally you can tick any of the boxes when the window is displayed by setting the their **value** property to **true**:

```
var w = new Window ("dialog");
  var check1 = w.add ("checkbox", undefined, "Prefer white");
  var check2 = w.add ("checkbox", undefined, "Prefer black and white");
  check1.value = true;
w.show ();
```



Here we see another window layout default: within a container, items are centred horizontally. This can be changed with the **alignChildren** property:

```
var w = new Window ("dialog");
  w.alignChildren = "left";
  var check1 = w.add ("checkbox", undefined, "Prefer white");
  var check2 = w.add ("checkbox", undefined, "Prefer black and white");
  check1.value = true;
w.show ();
```



You check whether a checkbox was ticked by comparing its value property:

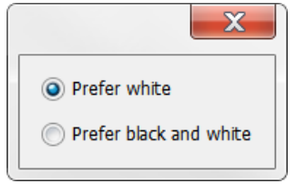
```
if (check1.value == true)
  return whatever
```

ScriptUI knows only two-state checkboxes. For three-state checkboxes, see p. 93.

radiobutton

Radio buttons are like checkboxes, but whereas you can tick all checkboxes in a container (a window, panel, group, or tabbed panel), in a container with radiobuttons only one of the buttons can be selected. Here is an example of a window with some radio buttons.

```
var w = new Window ("dialog");
  w.alignChildren = "left";
  var radio1 = w.add ("radiobutton", undefined, "Prefer white");
```



```
  var radio2 = w.add ("radiobutton", undefined, "Prefer black and white");
  radio1.value = true;
w.show ();
```

To determine which item in a radiobutton group is selected, you have to check all buttons until you hit on one whose value is true. If you have just two buttons, as in the example, that's easy:

```
if (radio1.value == true)
  // radio1 selected
else
  // radio2 selected
```

But radio buttons will typically be grouped together in a group or a panel, in which case you can cycle through the panel's children, which is an array:

```
var w = new Window ("dialog");
  var radio_group = w.add ("panel");
    radio_group.alignChildren = "left";
    radio_group.add ("radiobutton", undefined, "InDesign");
    radio_group.add ("radiobutton", undefined, "PDF");
    radio_group.add ("radiobutton", undefined, "IDML");
    radio_group.add ("radiobutton", undefined, "Text");
  w.add ("button", undefined, "OK");
  // set dialog defaults
  radio_group.children[0].value = true;

  function selected_rbutton (rbuttons) {
    for (var i = 0; i < rbuttons.children.length; i++) {
      if (rbuttons.children[i].value == true) {
        return rbuttons.children[i].text;
      }
    }
  }

  if (w.show () == 1) {
    alert ("You picked " + selected_rbutton (radio_group));
  }
```

It is wise to set a default so that the dialog always returns a valid choice. If you don't, you should take precautions against the possibility of returning an undefined object.

The example script returns the text of the selected button, but you could also return the button's index. In that case you could use that digit to pick an item

from an array. For instance, to return a file extension rather than the button's text, you could replace the last four lines of the script with these:

```
        return i;
    }

  if (w.show () == 1) {
    alert ("You picked " + ["indd", "pdf", "idml", "txt"][selected_rbutton (radio_group)]);
  }
```

Now, if you select the first item, instead of returning **InDesign**, the script returns **indd**.

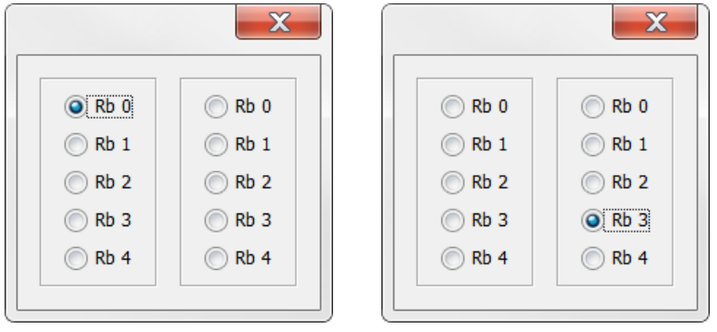
The scope of a group of radiobuttons is the group or panel in which they're defined. That means that if you want to use more than one group of radio buttons, you should place them in different groups and/or panels.

Make multiple groups act as one group

The scope of radiobuttons is their container, usually a group or a panel. This means that all radiobuttons are either laid out as a column or as a row. To achieve a more flexible layout, you can create two groups, say, group A and group B, add radiobuttons to them, and add an event listener to each group so that if you click a button in A, all buttons in B are unmarked and the other way around.

The following script adds two panels to a window, and adds five radio buttons to each panel. The event listeners respond to mouse clicks. If you click something in panel1, any button in panel2 is unmarked; and when you click something in panel2, the marked button in panel1 is unmarked.

```
var w = new Window ("dialog");
  w.orientation = "row";
  var panel1 = w.add ("panel");
    for (var i = 0; i < 5; i++) {panel1.add ("radiobutton", undefined, "Rb "+i);}
  var panel2 = w.add ("panel");
    for (var i = 0; i < 5; i++) {panel2.add ("radiobutton", undefined, "Rb "+i);}
  panel1.children[0].value = true;

  panel1.addEventListener ("click", function () {
      for (var i = 0; i < panel2.children.length; i++) {
        panel2.children[i].value = false;
      }
    }
  );
```



```
  panel2.addEventListener ("click", function () {
      for (var i = 0; i < panel1.children.length; i++) {
        panel1.children[i].value = false;
      }
    }
  );

w.show();
```

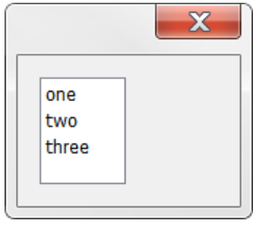
From CC, .addEventListener() doesn't work in AfterEffects and Illustrator. See https://forums.adobe.com/message/5449261#5449261#5449261 and https://forums.adobe.com/message/6173614#6173614.

You could combine the two panels in a group and define just one event listener for that group, but that just complicates the event listener and assumes some labels. We'll give that example in the section on labels.

listbox

A listbox adds a list to a window. The list can be created and filled with items when the window is created, or the items can be added later. To create a list when the window is created, include it as an array. Here is an example:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, ["one", "two", "three"]);
w.show ();
```



In InDesign CS6 on Windows 10 you have to place the listbox in a group or a panel, as follows:

```
var w = new Window ("dialog");
  var gr = w.add ("group");
    var myList = gr.add ("listbox", undefined, ["one", "two", "three"]);
w.show ();
```

Again, this is CS6 on Windows 10 only. This was noted by Werner Perplies on HilfDirSelbst.

To create a list later, or to add items to an existing list, use the method illustrated here:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox");
  myList.add ("item", "one");
  myList.add ("item", "two");
  myList.add ("item", "three");
w.show ();
```

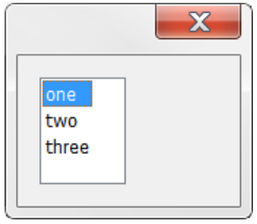
Select an item in the list by clicking it. By default, you can select just one item in a listbox. To enable multiple selection, include a creation property:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, ["one", "owo", "three"], {multiselect: true});
w.show ();
```

With multiselection enabled, you select items in the usual way: Ctrl/Cmd+click adds individual items to a selection; Shift+click adds ranges.

To let the script select an item in the list when the window is displayed, say, the first item, add this line:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, ["one", "two", "three"], {multiselect: true});
  myList.selection = 0;
w.show ();
```



In the screenshot you see white space after the highlight in the first item, and there looks to be an empty line at the foot of the the list. These spaces are reserved by ScriptUI for placing scrollbars if necessary; see the screenshots of later examples.

In a multiselect list, to select two or more items, write the indexes of the items as an array:

```
myList.selection = [0,2];
```

This selects items 0 and 2. To select a number of consecutive items you need to list them all:

```
myList.selection = [0,1,2];
```

selects the first three items in myList.

Note: making selections in a list adds to any existing selection. For instance, these lines:

```
myList.selection = [0,1];
myList.selection = [2];
```

select three items in the list. To avoid adding to a list's existing selection, start with making the selection null:

```
myList.selection = null;
myList.selection = [2];
```

Finding out which item is selected

In a single-select list, to determine which element was selected in a list, check the **selection** property of the list:

```
mySelection = myList.selection;
```

To make visible what we do in our example script, we'll use here another callback, onChange, which responds to changes to a window's control. We'll add a callback to our example script that monitors the listbox:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, ["one", "two", "three"]);

  myList.onChange = function () {
    $.writeln (myList.selection);
  }

w.show ();
```

Run the script, click e.g. "two", and the script will print **two** in the console. This looks like text, but when you try to test this with a line such as this one:

```
if (myList.selection == "two")
```

the result is **false**. The reason is that myList.selection is not text, but a **listItem** object. Amend the script as follows:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, ["one", "two", "three"]);
  myList.onChange = function () {
    $.writeln (myList.selection.constructor.name);
  }
w.show ();
```

Run the script again and click an item; the script now prints **ListItem** in the console. To get the contents of the list item, query the object's **text** property:

```
if (myList.selection.text == "two")
```

Another useful property of list items is **index**, which – unsurprisingly – returns the item's index within the list:

```
if (myList.selection.index == 2)
```

Note: before processing a list item, you should always check if anything is selected in the list:

```
myList.onChange = function () {
  if (myList.selection != null)
    $.writeln (myList.selection.constructor.name);
}
```

This goes for the other types of list, too – dropdown and treeview, see below. (The example may look odd in that **onChange** is triggered if you make a selection in a list, but there are other situations where you access a list and then it's necessary to check if any item is selected. It is therefore a good habit always to check the list's selection status.)

Forcing a list selection

Even if you select a list item when the dialog is drawn, a user can deselect the list. To make sure that there is always a selection in a list, you can monitor the list and select an item just in case the selection becomes null. My collegue Vlad Vladila suggested the following function:

```
myList.onChange = function () {
  if (myList.selection === null) {
    myList.selection = myList.prevSel;
  } else {
    myList.prevSel = myList.selection.index;
  }
}
```

Finding out which item is selected in multi-select lists

Selections in multi-select lists are a bit different in that they're arrays of list items. We'll amend slightly a previous script:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, ["one", "two", "three"], {multiselect: true});
  myList.onChange = function (){
    $.writeln (myList.selection.constructor.name);
  }
w.show ();
```

Run the script, select any two items, and the script prints **Array** in the console. This is the case even if the selection consists of one item.

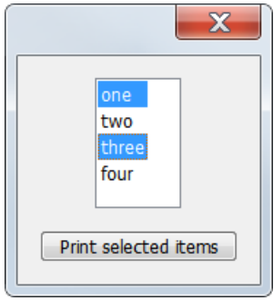
Processing lists

Lists are arrays of ListItems. They are processed a bit differently than standard arrays. The following script processes a list simply by printing the text attribute of each item:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, ["one", "two", "three"]);
  var print = w.add ("button", undefined, "Print");

  print.onClick = function () {
    for (var i = 0; i < myList.items.length; i++)
      $.writeln (myList.items[i].text);
  }

w.show ();
```

As we saw earlier, in multi-select lists, selections are arrays. Lists themselves are arrays, so selections are sub-arrays of lists. This can be shown by the following script, which prints *one* and *three* in the console:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, ["one", "two", "three", "four"], {multiselect: true});
  var print = w.add ("button", undefined, "Print selected items");

  print.onClick = function () {
    for (var i = 0; i < myList.selection.length; i++)
      $.writeln (myList.selection[i].text);
  }

w.show ();
```



Finding items in a list

To find an item in a list, use the **find** method:

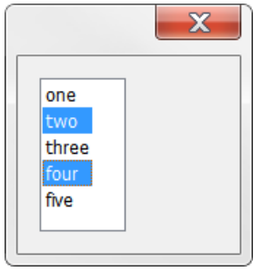
```
var myItem = myList.find ("two");
```

This returns an object of type **ListItem** if the item is found or **null** if the item is not in the list.

Using find() to make selections in a list

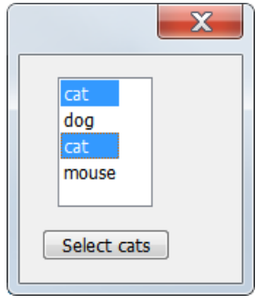
find() is a useful method to select items in a list because you can simply look for the list items' text:

```
var w = new Window ("dialog");
  var numbers = ["one", "two", "three", "four", "five"]
  var myList = w.add ("listbox", undefined, numbers, {multiselect: true});
  myList.selection = myList.find("two");
  myList.selection = myList.find("four");
w.show ();
```



Note that **find**() always returns just one item, even if there are more items with the same name. If the possibility exists that there are two or more list items with the same name and you want to find them all, then you can't use find() and you should iterate over the list and check each item's name. The following script does that: it selects all items whose name is *cat*:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, ["cat", "dog", "cat", "mouse"], {multiselect: true});
  var find = w.add ("button", undefined, "Select cats");

  find.onClick = function () {
    var found = [];
    for (var i = 0; i < myList.items.length; i++)
      if (myList.items[i].text == "cat")
        found.push (i);
    myList.selection = found;
  }

w.show ();
```



Inserting items into a list

We saw earlier that items can be added to an existing list using the **.add ("list")** method. This always adds items at the end of the list. To add an item at a particular place, include the target index. For instance, to add an item at the beginning of a list, use this line:

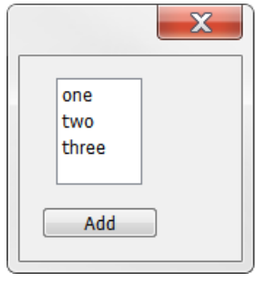
```
myList.add ("item", "zero", 0);
```

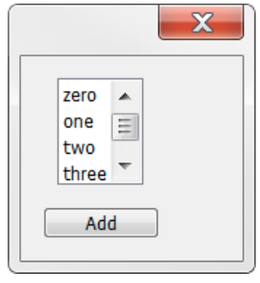
To avoid creating duplicate entries in a list, you can use the **find** method to check if the item is already in the list:

```
if (myList.find (myNewItem) == null)
  myList.add ("item", myNewItem);
```

If you add an item to a list, the window is updated automatically, as shown by this script:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, ["one", "two", "three"]);
  var b = w.add ("button", undefined, "Add");
  b.onClick = function () {myList.add ("item", "zero", 0)}
w.show ();
```





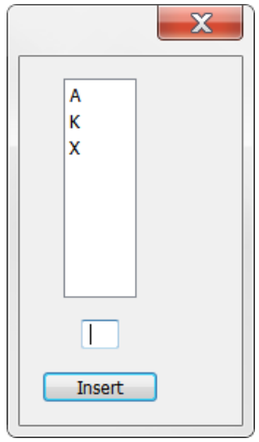
(As you can see, when the list grows out of its box, ScriptUI adds a scrollbar.)

Keeping a list sorted

Here is another example of list processing, this time inserting an item in a list and keeping the list in alphabetical order. The script activates the input prompt, so you can enter letters straight away. Notice that functions that process

a window control need not necessarily be defined within the block that defines the window (i.e. between **new Window** and **w.show**()).

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", [0, 0, 50, 150], ["A", "K", "X"]);
  var input = w.add ("edittext");
    input.active = true;
  var b = w.add ("button", undefined, "Insert", {name: "ok"});
  b.onClick = function () {
    insert_item (myList, input.text);
    input.text = "";
    input.active = true;
  }
w.show ();

function insert_item (list_obj, new_item) {
  if (list_obj.find (new_item) == null) {
    var stop = list_obj.items.length;
    var i = 0;
    while (i < stop && new_item > list_obj.items[i].text)
      i++;
    list_obj.add ("item", new_item, i);
  }
}
```

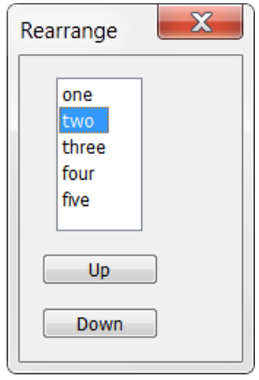


The script's interface is a bit klunky, and as we named the Insert button **ok** it responds to the Enter key – triggering the onClick handler – so it sits there just catching Enter key presses. Later on we'll handle such situations more elegantly by defining an event handler rather than using a button for this purpose.

Moving list items (single-selection lists)

To move items around in a list, we need two buttons: one to move the selected list item up, the other to move it down. We also need a function to swap two adjacent list items. Here is a simple example:

```
var w = new Window ("dialog", "Rearrange");
  var list = w.add ("listbox", undefined, ["one", "two", "three", "four", "five"]);
  list.selection = 1;
  var up = w.add ("button", undefined, "Up");
  var down = w.add ("button", undefined, "Down");

  up.onClick = function () {
    var n = list.selection.index;
    if (n > 0) {
      swap (list.items [n-1], list.items [n]);
      list.selection = n-1;
    }
  }

  down.onClick = function () {
    var n = list.selection.index;
    if (n < list.items.length-1) {
      swap (list.items [n], list.items [n+1]);
      list.selection = n+1;
    }
  }

  function swap (x, y) {
    var temp = x.text;
    x.text = y.text;
    y.text = temp;
  }

w.show ();
```



Actually, the list items themselves remain where they are: it's their text properties that are swapped in the function **swap**().

Moving list items (multi-selection lists)

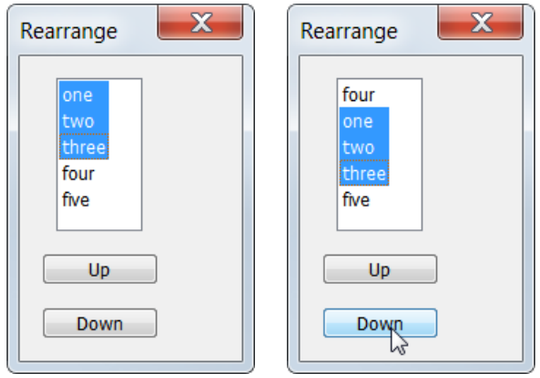
Moving a multiple selection is a bit more involved. In the previous selection we saw that to move a list item is to swap its name with the preceding or following

item's name. The same approach is used to move a multiple selection (see the script on the next page).

The **up** and **down** functions are almost the same. When we move selected items up, we first check that the first selected item is not the first item in the list. We then basically percolate the name of the item preceding the first selected item down through the list until it follows the last selected item. We then need to adjust the displayed selection. Moving items down follows the same principle.

Note that this method works correctly only on non-interrupted selections. It could be made to work on any selection, but that's a lot of extra coding. The check if the selected items are contiguous is performed by the function **contiguous**(), which is used together with the check if the selection can be moved.

```
var w = new Window ("dialog", "Rearrange");
  var list = w.add ("listbox", undefined, ["one", "two", "three", "four", "five"], {multiselect:
true});
  var up = w.add ("button", undefined, "Up");
  var down = w.add ("button", undefined, "Down");

  up.onClick = function () {
    var first = list.selection[0].index;
    if (first == 0 || !contiguous (list.selection)) return;
    var last = first+list.selection.length;
    for (var i = first; i < last; i++)
      swap (list.items [i-1], list.items [i]);
    list.selection = null;
    for (var i = first-1; i < last-1; i++)
      list.selection = i;
  }

  down.onClick = function () {
    var last = list.selection.pop().index;
    if (last == list.items.length-1 || !contiguous (list.selection)) return;
    var first = list.selection[0].index;
    for (var i = last; i >= first; i--)
      swap (list.items [i+1], list.items [i]);
    list.selection = null;
    for (var i = first+1; i <= last+1; i++)
      list.selection = i;
  }
```



```
  function contiguous (sel) {
    return sel.length == (sel[sel.length-1].index - sel[0].index + 1);
  }

  function swap (x, y) {
    var temp = x.text;
    x.text = y.text;
    y.text = temp;
  }

w.show ();
```

Removing items from a list

To remove an item from a list, use the **.remove**() method. For example, to remove the third item from a list, use this:

```
myList.remove (myList.items[2]);
```

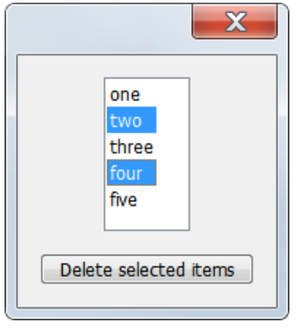
To remove an item by its name, you can use this method:

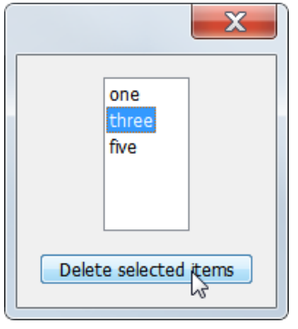
```
myList.remove (myList.find ("two"));
```

Removing items from a multi-selection list

To remove any selected items from a multi-selection list, delete the selected items back to front:

```
var w = new Window ("dialog");
  var list = w.add ("listbox", undefined, ["one", "two", "three", "four", "five"], {multiselect:
true});
  var del = w.add ("button", undefined, "Delete selected items");

  del.onClick = function () {
    // remember which line is selected
    var sel = list.selection[0].index;
    for (var i = list.selection.length-1; i > -1; i--)
      list.remove (list.selection[i]);
    // select a line after deleting one or more items
    if (sel > list.items.length-1)
      list.selection = list.items.length-1;
    else
      list.selection = sel;
  }

w.show();
```



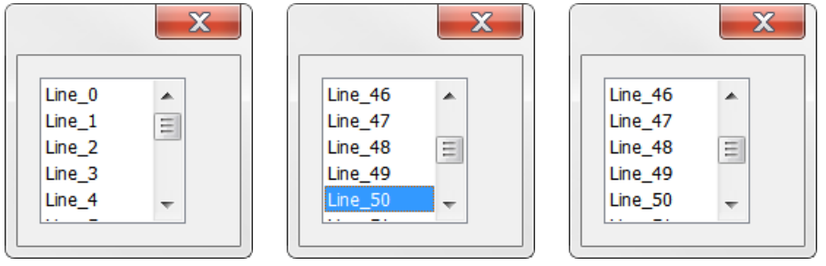


Working out which line to select after deleting the selecting list items takes more code than deleting the items themselves.

Selecting vs. revealing items

If a list is longer than its box, by default the beginning of that list is displayed, as shown in screenshot (a). If the script preselects an item, the list scrolls to make that selection visible, see (b). If you want to make a certain item visible without selecting it, use **revealItem()**; in screenshot (c) an item is displayed (Line_50) without selecting it. These three options are shown in the following script:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox");
  for (var i = 0; i < 100; i++) {
    myList.add ("item", "Line_" + String (i));
  }

  myList.preferredSize = [100,100];

  w.onShow = function () {
    //myList.selection = 50; // screenshot (b)
    //myList.revealItem ("Line_50");  // screenshot (c)
  }
w.show ();
```



(a) (b) (c)

As the script stands it displays window (a); uncomment **myList.selection = 50;** and the script shows window (b); comment out that line again and uncomment **list.revealItem ("Item_50");** to show window (c).

Note: In CC and later, selecting or revealing list items must be handled in an onShow callback. Another difference is that in CC the selected or revealed item appears at the top of the list box, not at bottom as in CS6.

There is a fourth possibility, but it's just a cosmetic variant: if you reveal an item at the end of the list and then immediately reveal an earlier item, then that earlier item is shown at the beginning of the list box. For example, with these two lines:

```
myList.revealItem (list.items.length-1);
myList.revealItem ("Line_50");
```

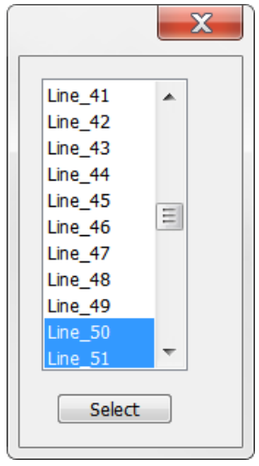
the last item is selected (but you don't get to see it) and the item **Line_50** is shown at the beginning of the box.

Like most list functions, revealItem() can be used not only with a list item's name, but also with a simple index:

```
list.revealItem (50);
```

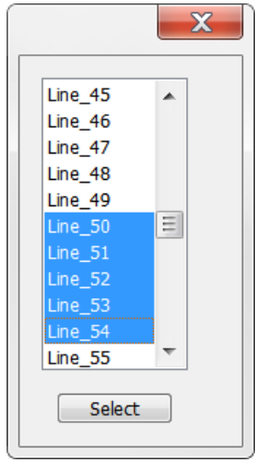
Finally, **revealItem**() can be used to create a workaround for a display bug in multiselect lists in InDesign before CC. The bug is shown by the following script:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, undefined, {multiselect: true});
  for (var i = 0; i < 100; i++)
    myList.add ("item", "Line_" + String (i))
  myList.preferredSize = [100,200];

  var b = w.add ("button", undefined, "Select");

  b.onClick = function () {
    myList.selection = [50, 51, 52, 53, 54];
  }

w.show ();
```



The problem is clear. ScriptUI's default behaviour is to scroll a list so that a selected item is shown. But in multiselect lists in which more than two items are selected, only the first two items in the selection are shown. To remedy this and show all selected items, we do a **revealItem**() on the last item of the selection. If that's handled by a separate function then other processes can make use of that same function:

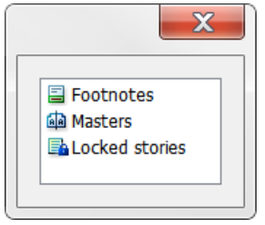
```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, undefined, {multiselect: true});
  for (var i = 0; i < 100; i++)
    myList.add ("item", "Line_" + String (i))
  myList.preferredSize = [100,200];

  var b = w.add ("button", undefined, "Select");
  b.onClick = function () {myList.selection = [50,51,52,53,54];}
  myList.onChange = ShiftList;

  function ShiftList () {
    if (this.selection != null) {
      var idx = this.selection.pop().index;
      if (idx < this.items.length) {
        this.revealItem (idx);
      }
    }
  }
w.show ();
```



Including images in a list

List items can include images; here is an example:

```
var descriptions = ["Footnotes", "Masters", "Locked stories"];
var imgs = ["footnotes.idrc", "masters.idrc", "locked_stories.idrc"];
var w = new Window ("dialog");
  var myList = w.add ("listbox");
  for (var i = 0; i < descriptions.length; i++) {
    myList.add ("item", descriptions[i]);
    myList.items[i].image = File ("~/Desktop/"+imgs[i])
  }
w.show ();
```

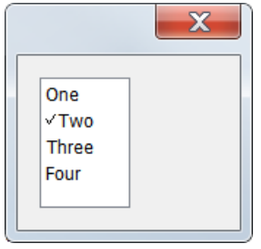


The first two lines create arrays of image names and list-item texts; the for-loop then adds list items and adds images to each item. (Like icons in icon buttons, images should be in PNG, IDRC, or JPG format.)

Adding checkmarks

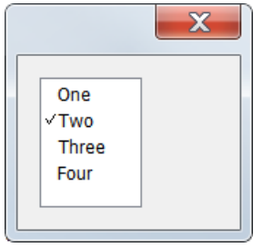
You can add checkmarks to list items by setting an item's **checked** property to true. Example:

```
var w = new Window("dialog");
  var list = w.add ("listbox", undefined, ["One", "Two", "Three", "Four"]);
  list.items[1].checked = true;
w.show();
```



But as you see, the list is now no longer aligned properly. To fix this, set the checked property of every item to false, which creates space for the check marks:

```
var w = new Window("dialog");
  var list = w.add ("listbox", undefined, ["One", "Two", "Three", "Four"]);
  for (var i = 0; i < list.items.length; i++)
    list.items[i].checked = false;
  list.items[1].checked = true;
w.show();
```



Multi-column lists

Using multi-column lists you can create table-like structures, complete with headings. Here is an example:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, "",
      {numberOfColumns: 3, showHeaders: true,
      columnTitles: ["English", "French", "Dutch"]});
  with (myList.add ("item", "One"))
    {
    subItems[0].text = "Un";
    subItems[1].text = "Een";
    }
  with (myList.add ("item", "Two"))
    {
    subItems[0].text = "Deux";
    subItems[1].text = "Twee";
    }
  with (myList.add ("item", "Three"))
    {
    subItems[0].text = "Trois";
    subItems[1].text = "Drie";
    }
w.show ();
```

Note: Multi-column lists with column headers cause Illustrator CS6 on the Mac to crash. CS5.5 and earlier and CC and later are fine, as are all versions on Windows. See this forum thread for details.

Note: like listboxes, in InDesign CS6 on Windows 10, multi-column listboxes should be placed in a group or a panel.

The widths of the columns are determined automatically, but you can set them by adding the creation property **columnWidths**. Any text that doesn't fit the column is clipped, which is indicated by an ellipsis:

```
var w = new Window ("dialog");
  var myList = w.add ("listbox", undefined, "",
    {numberOfColumns: 3, showHeaders: true,
      columnTitles: ["English", "French", "Dutch"],
      columnWidths: [30,30,100]});
```

Columns can be resized in the normal way: move the mouse cursor over a column separator in the header row (the cursor changes to ⇦⇨) and drag the column to the desired width.

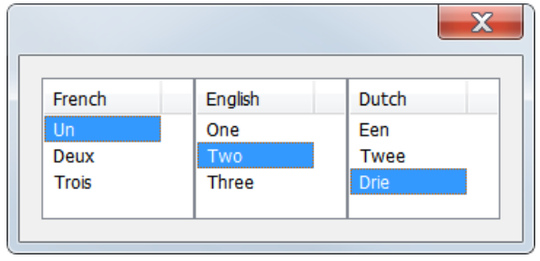
Images can be added to any of the items in any of the columns:

```
with (myList.add ("item", "One"))
  {
  subItems[0].text = "Un";
  subItems[0].image = myFile_1;
  subItems[1].text = "Een";
  subItems[1].image = myFile_2;
  }
```

Tables

Multi-column lists aren't really tables in that you can't select individual cells in a row: whichever element you click always selects the whole row. But you can fake a table by placing several list boxes next to each other without any space between them. Here's an example:

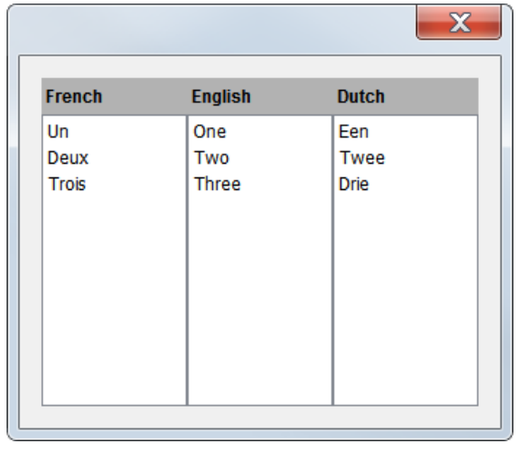
```
var w = new Window("dialog");
  var columns = w.add("group"); columns.spacing=0;
    var header = {numberOfColumns: 1, showHeaders: true, columnWidths: [80]};
    header.columnTitles = ["French"];
    var col1 = columns.add ("listbox", undefined, ["Un","Deux","Trois"], header);
    header.columnTitles = ["English"];
    var col2 = columns.add ("listbox", undefined, ["One","Two","Three"], header);
    header.columnTitles = ["Dutch"];
    var col3 = columns.add ("listbox", undefined, ["Een","Twee","Drie"], header);
w.show();
```



Now you can select individual cells. In addition, you now have some control over the appearance of individual columns (type, typesize, foreground and background colour, etc.).

But you don't have any control over the format of the column headers. If you want to change the appearnce of the headers, an altogether different script is needed. It's more elaborate, but it looks much better.

```
var w = new Window("dialog");
  w.spacing=0;
  var headers = w.add("group"); headers.spacing=0;
    headers.margins = [0,5,0,0];
    var dimH=[0,0,100,20];
    headers.add("statictext", dimH, "\u00A0French");
    headers.add("statictext", dimH, "\u00A0English");
    headers.add("statictext", dimH, "\u00A0Dutch");

    headers.graphics.backgroundColor = w.graphics.newBrush(
      w.graphics.BrushType.SOLID_COLOR,[0.7,0.7,0.7], 1);
```



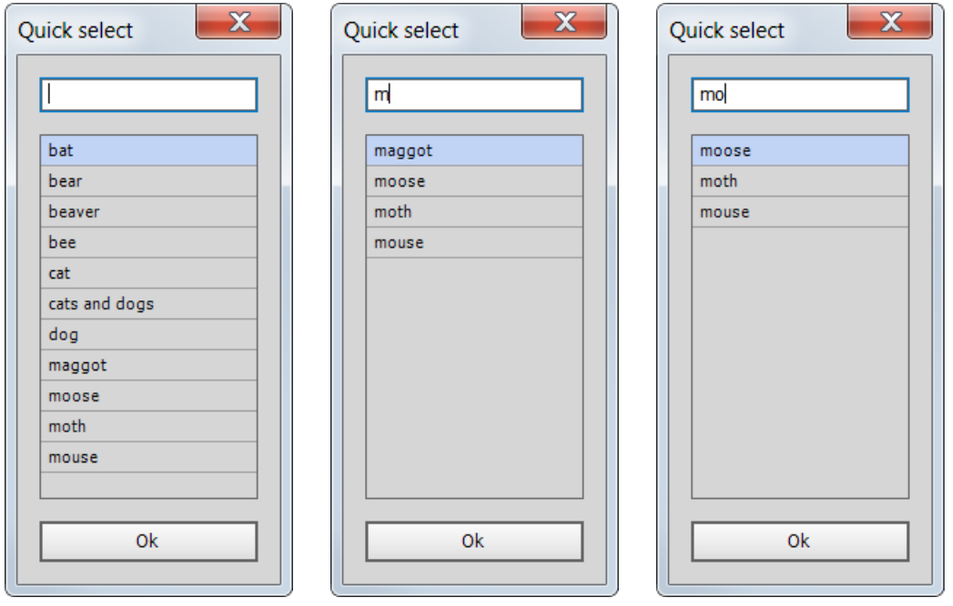
```
    for(var i = 0; i < headers.children.length; i++)
      headers.children[i].graphics.font = ScriptUI.newFont ("Arial", 'BOLD', 12)

  var columns = w.add("group"); columns.spacing=0;
    var dimC=[0,0,100,200];
    var col1 = columns.add ("listbox", dimC, ["Un","Deux","Trois"]);
    var col2 = columns.add ("listbox", dimC, ["One","Two","Three"]);
    var col3 = columns.add ("listbox", dimC, ["Een","Twee","Drie"]);

w.show();
```

The column headers are now statictext objects and are placed together in a group so that we can add a background colour. So the background is applied to the group, the font to the individual headers.

Type-ahead lists: select while you type

A useful type of list is the type-ahead list. An example of this type is InDesign's Quick Apply panel. You see a list (or part of it) and a text-entry field. While you type in the entry field, the list is filtered. Here is a script that mimics that behaviour:

```
picked = type_ahead (['bat', 'bear', 'beaver', 'bee', 'cat', 'cats and dogs', 'dog', 'maggot', 'moose',
'moth', 'mouse']);

function type_ahead (animals) {
  var w = new Window ('dialog {text: "Quick select", alignChildren: "fill"}');
    var entry = w.add ('edittext {active: true}');
    var list = w.add ('listbox', [0, 0, 150, 250], animals);
      list.selection = 0;

    entry.onChanging = function () {
      var temp = this.text;
      list.removeAll ();
      for (var i = 0; i < animals.length; i++) {
        if (animals[i].toLowerCase().indexOf (temp) == 0) {
          list.add ('item', animals[i]);
        }
      }
      if (list.items.length > 0){
        list.selection = 0;
      }
    }
```



```
    // We need the button to catch the Return/Enter key (CC and later)
    w.add ('button', undefined, 'Ok', {name: 'ok'});

    if (w.show () != 2){
      return list.selection.text;
    }
  w.close();
}
```

We make use of the callback **onChanging**, which in this script monitors the entry field. Each time we enter something in the entry field, the callback function records what's there (var temp = this.text) and empties the list. It then builds a new list by matching temp against the array we used originally to display the list. If temp matches the beginning an array element, that element is added to the list. Finally, if the list contains any entries (it's empty if you type something that doesn't match array item), that item is selected.

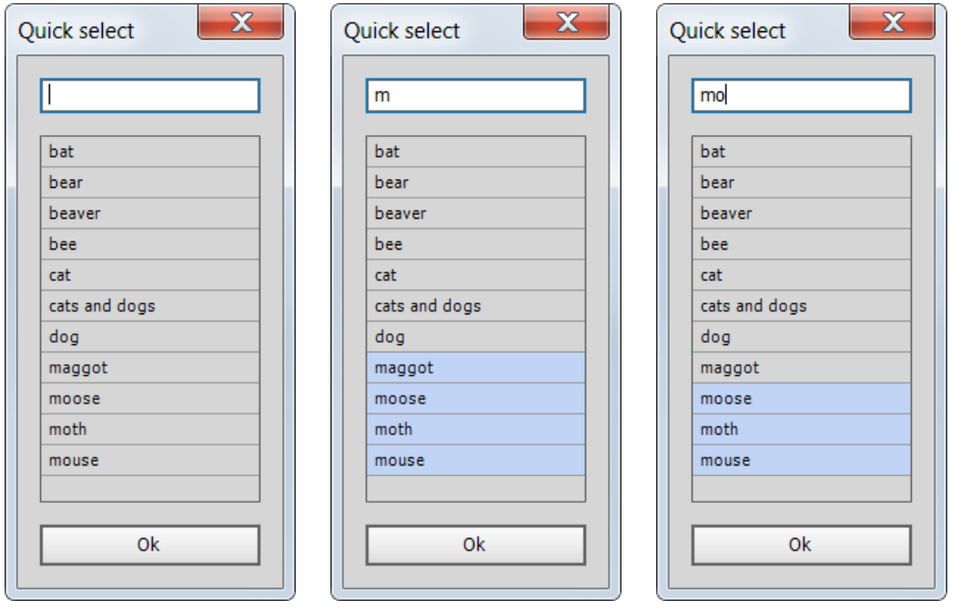
Another way of filtering a list is by highlighting the items that match some text. This is useful e.g. in long lists, which can be slow to rewrite (the above script rectreates the list on every key press). Highlighting list items is much quicker, and it is sometimes useful to keep the whole list visible. In addition, if you want to return more than one selected list item, then the following script is the one to use (naturally, a combination of both scripts is possible too).

In an earlier version of this guide, this script used the line

```
entry.onChange = function () {w.close (1)}
```

instead of the Ok button to close the window and return the selected list item. But unfortunately, that stopped working in CC, even if you explicitely set the edittext control's **enterKeySignalsOnChange** property. So if you use CC or later you have to use the button as a dummy to catch the Return/Enter key press.

```
picked = type_ahead (['bat', 'bear', 'beaver', 'bee', 'cat', 'cats and dogs', 'dog', 'maggot', 'moose', 'moth', 'mouse']);

function type_ahead (animals) {
  var selected, temp;
  var w = new Window ('dialog {text: "Quick select", alignChildren: "fill"}');
    var selected = [];
    var entry = w.add ('edittext {active: true}');

    var list = w.add ('listbox', [0,0,150,250], animals, {multiselect: true});

    entry.onChanging = function () {
      temp = this.text;
      selected = [];
      list.selection = null;
      for (var i = 0; i < animals.length; i++) {
        if (animals[i].toLowerCase().indexOf (temp) > -1) {
          selected.push(i);
        }
      }
```



```
        list.selection = selected;
      }

    w.add ('button {text: "Ok"}');

    if (w.show () != 2){
      selected = [];
      for (var i = 0; i < list.selection.length; i++) {
        selected.push (list.selection[i].text);
      }
      return selected;
    }
    w.close();
}
```

You can make the selecton of items sensitive to a regular expression by replacing **indexOf()** with **search()**, and checking whether the user typed a backslash:

```
    entry.onChanging = function () {
      temp = this.text.slice(-1) === '\\' ? this.text+'\\' : this.text;
      selected = [];
      list.selection = null;
      for (var i = 0; i < animals.length; i++) {
        if (animals[i].toLowerCase().search (temp) > -1) {
          selected.push(i);
        }
      }
      list.selection = selected;
    }
```

Now if you enter **d**, both dogs and cats and dogs are selected; but when you enter **^d**, only dogs is highlighted. Sinilarly, when you enter **\d**, all items that contain a digit are selected. The range of special symbols is restricted; you can use ^ and **$**, but [and (cause a syntax error.

For yet other application of lists, see the section on progressbars, below.

Processing long lists

The list in the **Quick select** script on the previous page is processed very quickly because it is so short. When you process long lists, however, the method used in that script (delete the list's items, then add new items) appears to slow down the script considerably.

Another approach was suggested in the HilfDirSelbst forum: create a new list and remove the old one, then assign the new list to the old list's variable:

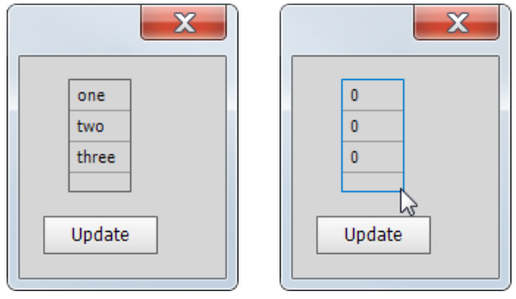
```
entry.onChanging = function () {
  var temp = entry.text,
    tempArray = [];

  for (var i = 0; i < array.length; i++)
    if (array[i].toLowerCase().indexOf (temp) == 0)
      tempArray.push (array[i]);

  if (tempArray.length > 0) {
    // Create the new list with the same bounds as the one it will replace
    tempList = w.add ("listbox", list.bounds, tempArray, {scrolling: true});
    w.remove(list);
    list = tempList;
    list.selection = 0;
  }
}
```

This method speeds up things dramatically.

Fixing display problems in listbox controls

From InDesign CC, lists aren't updated correctly if you change the contents of the text of the list items but otherwise do nothing to disturb the list. Take this script:

```
var w = new Window ('dialog');
  w.list = w.add ('listbox', undefined, ['one', 'two', 'three']);
  var b = w.add ('button {text: "Update"}');

  b.onClick = function () {
    for (var i = 0; i < w.list.items.length; i++) {
      w.list.items[i].text = '0';
    }
  }

w.show();
```



The script displays a list with three items and a button. If you click the button the script replaces the text strings in the list items with '0' – that, at least, is the intention. But in CC and later (up to CC2019 at the moment of writing) that doesn't work any longer. To update the list's display you need to click on it.

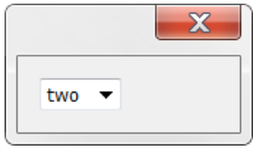
Clicking an a control to update it is not particularly good. But I was reminded of some code that Marc Autret uses in his sprites script to update the display of changed images. That method basically rattles the control, forcing ScriptUI to update the control. That method comes down to resize the control and restore its size immediately. Here is the script with the fix:

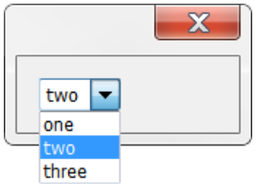
```
var w = new Window ('dialog');
  w.list = w.add ('listbox', undefined, ['one', 'two', 'three']);

  var b = w.add ('button {text: "Update"}');

  b.onClick = function () {
    for (var i = 0; i < w.list.items.length; i++) {
      w.list.items[i].text = '0';
    }
    if (parseInt(app.version) > 8) {
      refresh (w.list);
    }
  }

  function refresh (control) {
    var wh = control.size;
    control.size = [1+wh[0], 1+wh[1]];
    control.size = [wh[0], wh[1]];
  }

w.show();
```

dropdownlist

Dropdown lists are similar to listbox lists in many respects; the main differences, naturally, is that you can see one item at a time if the list is not selected and that you can select just one item at a time. Apart from that, they are processed in much the same way as list boxes. Here is an example:

```
var w = new Window ("dialog");
  var myDropdown = w.add ("dropdownlist", undefined, ["one", "two", "three"]);
  myDropdown.selection = 1;
w.show ();
```





Click the widget to expand the list, as shown in the second screenshot.

You can check which item is selected using the item's text property:

```
myChoice = myDropdown.selection.text;
```

You can also obtain the item's index:

```
myChoice = myDropdown.selection.index;
```

As with list boxes, in dropdown lists you can add images to list items:

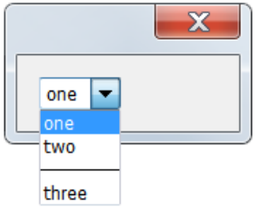
```
myDropdown.items[0].image = myImage
```

In many implementations of drop-lists, if you type a letter, the first item in the list that starts with that letter is selected in the lists's control without expanding the list (InDesign's scripted dialog system works like that, for instance). Unfortunately, ScriptUI's dropdown doesn't offer this, but that can be remedied with an event listener; see the script on p. 89.

Separators

In dropdowns (but, strangely, not in listboxes) you can add separators to a dropdown. Here's an example:

```
var w = new Window ("dialog");
  var myDropdown = w.add ("dropdownlist", undefined, ["one", "two", "-", "three"]);
  myDropdown.selection = 0;
w.show ();
```



The separator counts as a list item, so **myDropdown.items.length** returns **4**; and if you want to preselect three in the above script, you'd use **myDropdown.selection = 3**, because using **2** would do nothing – separators can't be selected.

Separators can be added after creating a dropdown, too, using the add method and specifying an insertion point:

```
var w = new Window ("dialog");
  var myDropdown = w.add ("dropdownlist", undefined, ["one", "two", "three"]);
  myDropdown.add ("separator", undefined, 2)
  myDropdown.selection = 0;
w.show ();
```

In this example the separator is added before the third item in the list. Note that you must use **undefined** as a dummy argument: the insertion point must the the third argument of the **add()** method.

Edit fields with dropdowns

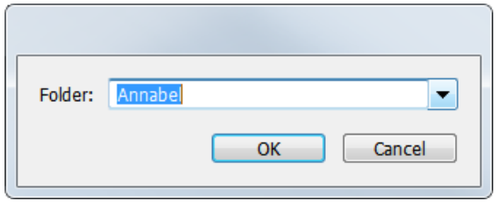
A useful list type which isn't available in ScriptUI is one that can be called 'editable dropdown', or perhaps 'edit field with a dropdown attached'. Examples are the Find What field in the Find/Change dialog and most fields in the Character panel: these are fields into which you can type something, but they

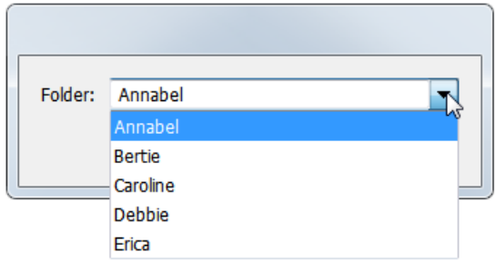
also have a widget which displays a dropdown when clicked so that you can select something from it.

But this type of list can be mimicked in ScriptUI by combining an edittext control and a dropdown list by placing one over the other using the **stack** orientation. (I have been using this method for a while, and it was mentioned by Harbs, too.) Before CC, though, the stack orientation behaves differently on Macs and PCs: on a PC, "A stack B" places B on top of A, but on a Mac it places B under A. So if you're using CS6 or earlier you need to check which operating system you're on. From CC, stack behaves the same on Macs and Windows (using the Mac style).

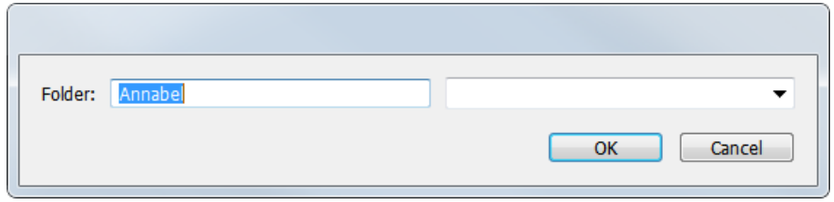
The script below displays the dialog shown in the first screenshot: a dropdown list and an editfield on top of it. The editfield is a bit narrower than the dropdown so that it doesn't mask the dropdown's widget, and the widget is the only thing you can see of the dropdown. When you press the widget, the dropdown is shown. A value can be selected in the dropdown, which is then placed in the edit field where it can be changed. We use an **onChange** callback to monitor the dropdown.

```
var names = ["Annabel", "Bertie", "Caroline", "Debbie", "Erica"];
var w = new Window ("dialog", "Place documents", undefined, {closeButton: false});
  w.alignChildren = "right";
  var main = w.add ("group");
    main.add ("statictext", undefined, "Folder: ");
    var group = main.add ("group {alignChildren: 'left', orientation: 'stack'}");
    if (File.fs !== "Windows") {
      var list = group.add ("dropdownlist", undefined, names);
      var e = group.add ("edittext");
    } else {
      var e = group.add ("edittext");
      var list = group.add ("dropdownlist", undefined, names);
    }
    e.text = names[0]; e.active = true;
    list.preferredSize.width = 240;
    e.preferredSize.width = 220; e.preferredSize.height = 20;

  var buttons = w.add ("group")
    buttons.add ("button", undefined, "OK");
    buttons.add ("button", undefined, "Cancel");

  list.onChange = function () {
    e.text = list.selection.text;
    e.active = true;
```





```
}

w.show ();
```



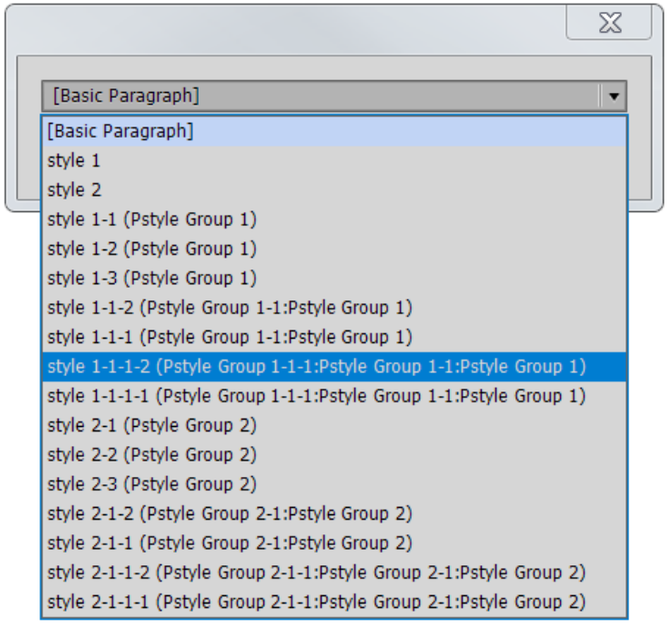
You can see how it works when you remove the stack orientation: the edittext control and the dropdown list are then displayed separately.

Creating lists on the fly

In the examples given so far, we predefined lists in the form of an array (["one", "two", "three"], ["Annabel", "Bertie", . . .]). Another way of creating a dropdown is creating it on the fly. I've used this approach to present a list of styles that includes paragraph style groups.

The following script uses a recursive function (**buildList()**) to create a list of paragraph style names with their path names, so to speak. The list's format is that of InDesign's format in the Table of Contents window.

```
function getParagraphStyle() {
  var w, temp, styles;

  function buildList (scope, list, str) {
    styles = scope.paragraphStyles.everyItem().getElements();
    for (var i = 0; i < styles.length; i++) {
      temp = list.add ('item', styles[i].name + (str == '' ? '' : ' ('+str+')'));
      temp.id = styles[i].id;  // Add property so we can easily get a handle on the style later
    }
    for (var j = 0; j < scope.paragraphStyleGroups.length; j++) {
      buildList (scope.paragraphStyleGroups[j], list, scope.paragraphStyleGroups[j].name+
(str == '' ? '' : ':') + str);
    }
  }

  w = new Window ('dialog');
    w.pstyles = w.add ('dropdownlist');
      buildList (app.activeDocument, w.pstyles, '');
      w.pstyles.remove (w.pstyles.items[0]);  // Remove [No Paragraph]
      w.pstyles.selection = 0;
    w.ok = w.add ('button {text: "OK"}');

    if (w.show () == 1){
      return app.activeDocument.paragraphStyles.itemByID (w.pstyles.selection.id);
    }
    return null;
}
```



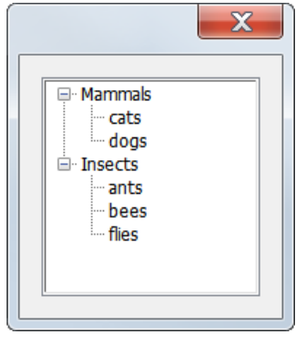
When buildList() dynamically creates the dropdown list, it adds the custom property **id** (we could use any name: it's a custom, private, property), so that we can later access the paragraph style. Because we deal with styles in groups, we can't use the name of a style, so we use its id.

This approach can be used for other types of control as well, such as listboxes and treeviews (see p. 48 for a similar example creating a tree).

treeview

Lists of type treeview create tree-like structures, much like, for instance, folder trees that you see in file managers. Here is an example:

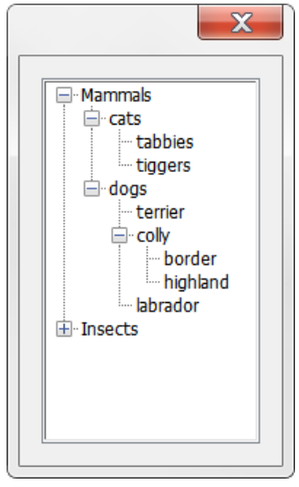
```
var w = new Window ("dialog");
  var tree = w.add ("treeview", [0, 0, 150, 150]);
  var mammals = tree.add ("node", "Mammals");
    mammals.add ("item", "cats");
    mammals.add ("item", "dogs");
  var insects = tree.add ("node", "Insects");
    insects.add ("item", "ants");
    insects.add ("item", "bees");
    insects.add ("item", "flies");
  mammals.expanded = true;
  insects.expanded = true;
w.show ();
```



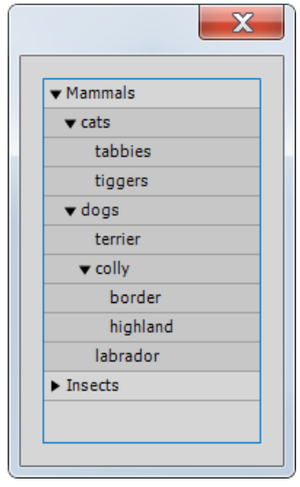
Nodes are collapsed by default (they are distinguished from items by the + or – shown before them up to CS6, by little triangles in CC). To expand a node, double-click it or single-click the plus that precedes it. To expand any node when the window is drawn, use **myNode.expanded** = **true** as shown in the example.

Nodes can be embedded under nodes to create multi-level trees. Here is an example:

```
var w = new Window ("dialog");
  var tree = w.add ("treeview", [0, 0, 150, 250]);
  var mammals = tree.add ("node", "Mammals");
    mammals.cats = mammals.add ("node", "cats");
      mammals.cats.add ("item", "tabbies");
      mammals.cats.add ("item", "tiggers");
    mammals.dogs = mammals.add ("node", "dogs");
      mammals.dogs.add ("item", "terrier");
        mammals.dogs.collies = mammals.dogs.add ("node", "colly");
        mammals.dogs.collies.add ("item", "border");
```



InDesign up to CC



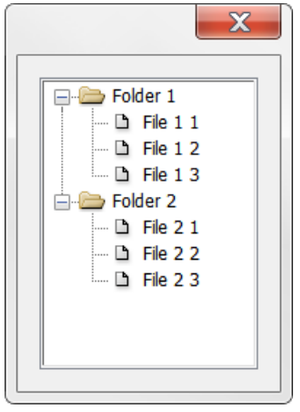
InDesign from CC

```
        mammals.dogs.collies.add ("item", "highland");
      mammals.dogs.add ("item", "labrador");
  var insects = tree.add ("node", "Insects");
    insects.add ("item", "ants");
    insects.add ("item", "bees");
    insects.add ("item", "flies");
w.show();
```

Images in treeviews

As in lists, you can add an image to nodes and items, as in the following script:

```
var w = new Window ("dialog");
  var tree = w.add ("treeview", [0, 0, 150, 200]);
  var folder_1 = tree.add ("node", "Folder 1");
    folder_1.image = File ("/d/scriptui/folder_icon.idrc");
    folder_1.add ("item", "File 1 1");
      folder_1.items[0].image = File ("/d/scriptui/file_icon.idrc");
    folder_1.add ("item", "File 1 2");
      folder_1.items[1].image = File ("/d/scriptui/file_icon.idrc");
    folder_1.add ("item", "File 1 3");
      folder_1.items[2].image = File ("/d/scriptui/file_icon.idrc");
  var folder_2 = tree.add ("node", "Folder 2");
    folder_2.image = File ("/d/scriptui/folder_icon.idrc");
    folder_2.add ("item", "File 2 1");
    folder_2.add ("item", "File 2 2");
    folder_2.add ("item", "File 2 3");
    // another method to add the icons
    for (var i = 0; i < folder_2.items.length; i++){
      folder_2.items[i].image = File ("/d/scriptui/file_icon.idrc");
    }
  folder_1.expanded = true;
  folder_2.expanded = true;
w.show ();
```



Expanding all nodes and their subnodes

In the scripts in the previous section, we expanded the two top-level nodes simply by using these two statements:

```
mammals.expanded = true;
insects.expanded = true;
```

The **expanded** property expands the node to show just its items, but any subnodes aren't expanded. This expands the whole tree because each top-

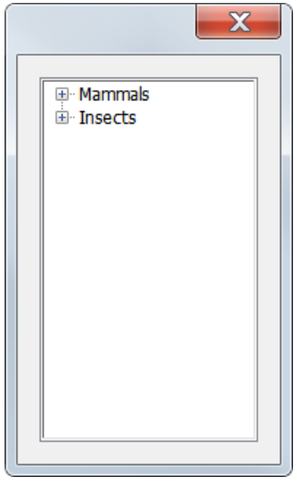
level node contained just items, not any subnodes. What we need therefore is a method of expanding all nodes when the script starts.

Another limitation of treeview nodes is that you can expand them just one level. It's not possible, for instance, to double click a node (or Alt-click, or whatever) to expand that node and all its subnodes. These two things – expanding a whole tree when the dialog is first drawn and expanding nodes exhaustively when they are double-clicked – can be handled by one function which is called in two different ways: by an **onShow()** callback when the dialog is drawn and by an event listener when a node is double-clicked.

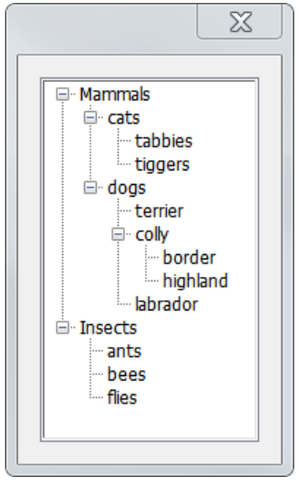
The function **expand_node()** takes two parameters, a node and a boolean. The **onShow()** function passes through the whole tree (the root node, so to speak), so that the whole tree is expanded when the window is drawn. The script also defines an onDoubleClick callback, which expands just the selected node. (Kasyan Servetsky noted that if trees are expanded using this method and the tree doesn't completely fit in the box, on Macs the top of the tree is displayed, on Windows, the last part.)

The tree does show some unexpected behaviour: when a branch is fully expanded and you double-click on it, that branch is collapsed even though the double-click callback passes true as the state for all nodes, in other words, 'expand'. It's not clear to me why this happens. It has definitely something to do with the callback, because if you leave out the function, double-click doesn't do anything at all. The function, by the way, doesn't work very well in CC.

```
var w = new Window ('dialog');
  var tree = w.add ('treeview', [0, 0, 150, 350]);
  var mammals = tree.add ('node', 'Mammals');
    mammals.cats = mammals.add ('node', 'cats');
      mammals.cats.add ('item', 'tabbies');
      mammals.cats.add ('item', 'tiggers');
    mammals.dogs = mammals.add ('node', 'dogs');
      mammals.dogs.add ('item', 'terrier');
        mammals.dogs.collies = mammals.dogs.add ('node', 'colly');
        mammals.dogs.collies.add ('item', 'border');
        mammals.dogs.collies.add ('item', 'highland');
      mammals.dogs.add ('item', 'labrador');
  var insects = tree.add ('node', 'Insects');
    insects.add ('item', 'ants');
    insects.add ('item', 'bees');
    insects.add ('item', 'flies');

  tree.onDoubleClick = function () {
```



Without the onShow callback



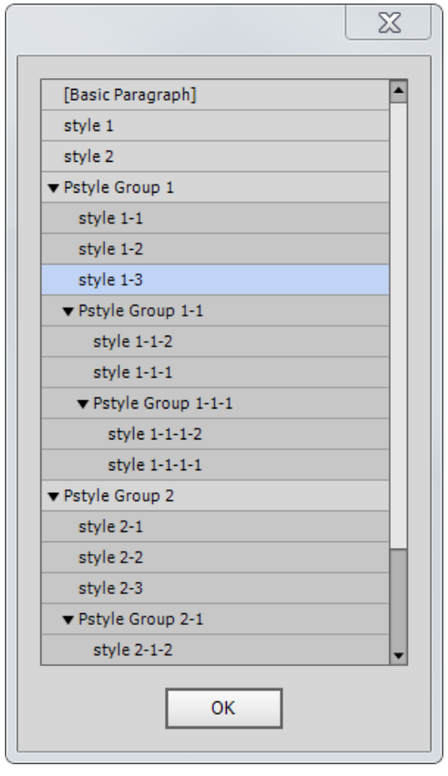
With the onShow callback

```
    if (tree.selection.type == 'node'){
      expand_node (tree.selection);
    }
  }

  function expand_node (tree){
    tree.expanded = true;
    var branches = tree.items;
    for (var i = 0; i < branches.length; i++) {
      if (branches[i].type == 'node') {
        expand_node (branches[i]);
      }
    }
  }

  w.onShow = function (){
    expand_node (tree);
  }

w.show ();
```

Creating a tree on the fly

In addition to predefining treeviews in a script, trees can be created dynamically from some structure, e.g. paragraph styles in style groups. The following script is an example.

```
function getParagraphStyle() {
  var w, temp, style;

  function buildTree (doc, tree) {
    styles = doc.paragraphStyles.everyItem().getElements();
    for (var i = 0; i < styles.length; i++) {
      temp = tree.add ('item', styles[i].name);
      temp.id = styles[i].id; // Add property so we can easily get a handle on the style later
    }
    for (var j = 0; j < doc.paragraphStyleGroups.length; j++) {
      buildTree (doc.paragraphStyleGroups[j], tree.add ('node', doc.paragraphStyleGroups[j].
name));
    }
  }

  function expand_node (tree) { /* See p. 48 */}
```



```
    w = new Window ('dialog');
      w.pstyles = w.add ('treeview', [0, 0, 250, 400]);
      buildTree (app.activeDocument, w.pstyles);
      w.pstyles.remove (w.pstyles.items[0]); // Remove [No Paragraph]
      w.ok = w.add ('button {text: "OK", enabled: false}');

      w.pstyles.onChange = function () {
        if (w.pstyles.selection != null && w.pstyles.selection.type == 'node') {
          w.pstyles.selection = null; // Non-terminal nodes should not be selectable
        }
        w.ok.enabled = w.pstyles.selection != null;
      }

      w.onShow = function () {
        expand_node (w.pstyles);
      }

    if (w.show () == 1){
      return app.activeDocument.paragraphStyles.itemByID (w.pstyles.selection.id);
    }
    return null;
}
```

The buildTree() function creates a tree that represents the paragraph-style structure of the active document. This is the same document we used for the script on p. 44 that creates a dropdownlist on the fly.

Finding and highlighting items in a tree

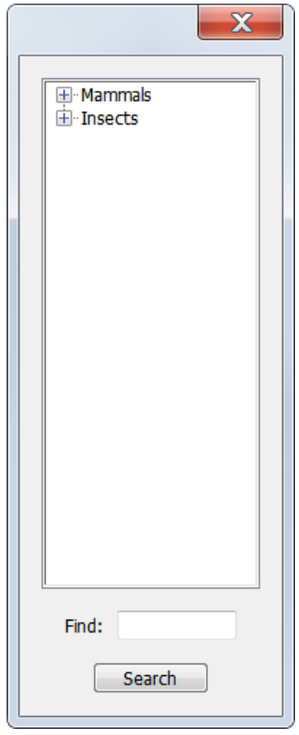
To find an item in a tree, we need to traverse the tree. The following script defines a function **find_item()**, which is a traditional recursive function, very similar to functions that traverse folders and XML structures. The function returns an array, because the tree can contain more than one item with the name we're searching.

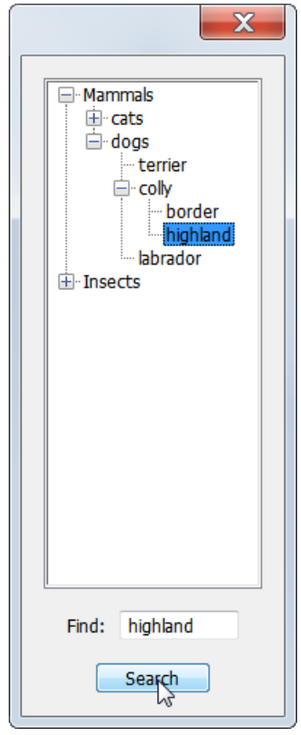
The callback defined for the Search button then expands the path of the found item and highlights the item.

Strangely, in CS6 the first time you search an item you need to press Search twice to highlight it. The first press expands the tree, the second press highlights the item. Subsequent searches work fine.

In CC, the branch is expanded correctly, but the found item is not highlighted. In addition, if you search for the last item in the tree, InDesign crashes (this is the case up to and including CC 2015).

```
var w = new Window ('dialog');
  var tree = w.add ('treeview', [0, 0, 150, 350]);
  var mammals = tree.add ('node', 'Mammals');
    mammals.cats = mammals.add ('node', 'cats');
      mammals.cats.add ('item', 'tabbies');
      mammals.cats.add ('item', 'tiggers');
    mammals.dogs = mammals.add ('node', 'dogs');
      mammals.dogs.add ('item', 'terrier');
        mammals.dogs.collies = mammals.dogs.add ('node', 'colly');
        mammals.dogs.collies.add ('item', 'border');
        mammals.dogs.collies.add ('item', 'highland');
      mammals.dogs.add ('item', 'labrador');
  var insects = tree.add ('node', 'Insects');
    insects.add ('item', 'ants');
    insects.add ('item', 'bees');
    insects.add ('item', 'flies');

  var fgroup = w.add ('group {_: StaticText {text: "Find: "}}');
    var srch = fgroup.add ('edittext {characters: 10}');

  var search_button = w.add ('button {text: "Search"}');

  search_button.onClick = function (){
    var items = find_item (tree, [], srch.text);
    if (items.length == 0) { // Nothing found
      tree.selection = null;
      return;
    }
    var item = items[0];
    var temp = item; // store this so we can select it later
    // Expand the full path
    while (item.parent.constructor.name != 'TreeView') {
      item.parent.expanded = true;
      item = item.parent;
    }
    tree.selection = temp;
    tree.active = true;
  } // search_button.onClick
```



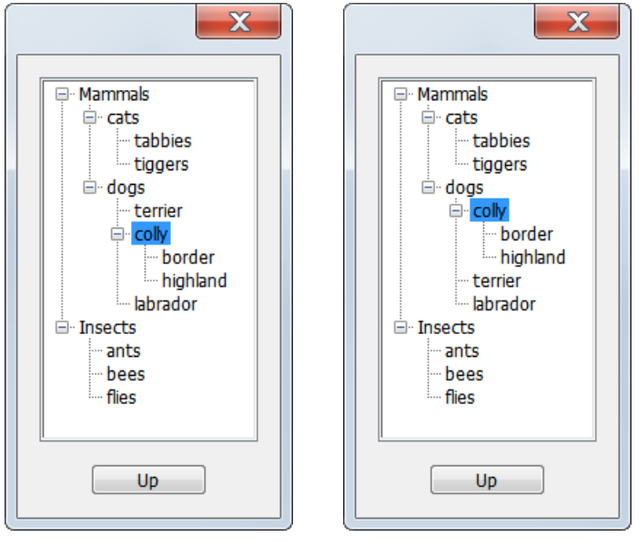


```
  function find_item (tree, list, item){
    var branches = tree.items;
    for (var i = 0; i < branches.length; i++) {
      if (branches[i].type == 'node') {
        find_item (branches[i], list, item);
      } else if (branches[i].text == item) {
        list.push (branches[i]);
      }
    }
    return list;
  }

w.show ();
```

Moving items and nodes: processing treeviews

Moving items and nodes in a treeview isn't as straightforward as moving items in a listbox. We saw that in a listbox we don't really move an item, but rather simply exchange the text properties of the item to be moved and of that of an adjacent item. But in a treeview that's possible only if the selected thing and the adjacent one are both items or if one of them is a node and the other one an item. If however the selected thing and the adjacent one are both nodes, then we really have to move the node. The following script, which is the result of an exchange with Michel Pensas, who set the exchange in motion, illustrates moving things around in a treeview.

```
var w = new Window ("dialog");
  var tree = w.add ("treeview", [0, 0, 150, 250]);
    var mammals = tree.add ("node", "Mammals");
      mammals.cats = mammals.add ("node", "cats");
        mammals.cats.add ("item", "tabbies");
        mammals.cats.add ("item", "tiggers");
      mammals.dogs = mammals.add ("node", "dogs");
        mammals.dogs.add ("item", "terrier");
        mammals.dogs.collies = mammals.dogs.add ("node", "colly");
          mammals.dogs.collies.add ("item", "border");
          mammals.dogs.collies.add ("item", "highland");
        mammals.dogs.add ("item", "labrador");
    var insects = tree.add ("node", "Insects");
      insects.add ("item", "ants");
      insects.add ("item", "bees");
      insects.add ("item", "flies");
```



```
up = w.add ("button", undefined, "Up");
up.onClick = MoveUp;

function MoveUp ()
  {
  if (tree.selection.index > 0)
    {
    var sel = tree.selection;
    var previous = sel.parent.items[sel.index-1]
    if (sel.type == "item" && previous.type == "item"){
      swap (sel, previous); tree.selection = previous;
      return;
    }
    if (sel.type == "node" && previous.type == "item"){
      sel.parent.add ("item", previous.text, sel.index+1);
      tree.remove (previous); // no need to select
      return;
    }
    if (sel.type == "item" && previous.type == "node"){
      tree.selection = sel.parent.add ("item", sel.text, sel.index-1);
      tree.remove (sel);
      return;
    }
    // When we get here we know that both items are nodes
    var target = sel.parent.add ("node", sel.text, sel.index-1);
    for (var i = 0; i < sel.items.length; i++)
      copy_branch (sel.items[i], target);
    tree.selection = target;
    tree.remove(sel);
  } // if (tree.
} // MoveUp

function copy_branch (N, Ncopy)
  {
  var NewNode = Ncopy.add (N.type, N.text);
  if (N.type == "node")
    {
    var kids = N.items;
    for (var i = 0; i < kids.length; i++)
      {
      if (kids[i].type == "node")
        copy_branch (kids[i], NewNode);
      else
```

```
            NewNode.add ("item", kids[i].text);
          }
        }
      }

    function swap (x, y){
      var temp = x.text;
      x.text = y.text;
      y.text = temp;
      }

    w.show ();
```

The script moves items up, not down; moving down is just a slight variant. The MoveUp function checks the types of the selected thing and, if the selected thing's index is bigger than 0, in other words, if it isn't the first in the list, then it looks at the selected thing and the immediately preceding thing.

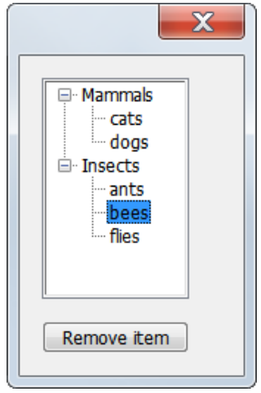
1. If the selection and what precedes it are both items, then the script simply swaps the text properties.
2. If the selection is a node and what precedes it is an item, we create a copy of the preceding item below the node and delete the original.
3. If the selecion is an item and the thing above it is a node, then the script creates a copy of the item above the node and deletes the item.
4. And finally it gets interesting: the last logical case is that both the selection and what's above it are nodes. The script creates a copy of the selected node before the preceding node, then call the function **copy_branch**, which copies all subnodes and items from the selected node to the copied node. When that's done, the selected node is deleted.

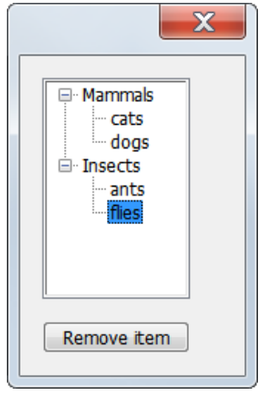
Note that in the MoveUp function the script always selects the copied item or node. This must be written into the script: it doesn't happen automatically. It's not really necessary, it's just convenient in case you want to continue moving the selected item.

Removing items and nodes from treeviews

Removing items from a treeview is similar to removing items from lists. If you remove a node, all its children are deleted as well, so you may want to add a test to prevent nodes from being deleted. In the following script a button is added to the dialog which calls a function that removes an item only if that item is not a node.

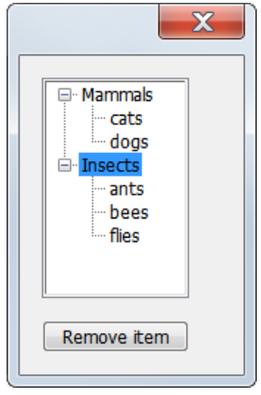
```
var w = new Window ("dialog");
  var tree = w.add ("treeview", [0, 0, 100, 150]);
    var mammals = tree.add ("node", "Mammals");
      mammals.add ("item", "cats");
      mammals.add ("item", "dogs");
    var insects = tree.add ("node", "Insects");
      insects.add ("item", "ants");
      insects.add ("item", "bees");
      insects.add ("item", "flies");
    mammals.expanded = true;
    insects.expanded = true;

  var remove_btn = w.add ("button", undefined, "Remove item");

  remove_btn.onClick = function (){
    if (tree.selection.type != "node")
      tree.remove (tree.selection);
  }
w.show ();
```





Another way of protecting nodes is to disable the delete button if a node is selected. So instead of the single function in the above script, you could have this code:

```
tree.onChange = function () {
  if (tree.selection.type == "node") {
    remove_btn.enabled = false;
  } else {
    remove_btn.enabled = true;
  }
}

remove_btn.onClick = function (){tree.remove (tree.selection);}
```



As you can see in the screenshot, when you select a node in the tree, the **Remove item** button is disabled. There is therefore now no need to do any checks in the function that deletes the item.

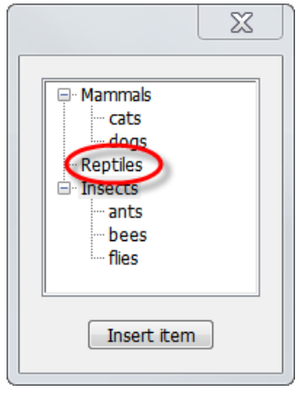
Adding items to a treeview

Adding items to a treeview is less straightforward than removing items. The trouble is that you can't use the index of items in the same way as in plain lists. The reason is that each node creates its own set of indexes. The indexes in our example tree are as follows:

```
Mammals   0
    cats    0
    dogs    1
Insects   1
    ants    0
    bees    1
    flies   2
```

Within the treeview, Mammals has index 0, Insects has index 1. Within Mammals, cats is 0; within Insects, ants is 0. To insert an item you must therefore address the correct parent node. Using the structure in our example, that can be done as follows:

```
var w = new Window ("dialog");
  var tree = w.add ("treeview", [0, 0, 150, 150]);
  var mammals = tree.add ("node", "Mammals");
    mammals.add ("item", "cats");
    mammals.add ("item", "dogs");
  var insects = tree.add ("node", "Insects");
    insects.add ("item", "ants");
    insects.add ("item", "bees");
    insects.add ("item", "flies");
  mammals.expanded = true;
  insects.expanded = true;

  var insert_btn = w.add ("button", undefined, "Insert item");

  insert_btn.onClick = function ()
    {
    tree.selection.parent.add ("item", "Reptiles", tree.selection.index);
    }

w.show ();
```

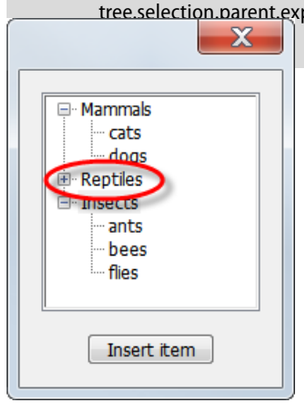


Elements are added within their branch. If you select a node, the added element is inserted at the level of that node but as an item, as shown in the screenshot next to the script code. To insert that element as a node, you need to check the current selection's type. The following context-sensitive function inserts an element of the correct type at each level:

```
  insert_btn.onClick = function ()
    {
    tree.selection.parent.add (tree.selection.type, "Reptiles", tree.selection.index);
    }
```

In CC and later, adding an item collapses the selection's parent. To fix that, add these two lines so that the callback looks as follows:

```
  insert_btn.onClick = function () {
        tree.selection.parent.add ("item", "Reptiles", tree.selection.
  index);
        tree.selection.parent.expanded = false;
        tree.selection.parent.expanded = true;
```



Writing a treeview as XML

It will be clear by now that any form of tree-processing will have to be done by recursive functions. The **expand_all** and **move_up** functions shown earlier are examples. Another example would be to write out a treeview as an XML file. Here is an example – the panel on the right shows the script's output:

```
var w = new Window ("dialog");
  var tree = w.add ("treeview", [0, 0, 150, 250]);
    var mammals = tree.add ("node", "Mammals");
      mammals.cats = mammals.add ("node", "cats");
        mammals.cats.add ("item", "tabbies"); mammals.cats.add ("item", "tiggers");
      mammals.dogs = mammals.add ("node", "dogs");
        mammals.dogs.add ("item", "terrier");
          mammals.dogs.collies = mammals.dogs.add ("node", "colly");
          mammals.dogs.collies.add ("item", "border");
          mammals.dogs.collies.add ("item", "highland");
        mammals.dogs.add ("item", "labrador");
    var insects = tree.add ("node", "Insects");
      insects.add ("item", "ants"); insects.add ("item", "bees"); insects.add ("item", "flies");

    xml = w.add ("button", undefined, "Write XML");
    xml.onClick = exportXML;

  function exportXML (){
    $.writeln ('<?xml version="1.0" encoding="UTF-8" standalone="yes"?>\r<TreeView>');
    for (var i = 0; i < tree.children.length; i++)
      writeXML (tree.children[i], 1);
    $.writeln ("</TreeView>");
    }

  function writeXML (node, level) {
    $.writeln (indent (level), "<"+node.text+">");
    var kids = node.items;
    for (var i = 0; i < kids.length; i++) {
      if (kids[i].type == "node")
        writeXML (kids[i], level+1);
      else
        $.writeln (indent (level+1), "<item>" + kids[i].text + "</item>");
    } // for
    $.writeln (indent (level), "</"+node.text+">");
  }

  function indent (n) {
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<TreeView>
    <Mammals>
        <cats>
            <item>tabbies</item>
            <item>tiggers</item>
        </cats>
        <dogs>
            <item>terrier</item>
            <colly>
                <item>border</item>
                <item>highland</item>
            </colly>
            <item>labrador</item>
        </dogs>
    </Mammals>
    <Insects>
        <item>ants</item>
        <item>bees</item>
        <item>flies</item>
    </Insects>
</TreeView>
```

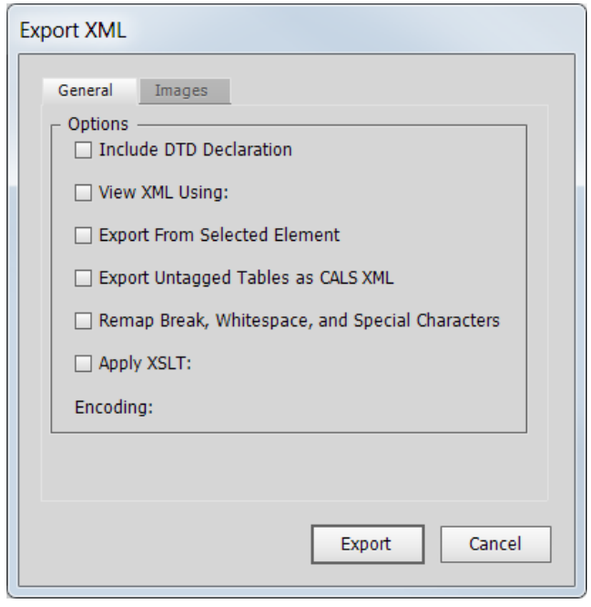
```
    var s = ""; for (var i = 0; i < n; i++) {s += "\t";}
    return s;
    }

  w.show ();
```

Two examples of treeview controls are Gabe Harbs's script that shows based-on relationships between paragraph styles and the class picker in a GREP editor script.

tabbedpanel

Tabbed panels are are ScriptUI's third main container type. They are defined like groups and panels using the identifier **tabbedpanel**. You're familiar with them because InDesign uses several dialogs of this type. By way of example, the following script partially reproduces one of InDesign's tabbed-panel dialogs, namely, XML export. There's nothing you can do about the appearance of the borders of the tabs.

```
var w = new Window ("dialog", "Export XML", undefined, {closeButton: false});
  w.alignChildren = "right";
  var tpanel = w.add ("tabbedpanel");
    tpanel.alignChildren = ["fill", "fill"];
    tpanel.preferredSize = [350,300];
    var general = tpanel.add ("tab", undefined, "General");
      general.alignChildren = "fill";
      var g_options = general.add ("panel", undefined, "Options");
        g_options.alignChildren = "left";
        g_options.dtd_decl = g_options.add ("checkbox", undefined, "Include DTD
Declaration");
        g_options.view_XML = g_options.add ("checkbox", undefined, "View XML Using: ");
        g_options.export_sel = g_options.add ("checkbox", undefined, "Export From
Selected Element");
        g_options.export_untagged = g_options.add ("checkbox", undefined, "Export
Untagged Tables as CALS XML");
        g_options.remap = g_options.add ("checkbox", undefined, "Remap Break,
Whitespace, and Special Characters");
        g_options.xslt = g_options.add ("checkbox", undefined, "Apply XSLT: ");
        g_options.add ("statictext", undefined, "Encoding: ");
    var images = tpanel.add ("tab", undefined, "Images");
      images.alignChildren = "fill";
      var img_options = images.add ("panel", undefined, "Image Options");
  var buttons = w.add ("group");
    buttons.add ("button", undefined, "Export", {name: "ok"});
```



```
    buttons.add ("button", undefined, "Cancel");
w.show ();
```

Tabs can be preselected just like items in a list. For example, the above script starts with the General tab selected; to open the Images tab on start-up, use this line:

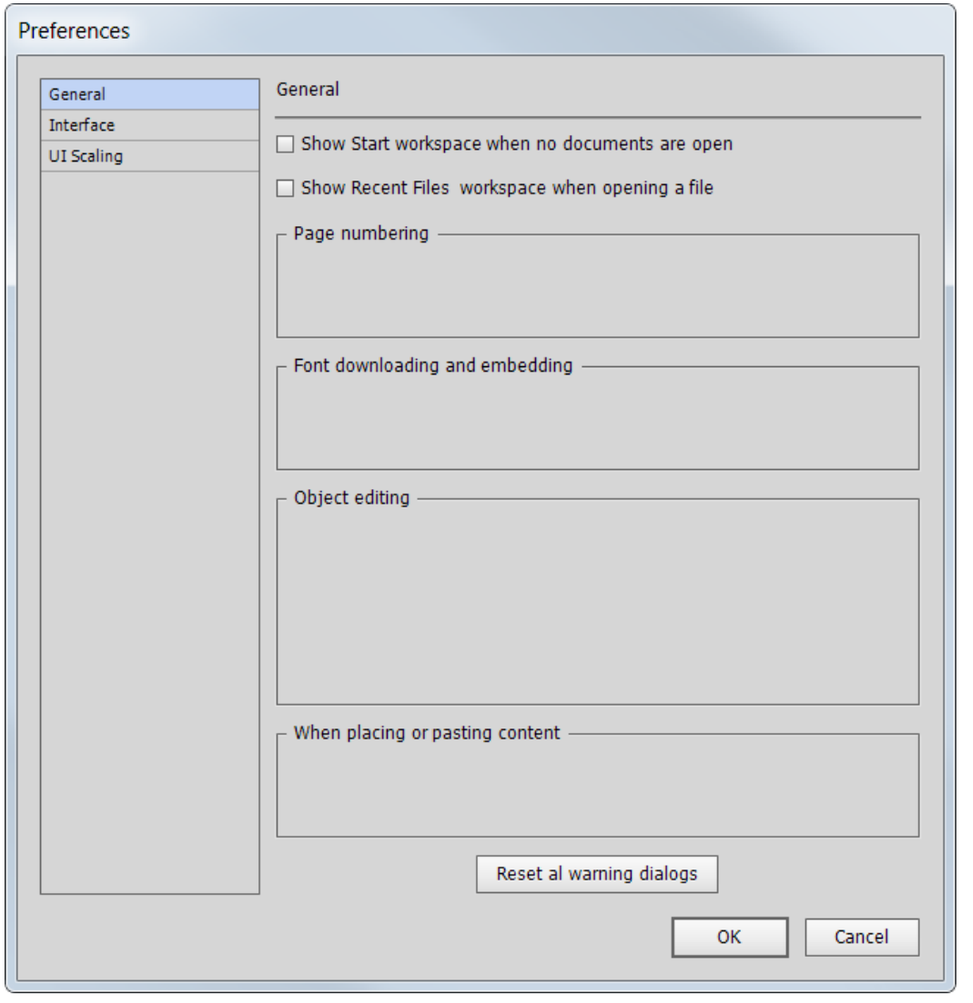
```
tpanel.selection = 1;
```

Vertical tabs

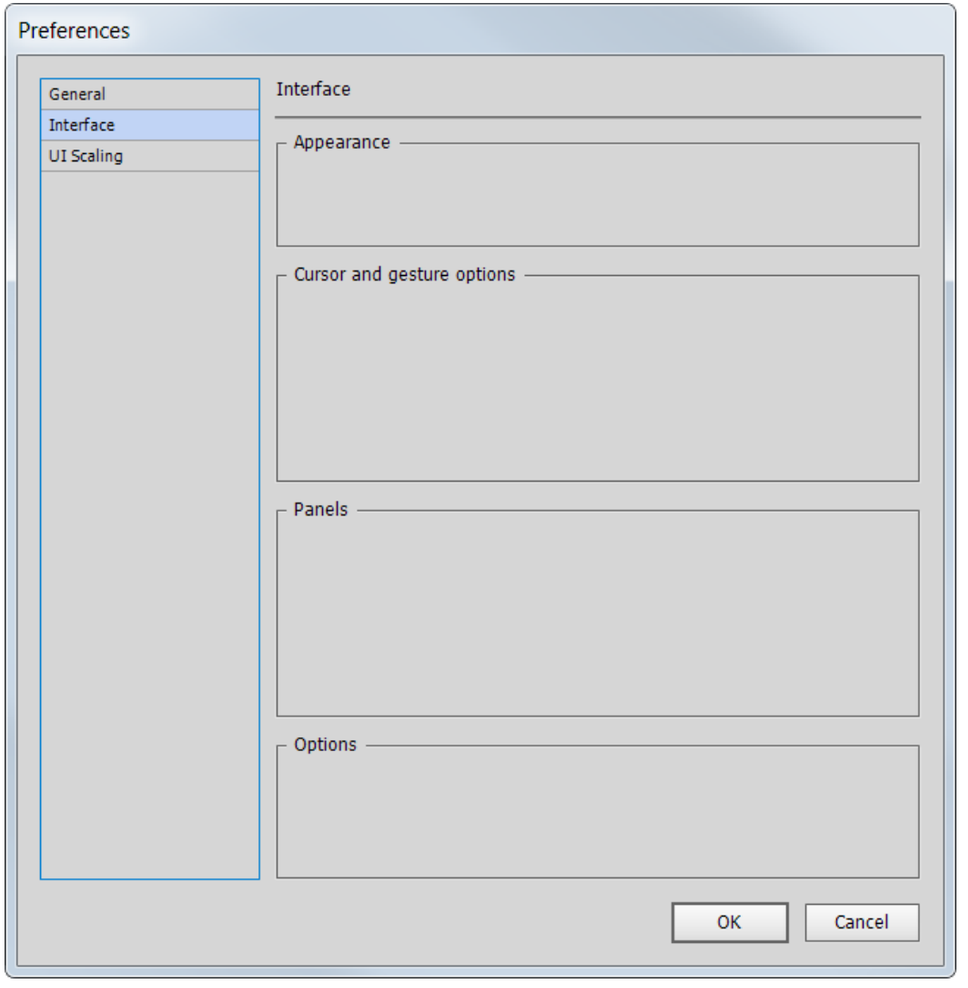
ScriptUI doesn't natively support the kind of vertically tabbed windows that you see in InDesign. Examples of such windows are InDesign's Preferences window and the paragraph and character style windows. But using a combination of the properties **stack** and **visible**, that type of window is not difficult to emulate. The following script mimics (part of) InDesign's Preferences window.

The idea is to create a group or a panel for each tab and stack align them so that they're all in the same location in the window. An onChange event handler on the list makes all 'tabs' (groups/panels) invisible, then makes the selected tab visible.

```
var w = new Window ('dialog {text: "Preferences", orientation: "column", alignChildren:
                    ["fill","fill"], properties: {closeButton: false}}');
  w.main = w.add ('group {preferredSize: [600, 500], alignChildren: ["left","fill"]}');
  w.stubs = w.main.add ('listbox', undefined, ['General', 'Interface', 'UI Scaling']);
    w.stubs.preferredSize.width = 150;
  w.tabGroup = w.main.add ('group {alignment: ["fill","fill"], orientation: "stack"}');

  w.tabs = [];
    w.tabs[0] = w.tabGroup.add ('group');
      w.tabs[0].add ('statictext {text: "General"}');
      w.tabs[0].add ('panel');
      w.tabs[0].add ('checkbox {text: "Show Start workspace when no documents are open"}');
      w.tabs[0].add ('checkbox {text: "Show Recent Files  workspace when opening a file"}');
      w.tabs[0].add ('panel {text: "Page numbering", preferredSize: [-1, 80]}');
      w.tabs[0].add ('panel {text: "Font downloading and embedding", preferredSize: [-1, 80]}');
      w.tabs[0].add ('panel {text: "Object editing", preferredSize: [-1, 150]}');
      w.tabs[0].add ('panel {text: "When placing or pasting content", preferredSize: [-1, 80]}');
      with (w.tabs[0]) {
        with (add ('group {alignment: "center"}')) {
          add ('button {text: "Reset al warning dialogs"}');
        }
```



```
    }

  w.tabs[1] = w.tabGroup.add ('group');
    w.tabs[1].add ('statictext {text: "Interface"}');
    w.tabs[1].add ('panel {preferredSize: [-1, -10]}');
    w.tabs[1].add ('panel {text: "Appearance", preferredSize: [-1, 80]}');
    w.tabs[1].add ('panel {text: "Cursor and gesture options", preferredSize: [-1, 150]}');
    w.tabs[1].add ('panel {text: "Panels", preferredSize: [-1, 150]}');
    w.tabs[1].add ('panel {text: "Options", preferredSize: [-1, 100]}');

  w.tabs[2] = w.tabGroup.add ('group');
    w.tabs[2].add ('statictext {text: "UI Scaling"}');
    w.tabs[2].add ('panel');
    w.tabs[2].add ('panel {text: "Options", preferredSize: [-1, 200]}');

w.buttons = w.add ('group {alignment: "right"}');
  w.buttons.add ('button {text: "OK"}');
  w.buttons.add ('button {text: "Cancel"}');

for (var i = 0; i < w.tabs.length; i++) {
  w.tabs[i].orientation = 'column';
  w.tabs[i].alignChildren = 'fill';
  w.tabs[i].alignment = ['fill','fill'];
  w.tabs[i].visible = false;
}

w.stubs.onChange = showTab;

function showTab () {
  if (w.stubs.selection !== null) {
    for (var i = w.tabs.length-1; i >= 0; i--) {
      w.tabs[i].visible = false;
    }
    w.tabs[w.stubs.selection.index].visible = true;
  }
}

w.onShow = function () {
  w.stubs.selection = 1;
  showTab;
}

w.show();
```



progressbar

The **progressbar** control, unsurprisingly, is used to display a progress bar so that the script's user gets an idea of how long a script will run. **progressbar** takes four parameters:

```
w.add ("progressbar", undefined, start, stop);
```

in which **start** and **stop** are the start and stop values of the bar itself, corresponding to the first and last items of whatever you're processing. The start value will usually be 0, while stop could be, for instance, the index of last array element if you're dealing with an array. Here is an example:

```
var found = new Array (50);

var w = new Window ('palette');
  w.pbar = w.add ('progressbar', undefined, 0, found.length);
  w.pbar.preferredSize.width = 300;
w.show();

for (var i = 0; i < found.length; i++){
  w.pbar.value = i+1;
  $.sleep(20); // Do something useful here
}
```

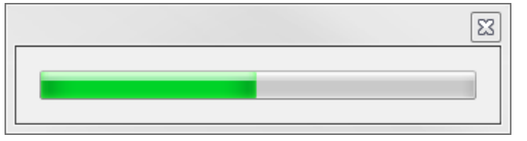
Note that because we start counting at 1, we write the value of of the bar as **i+1** to make sure that the bar displays before the first item is processed, and that the bar fills up completely. If you don't, you'll see that the bar doesn't fill its container so that it might look as if the last item in the array isn't processed.

In InDesign scripts, many arrays are processed from back to front. If you use a progress bar in such a script and use the above code, then the bar will be updated the wrong way round, which looks silly. To get the bar to display from left to right as it should, update it using the following code:

```
for (var i = found.length-1; i > -1; i--){
  w.pbar.value = found.length-i;
  $.sleep(20);
}
```

On a Mac you can create vertical progress bars simply by making their width smaller than their height. Unfortunately, on PCs running Windows this isn't possible (you can create a vertical bar, but the progress indication will always be left-to-right).

Note: in scripts that use progress bars, you cannot set **app.scriptPreferences.enableRedraw** to **false**. If you do, the progress bar doesn't display correctly. Across InDesign versions and operating systems, the situation is a bit complicated. See Ariel Walden's excellent overview here.



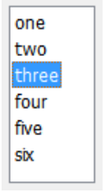
```
var w = new Window ("dialog");
  w.orientation = "row";
  pb1 = w.add ("progressbar", undefined, 0, 100);
  pb2 = w.add ("progressbar", undefined, 0, 100);
  pb3 = w.add ("progressbar", undefined, 0, 100);
  pb1.preferredSize = pb2.preferredSize = pb3.preferredSize = [20, 300];
  pb1.value = 20;
  pb2.value = 60;
  pb3.value = 40;
w.show();
```

There are more sophisticated ways of applying progress bars, of which Marc Autret's script is an excellent example.

Lists as progress indicators

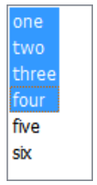
ScriptUI has a control **progressbar**, but using a list you can create a more informative kind of progress indication. For example, if you're processing several files, you can display the names of these files in a list, and use a highlight to show which file is being processed. Here is the script:

```
var list = ["one", "two", "three", "four", "five", "six"];
var hlights = highlight_list (list);

for (var i = 0; i < list.length; i++) {
  hlights.children[0].selection = i;
  hlights.show ();
  // user functions
  $.sleep (400);
}
hlights.close();

function highlight_list (array) {
  var w = new Window ("palette", undefined, undefined, {borderless: true});
  w.margins = [5,5,5,5];
  w.add ("listbox", undefined, array);
  return w;
}
```



The function **highlight_list** initialises the window but doesn't show it yet. In the for-loop, we address list items by using **hlights.children[n]**, where **n** is the loop counter. In each iteration the window is shown using **show()** to show the change. (I don't know why it's necessary to do it like this, but it was the only way to make it work; using **layout()** doesn't seem to work here.)

A small change in the definition of the listbox makes for a slightly different apprearance of the window:




```
w.add ("listbox", undefined, array, {multiselect: true});
```

By adding **{multiselect: true}** to the listbox definition, processed items remain highlighted.

Counters as progress indicators

The simplest progress indicator is a window that shows you the value of the counter of a for-loop, for example. When I suspect that an operation may take a bit longer than I care to wait for, I use this method because it's so easy to write. Here's an example:

```
#targetengine "session";
var counter = new Window("palette");
counter.prompt = counter.add("statictext",[0,0,80,20]);
counter.show();
var cells = app.activeDocument.allCellStyles;
for(var i = cells.length-1; i > 0; i--) {
  counter.prompt.text = String(i);
  cells[i].verticalJustification = VerticalJustification.bottomAlign;
}
counter.close();
```

Lines 2 to 4 define and show the counter window, which is a palette with just one statictext control, which is updated at every iteration of the for-loop. By counting down, the display starts at the highest value, working its way down to 1.

image

Image controls take a file object as their contents parameter. Here is an example:


```
var w = new Window ("dialog", "Bouquet");
  var flowers = w.add ("image", undefined, File ("/d/scriptui/bouquet.jpg"));
w.show ();
```

It's not possible to position images absolutely (using **.location** = [**x,y**]), but they can be positioned with the alignment properties left, right, etc. Nor is it possible to scale images just like that: they're always shown at their native size. Setting a smaller size merely crops the image:

```
var w = new Window ("dialog", "Bouquet");
  var flowers = w.add ("image", undefined, File ("/d/scriptui/bouquet.jpg"));
  flowers.size = [50,50];
w.show ();
```

Resizing images

I said "not just like that" because as Marc Autret points out, with a prototype extension of the **Image** object it *is* possible to resize images. Marc's brilliant extension is added at the beginning of the previous script:

```
Image.prototype.onDraw = function()
  { // written by Marc Autret
  // "this" is the container; "this.image" is the graphic
  if( !this.image ) return;
  var WH = this.size,
    wh = this.image.size,
    k = Math.min(WH[0]/wh[0], WH[1]/wh[1]),
    xy;
  // Resize proportionally:
  wh = [k*wh[0],k*wh[1]];
  // Center:
  xy = [ (WH[0]-wh[0])/2, (WH[1]-wh[1])/2 ];
  this.graphics.drawImage(this.image,xy[0],xy[1],wh[0],wh[1]);
  WH = wh = xy = null;
  }

var w = new Window ("dialog", "Bouquet");
  var flowers = w.add ("image", undefined, File ("/d/scriptui/bouquet.jpg"));
  flowers.size = [50,50];
w.show ();
```

Note: If an image does not redraw properly in Photoshop CS5 or later, try using app.refresh(); This is part of Photoshop's object model, not ScriptUI's.

slider

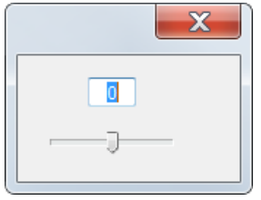
A slider is just that: the familiar slide bar with a control which you move with the mouse. Most sliders use some text field to display the slider's value as it being moved around; you use the **onChanging** callback for that, as illustrated in the following script.

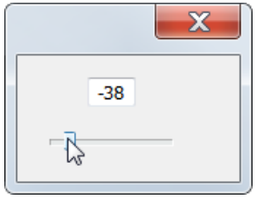
```
var w = new Window ("dialog");
  var e = w.add ("edittext", undefined, 50);
  var slider = w.add ("slider", undefined, 50, 0, 100);
  slider.onChanging = function () {e.text = slider.value;}
w.show ();
```

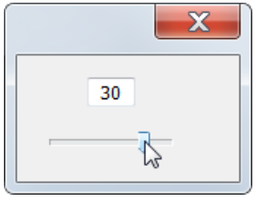
The slide bar takes three numeric parameters: in the example, 0 and 100 are the minimum and maximum values, 50 is the value used when the window is drawn.

Some sliders use negative values. In Photoshop you'll find many examples, such as the Brightness/Control and the Exposure dialogs. By default, ScriptUI's sliders use positive values only, but you can easily get around that, as shown in the following script (we use a slightly different format here, the so-called resource format, for clarity):

```
var w = new Window ('dialog');
  var value = w.add ('edittext {text: 0, characters: 3, justify: "center", active: true}');
  var slider = w.add ('slider {minvalue: -50, maxvalue: 50, value: 50}');
  slider.onChanging = function () {value.text = slider.value - 50}
  value.onChanging = function () {slider.value = Number (value.text) + 50}
w.show();
```







You would think that {**minvalue: -50, maxvalue: 50, value: 0**} would place the slider's handle in the slider's centre, but it doesn't, so you have to use 50 as the start value. In the example we use an edittext control to display the slider's value, and that field can be used to set the slider's value, too, because we added a callback (**value.onChanging**) to the edittext control.

On Macs, but not on Windows, you can create a vertical slider simply by setting its width to a smaller value than its height:

```
var w = new Window ("dialog");
  var slider = w.add ("slider", undefined, 0, 100);
  slider.size = "width: 30, height: 300";
w.show ();
```

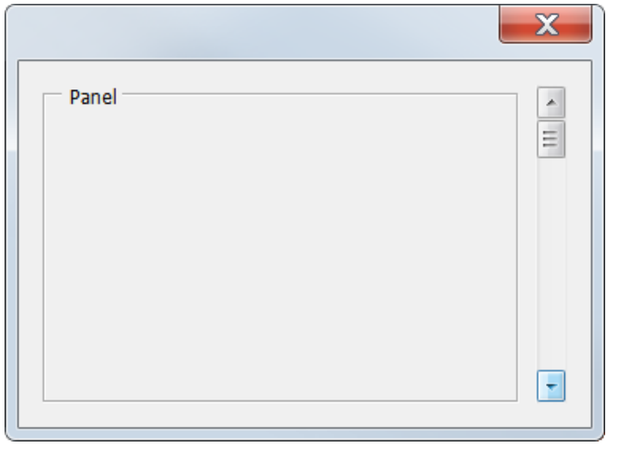
scrollbar

Scrollbars are added automatically to controls of type **edittext** and **listbox** when the text or the list items don't fit their containers; see the examples on pages 11 and 27, respectively.

If you want to add scrollbars to any other control, you're very much on your own in that you have to code all events yourself. This is not so straightforward, witness the daunting code produced by Marc Autret and Bob Stucky in Adobe's scripting forum in this thread. Gerald Singelmann posted a more accessible example in his blog. Below, after some scrollbar basics, I outline a general approach to scrolling panels.

A scrollbar can be placed horizontally or vertically. Its orientation is determined by its coordinates: if the height is bigger than the width, as in the script below, the bar is shown vertically; for horizontal scrollbars, use a width bigger than the height.

```
var w = new Window("dialog");
  var g = w.add ("group");
    var panel = g.add("panel", [0,0,300,200], "Panel");
    var sbar = g.add ("scrollbar", [0,0,20,200]);
w.show();
```



The bar's width is set to 20 and its height to 200: this creates a vertical scrollbar with the same height as the panel.

The scrollbar's value

You manipulate the scrollbar in the familiar way: use the arrow buttons at the top and bottom of the bar; move the slider up or down; or click on the bar between the slider and one of the arrow buttons.

Any of these manipulations returns a value which corresponds with the position of the slider on the bar. You capture these values with a callback, as shown in the following script, which outputs values to the ESTK's console:

```
var w = new Window("dialog");
  var g = w.add ("group");
    var panel = g.add("panel", [0,0,300,200]);
    var sbar = g.add ("scrollbar", [0,0,20,200]);
    sbar.onChanging = function () {$.writeln (sbar.value)}
w.show();
```

Notice that when you use the up and down buttons, integer values are returned. But if you use the slider, the bar's value is usually a fraction.

Scrollbars have a minimum and a maximum value. The defaults are 0 and 100. A scrollbar's default value when the window is drawn is 0, which can be changed by setting that value, as follows:

```
sbar.value = 50;
```

If you want to set the minumum and maximum values as well, you can set these and the default value in one line of code:

```
  var sbar = panel.add ("scrollbar", [0,0,20,200], 0, 0, 60);
```

where **0, 0, 60** are the default, minimum, and maximum values, respectively.

stepdelta

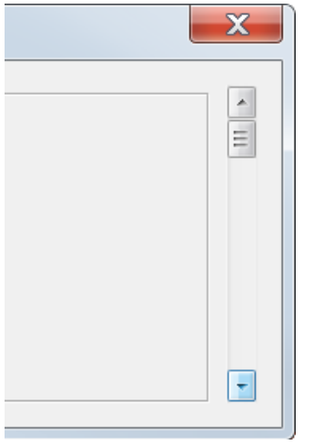
When you click an arrow button, the scrollbar's value changes by 1 by default. So in a scrollbar which uses the default minimum and maximum of 0 and 100, you need to press the down button 100 times to get the slider to the bottom of the bar. To change that, set the bar's stepdelta value:

```
sbar.stepdelta = 10;
```

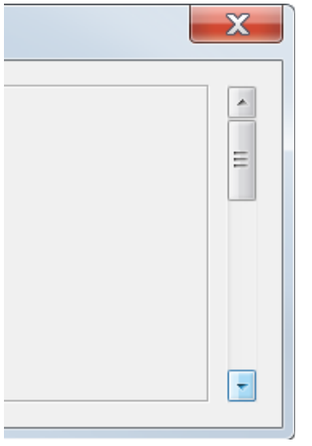
Now you need just ten clicks to move the slider from top to bottom. And with a stepdelta of 100, one press moves the slider from the top to the bottom of the scrollbar.

jumpdelta

The jumpdelta value determines in how many steps the slider moves up or down when you click between the slider and the up or down button. The jumpdata therefore also determines the size of the slider. The default value is 20, which means that in a scrollbar with the default min and max values of 0 and 100, you click five times to move the slider from one end to the other. With a jumpdata of 50, you need just two clicks.



jumpdelta = 20



jumpdelta = 50

Scrolling panels and groups

To scroll a panel (or a group), what you do is move that panel behind the window. You create a panel that is larger than its container, and scrolling a panel is therefore like moving the whole thing so that you get a different view on it – hence the name viewport approach.

The following script creates a group (**scrollGroup**) with 36 lines, and sets a limit on the window's height so that much of the group's content is not visible. When you scroll down using the scroll bar, you move the group up, providing a new window on the group. Thus, in this case, the changing scrollbar value is used to change the vertical position of the group. (This script was inspired by one of Vlad Vladila's scripts and replaced my own earlier clumsy effort.)

```
var w = new Window('dialog');
  w.maximumSize.height = 300;

  var panel = w.add ('panel {alignChildren: "left"}');
    var scrollGroup = panel.add ('group {orientation: "column"}');
    for (var i = 0; i <= 35; i++) {
      scrollGroup.add ('statictext', undefined, 'Label ' + i);
    }
```

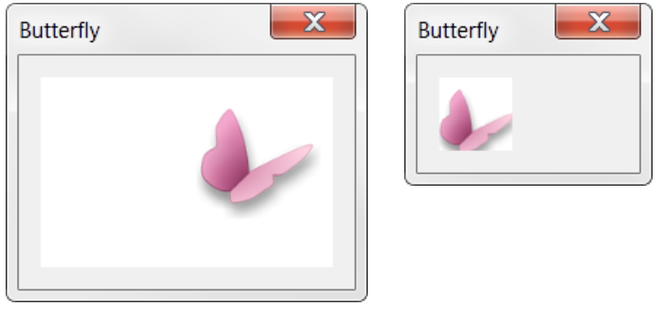
```
    var scrollBar = panel.add ('scrollbar {stepdelta: 20}');

    // Move the whole scroll group up or down
    scrollBar.onChanging = function () {
      scrollGroup.location.y = -1 * this.value;
    }

    w.onShow = function() {
      // Set various sizes and locations when the window is drawn
      panel.size.height = w.size.height-20;
      scrollBar.size.height = w.size.height-40;
      scrollBar.size.width = 20;
      scrollBar.location = [panel.size.width-30, 8];
      scrollBar.maxvalue = scrollGroup.size.height - panel.size.height + 15;
    };

  w.show();
```

Note: On Mac OS the Mondo rendering engine was introduced in Photoshop CC 2015. As a result, JPG images are no longer accepted, all images must be PNG. According to Davide Barranca this will be fixed (see cc2015-survival-guide, point 8).

flashplayer

Flashplayer controls are similar to image controls. And like images, if you set their size, you just get a crop or an oversized frame:

```
var w = new Window ("dialog", "Butterfly");
  var flash = w.add ("flashplayer", undefined, File ("/d/scriptui/wave.swf"));
w.show ();
```



The controls for stopping and starting movies in CS3 were discontinued in CS4. You need ActionScript and Flash/Flex to regain any control over movie clips. See Loïc Aigon's post at here for an interesting discussion on the interaction of Flash, Flex, and ScriptUI.

The Object-Model Viewer in the ESTK is in fact a flashplayer control in a ScriptUI window. See the script 35omvUI.jsx, which (in CS5/CS6 on Windows) lives in Adobe/Adobe Utilities - CS5(CS6)/ExtendScript Toolkit CS5(CS6)/Required, from CC, in /Adobe/Adobe ExtendScript Toolkit CC. This script is educational not only because of the flashplayer control used in it.

(The flashplayer control was discontinued in CC2017.)

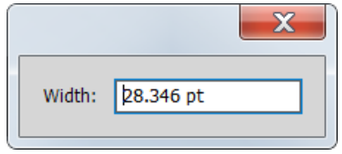
Measurement control

Earlier we mentioned that ScriptUI has just one input method, edittext, which, as the name suggests, accepts text, which in effect means that it accepts any input. InDesign's classic dialog system, on the other hand, has controls for text,

measurement units, reals, integers, and percentages. However, it's possible to script those controls. Here we give an example of a measurement control.

The control behaves just like InDesign's measurement controls: type a value followed by some measurement unit and the script shows that value in the document's units. For example, if your document's units are points and you enter *10mm*, this will be displayed as *28.346 pt*. Enter a value without a unit and that value will be displayed simply adding the document's unit. One restriction of this script is that it doesn't let you do any arithmetic: in InDesign's measurement controls you can add something like *12p*3* (3 times 12 picas), but the script doesn't support that.

```
var w = new Window ('dialog');
  var grp = w.add ('group');
    grp.add ('statictext {text: "Width: "}');
    var width = grp.add ('edittext {characters: 12, active: true}');
    width.text = convert_units ("10mm", documentUnits());
w.show();

function convert_units (n, to) {
  var unitConversions = {
    'pt': 1.0000000000,
    'p': 12.0000000000,
    'mm': 2.8346456692,
    'in': 72.00000000,
    'ag': 5.1428571428,
    'cm': 28.3464566929,
    'c': 12.7878751998,
    'tr': 3.0112500000
  }
  var obj = fix_measurement (n);
  var temp = (obj.amount * unitConversions[obj.unit]) / unitConversions[to];
  return output_format (temp, to)
}
```

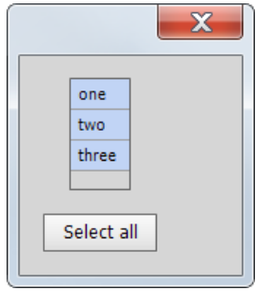


```
function output_format (amount, target) {
  // Add the target unit to the amount, either suffixed pt, ag, mm, cm, in, or infixed p or c
  // The decimals, no trailing zeros
  amount = amount.toFixed(3).replace(/\.?0+$/, '');
  if (target.length == 2) { // two-character unit: pt, mm, etc.
    return String (amount) + ' ' + target;
  } else {// 'p' or 'c'
    // calculate the decimal
    var decimal = (Number (amount) - parseInt (amount)) * 12;
    // return the integer part of the result + infix + formatted decimal
    return parseInt (amount) + target + decimal;
    }
}

function fix_measurement (n) {
  n = n.replace(/ /g, '');  // Delete all spaces
  n = n.replace (/^([pc])/, '0$1');  // Change p3 to 0p3
  // Infixed 'p' and 'c' to decimal suffixes: 3p6 > 3.5 p
  n = n.replace (/(\d+)([pc])([.\d]+)$/, function () {
    return Number (arguments[1]) + Number (arguments[3]/12) + arguments[2]}
  );
  // Split on unit
  var temp = n.split (/(ag|cm|mm|c|pt|p|in)$/);
  return {
    amount: Number (temp[0]),
    unit: temp.length === 1 ? doc_units() : temp[1]
  };
}

function documentUnits () {
  switch (app.documents[0].viewPreferences.horizontalMeasurementUnits){
    case 2051106676: return 'ag';
    case 2053336435: return 'cm';
    case 2053335395: return 'c';
    case 2053729891: return 'in';
    case 2053729892: return 'in';
    case 2053991795: return 'mm';
    case 2054187363: return 'p';
    case 2054188905: return 'pt';
    }
}
```

Simulating keypresses

Suppose you have a script that contains a list and a button to select all items in that list. The code will include an onClick function that selects the whole list when that button is pressed. Now suppose that when the scripts starts, all items in the list should be selected. You don't want to repeat the code to select that list on startup: you can tell that button to consider itself pressed by including adding the **.notify()** method to the button:

```
var w = new Window ('dialog');
  var list = w.add ('listbox', undefined, ['one', 'two', 'three'], {multiselect: true});
  var select_all = w.add ('button', undefined, 'Select all');

  select_all.onClick = function (){
    var selected = [];
    for (var i = 0; i < list.items.length; i++){
      selected[i] = list.items[i];
    }
    list.selection = selected;
  }

  select_all.notify();

w.show();
```

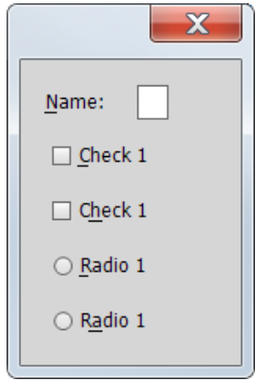


The notify() method can be used for various other controls as well.

Adding shortcut keys to controls

To select certain controls quickly or to move the cursor to a certain edit field, you can define shortcut keys for those controls. Unfortunately, this is Windows-only. Here is an example with several shortcut keys:

```
var w = new Window ("dialog");
  var grp = w.add("group");
    var st = grp.add("statictext", undefined, "&Name:");
    var txt = grp.add("edittext"); txt.shortcutKey = "n";
  var c1 = w.add ("checkbox", undefined, "&Check 1"); c1.shortcutKey = "c";
  var c2 = w.add ("checkbox", undefined, "C&heck 1"); c2.shortcutKey = "h";
  var r1 = w.add ("radiobutton", undefined, "&Radio 1"); r1.shortcutKey = "r";
  var r2 = w.add ("radiobutton", undefined, "R&adio 1"); r2.shortcutKey = "a";
w.show();
```



Naturally, to make it clear to the user that certain shortcut keys can be used, those letters should be cued. In ScriptUI you do that by placing an **&** before the

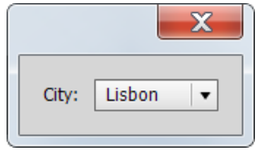
letter that you want marked, which adds an underscore; see the screenshot. *Note*: The underscores of most controls become visible only after pressing the Alt key.

In almost all cases, shortcut keys have the same effect as mouse clicks. For instance, pressing Alt+n in the above script activates the edit field at Name, Alt+c toggles the first checkbox, etc.

Control titles

So far we've added control titles by placing them in a separate statictext control, as follows:

```
var w = new Window ('dialog');
  var grp = w.add ('group');
    grp.add ('statictext', undefined, 'City: ');
    var drop = grp.add ('dropdownlist', undefined, ['Lisbon', 'Lancaster']);
      drop.selection = 0;
w.show();
```



At some stage ScriptUI acquired a method to add titles. The above script can be changed to use that property as follows:

```
var w = new Window ('dialog');
  var drop = w.add ('dropdownlist', undefined, ['Lisbon', 'Lancaster']);
    drop.selection = 0;
    drop.title = 'City:';
w.show();
```

In addition, titles can be formatted using the **titleLayout** property. For example, to place the title after the control, use this code:

```
var w = new Window ('dialog');
  var drop = w.add ('dropdownlist', undefined, ['Lisbon', 'Lancaster']);
    drop.selection = 0;
    drop.title = 'City';
    drop.titleLayout = {alignment: ['right', 'center']};
w.show();
```

Unfortunately, the title property is available only for controls of type **dropdownlist**, **flashplayer**, **iconbutton**, **image**, and **tabbedpanel**, not, crucially for **edittext** and **list** controls. Since I most often need titles for edittext controls, and since I always place titles on the left-hand side of controls, I use a different

method: I continue to define titles as statictext controls, and place the statictext as a property of a group:

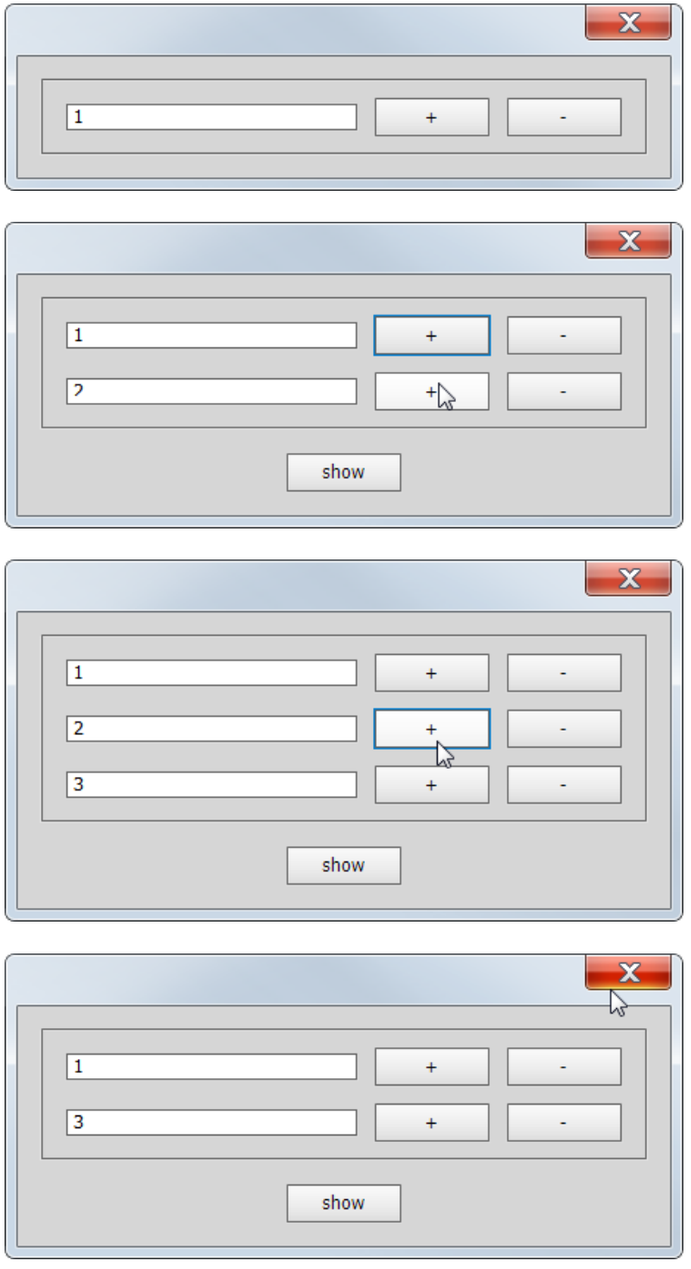
```
var w = new Window ('dialog');
  var grp = w.add ('group {_: StaticText {text: "City:"}}');
    var drop = grp.add ('dropdownlist', undefined, ['Lisbon', 'Lancaster']);
    drop.selection = 0;
w.show();
```

This method uses a so-called resource string, which makes for compact code (see p. 106 for more details on resource strings). For consistency I use this method not only for edittext controls, but for all controls.

Adding and removing controls dynamically

It is possible to add and remove controls after a window has been drawn and have the window resized to accommodate the added and removed items. This is handled by the layout manager, which is invoked by the **.layout** function. The following script, written by Gerald Singelmann, illustrates this. See also some links in the resources in "Resize windows" on page 113.

```
// Slightly adapted from G. Singelmann's script, see here.
var win = new Window ("dialog");

  var maingroup = win.add ("panel {orientation: 'column'}");
  add_row  (maingroup);

  var show_btn = win.add ("button", undefined, "show");
  show_btn.onClick = function () {
  var txt = "";
  for  (var n = 0; n < maingroup.children.length; n++) {
    txt += maingroup.children[n].edit.text + "\n";
  }
  alert ("Rows: \n" + txt);
  }

win.show ();

function add_row (maingroup) {
  var group = maingroup.add ("group");
  group.edit = group.add ("edittext", ["", "", 200, 20], maingroup.children.length);
  group.plus = group.add ("button", undefined, "+");
  group.plus.onClick = add_btn;
```



```
    group.minus = group.add ("button", undefined, "-");
    group.minus.onClick = minus_btn;
    group.index = maingroup.children.length - 1;
    win.layout.layout (true);
}

function add_btn () {
    add_row (maingroup);
}

function minus_btn () {
    maingroup.remove (this.parent);
    win.layout.layout (true);
}
```

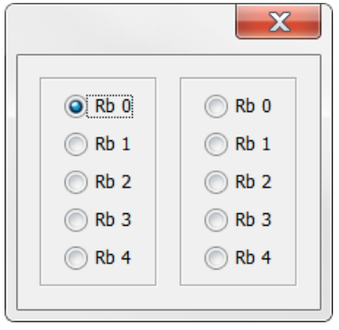
Other aspects of windows, too, can be changed dynamically. On p. 105 is an example of changing a container's orientation dynamically.

Labelling controls

Controls can be labelled so that you can later identify them. This works much like using script labels on objects outside of ScriptUI. When you add a label to a control, you in fact simply add a text property to it.

We saw an example earlier which defined two event listeners on two panels with radio buttons so that it seemed as if those radio buttons were in one group. This still required two listeners. I mentioned there that these panels could be grouped together, in which case one event listener would suffice. This would be a more complex listener in that it should be able to tell which panel was clicked on. This is where labels come in; the following script illustrates:

```
var w = new Window ("dialog");
    var radiogroup = w.add ("group");
        var panel1 = radiogroup.add ("panel"); panel1.label = "p1";
            for (var i = 0; i < 5; i++) {panel1.add ("radiobutton", undefined, "Rb "+i);}
        var panel2 = radiogroup.add ("panel"); panel2.label = "p2";
            for (var i = 0; i < 5; i++) {panel2.add ("radiobutton", undefined, "Rb "+i);}
        panel1.children[0].value = true;

    radiogroup.addEventListener ("click", function (event)
        {
        if (event.target.parent.label == "p1")
            {
```



```
        for (var i = 0; i < panel2.children.length; i++)
          panel2.children[i].value = false;
        }
      else
        {
        for (var i = 0; i < panel1.children.length; i++)
          panel1.children[i].value = false;
        }
      } // if
  );

w.show();
```

The two panels are labelled using a property **label**, but you can use any name you like, such as panel1.id = "left". The event listener then checks the value of the label and acts according to it: if panel1 was clicked, panel2's buttons are disabled, if panel2 was clicked, panel1's children are unmarked.

Labels can be used on all types of control. For a special type of label – **name** – see the section **Communication between windows**, below.

Finding windows

While a dialog is displayed, you can't do anything else: InDesign sits there waiting patiently for you to press OK (or Cancel) so it can continue and go about its business. But as we've seen, palettes are different: you can create a palette and then go and do something different: edit a document, generate an index – anything at all. Even run another script.

Palettes are useful, for instance, to display things that aren't so easy to access in InDesign. The state of the No break attribute, for example, can be checked only by opening the Characters panel, then opening the panel's flyout (it doesn't even show in the Control panel). This is tedious and we would like a panel that displays the state of No break of the selected text and updates automatically when the state of No break changes.

Such a script works as follows in general: (1) create a panel; (2) create an event listener; and (3) create a function which, every time something changes in a document checks if that change involves No break and if it does, looks for the palette and updates it.

On page 15 we gave a script that can be used display the state of the No break character attribute (it's repeated here in the sidebar). That version doesn't update automatically, you have to click a button to update the displayed state.

```
// The original script
#targetengine miscellaneous;
var w = new Window ('palette')
        w.nobreak = w.add ('statictext {text: "No break: ", characters: 10, justify:
"center"}');
        w.button = w.add ('button {text: "Update"}');

        function updateStatus() {
          var state = '';
          if (app.selection.length > 0) {
            var str = app.selection[0].insertionPoints.everyItem().noBreak.join();
            if (str.indexOf ('false') > -1 && str.indexOf ('true') > -1) {
              state = '0/1';
            } else if (str.indexOf ('false') > -1) {
              state = '0';
            } else {
              state = '1';
            }
          }
          w.nobreak.text = 'No break: ' + state;
        }

        w.onShow = w.button.onClick = updateStatus;

        w.show();
```

The version that updates automatically using an event listener is listed below on the next page.

The script first checks if a window exists with the name **Attributes** using the **find**() method:

```
var windowName = 'Attributes';
var w = Window.find ('palette', windowName);
```

This method takes two obligatory parameters: the type of window we're looking for (here, a palette) and the window's name – you must therefore name a window if you want to find it later.

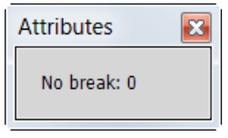
If the window is not found, a new one is created and an event listener is installed that calls the **updateStatus**() function whenever the selection changes. (To achieve that, note that we took that function out of the window's definition.) If the window is found, it's simply displayed and its status updated.

With this script installed, whenever you select some text, the state of No break is shown in the window: 0 if No break is not applied, 1 when it is applied, and 0/1 to indicate a mixed state.

Note: to debug and edit a script that looks for an existing window, you need to change the name on every new run because if you don't, the script uses the window set in the previous run and your changes won't execute.

A more elaborate version of the script, with which the no break state can be changed, is shown on p. 93.

```
#targetengine miscellaneous;
var windowName = 'Attributes';
var attr = Window.find ('palette', windowName);
if (attr === null){
  attr = createWindow (windowName);
  app.addEventListener ('afterSelectionChanged', updateStatus);
}
attr.show();

function updateStatus() {
  var w = Window.find ('palette', windowName);
  if (w !== null) {
    try {
      var str;
      var state = '';
      if (app.selection.length > 0) {
        var str = app.selection[0].textStyleRanges.everyItem().noBreak.join();
```



```
            if (str.indexOf ('false') > -1 && str.indexOf ('true') > -1) {
              state = '0/1';
            } else if (str.indexOf ('false') > -1) {
              state = '0';
            } else {
              state = '1';
            }
          }
          w.nobreak.text = 'No break: ' + state;
        } catch (_){}
      }
    }

    function createWindow (title) {
      var w = new Window ('palette');
        w.text = title;
        w.nobreak = w.add ('statictext {text: "No break: ", characters: 10}');
        w.onShow = updateStatus;
      return w;
    }
```

Finding controls

Apart from windows, you can find controls inside windows as well. Like finding windows, you have to name a control if you want to find it later. For example, suppose you have a palette that defines an edittext control, as in:

```
#targetengine "session";
var console = createConsole();
function createConsole () {
  var w = new Window ("palette", "Console");
    w.add ("edittext", [0,0,400,400], "", {multiline: true, name: "consoleText"});
  return w;
}
```

There's no need to display the window yet, we do that when we actually access it. A function in the same script or one in a different script gets access to the name control as follows:

```
function writeConsole (txt) {
  var w = Window.find ("palette", "Console");
  if (w !== null) {
    if (!w.visible) w.show();
    w.findElement ("consoleText").text = txt;
    }
  }
```

This function locates the palette using **Window.find**(), using two obligatory parameters: the window type (here, a palette) and the window's title. Then, if the palette was found, it checks to make sure that the console is visible, and if it isn't, it displays it. Finally, the function uses **findElement**() to locate the edittext control and adds the message to whatever is in the text field.

Closing windows

Remember that in a script that uses a palette, you need to include the **#targetengine** directive. In other words, a palette is always created in a persistent engine. Now, when a script that runs in a persistent engine finishes, any variables created by the script remain in memory (though accessible only from within that engine). The same goes for palettes: even if you close a palette using **close**(), it remains in memory.

You should therefore always check if a particular palette exists before you create it, along the following lines:

```
function createMessageWindow () {
  var w = Window.find ("palette", "Message");
  if (w == null) {
    w = new Window ("palette", "Message");
    w.mess = w.add ("statictext", [0,0,300,20], "");
  }
  w.show();
  }
```

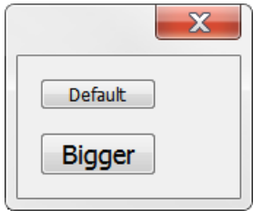
We use the **Window.find**() method to locate a particular palette, and if we can't find it, we create it.

All this applies to dialogs, too, but only if you create them in a persistent engine. A dialog that's closed in a script that does not run in a persistent engine is deleted when you close it.

Fonts

The default type size used in ScriptUI windows is much too small for me, so I usually make that bigger. The typeface doesn't bother me (it's usually Tahoma or something similar), it's size that matters. The easiest way to set a control's type size is as follows:

```
var w = new Window ("dialog");
  button1 = w.add ("button", undefined, "Default");
  button2 = w.add ("button", undefined, "Bigger");
  button2.graphics.font = "dialog:18";
w.show ();
```



This sets the text of the second button's type size to 18, using the window's default font. To change the typeface as well, use the following:

```
button2.graphics.font = "Tahoma:18";
```

And to change the font style too, for example, to bold, use this construction:

```
button2.graphics.font = "Tahoma-Bold:18";
```

Another, more elaborate, way of setting a window's font is the **newFont()** method (because it's more elaborate I rarely use it). Here is an example:

```
button2.graphics.font = ScriptUI.newFont ("dialog", "Bold", 18);
```

This type of construction can be used to set the typeface as well:

```
button2.graphics.font = ScriptUI.newFont ("Verdana", "Bold", 18);
```

Both constructions can be used to change the typeface:

```
button2.graphics.font = "Verdana";
button2.graphics.font = ScriptUI.newFont ("Verdana");
```

But this resets the style and the type size to the default. Therefore, if you want to change just the typeface, you need to store the font's size and style and use those values when you change the face:

```
var ft = button2.graphics.font;
button2.graphics.font = "Verdana-" + ft.style + ":" + ft.size;
```

Note: For the font name you must use the font's PostScript name, which is not necessarily the same as the menu name used in InDesign's or Photoshop's interface. If setting a font throws an error, chances are that that font's PostScript name is not the same as its menu name. For example, the PostScript name of the Gill Sans family is GillSans. To find a font's PostScript name, run this one-line script in the ESTK with the Console visible:

Note: Fonts and colours cannot be changed in InDesign CC and later.

Font names are treated differently on Macs and PCs See Marc Autret's article at www.indiscripts.com/post/2012/05/scriptui-fonts-facts

```
app.fonts.item ("Gill Sans").postscriptName;
```

Note: ScriptUI recognises just four style names: Regular, Italic, Bold, and Bold-Italic.

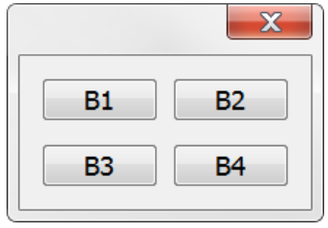
Note: the font object has a (read-only) property **substitute**, but it's not clear what it does. We can guess that it's something like 'if font x can't be found, use font y', but it doesn't work.

Unfortunately, you can set fonts for one control at a time only. If you want to apply a font to all elements in a window or in a group or panel, you need to process that control. I use this function:

```
function set_font (control, font) {
  for (var i = 0; i < control.children.length; i++) {
    if ("GroupPanel".indexOf (control.children[i].constructor.name) > -1)
      set_font (control.children[i], font);
    else
      control.children[i].graphics.font = font;
  }
}
```

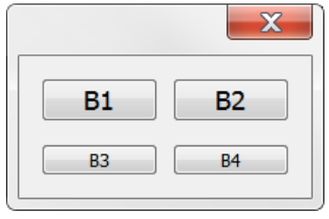
You use it to apply a font to all elements in a window:

```
var w = new Window ("dialog");
  var group1 = w.add ("group");
    var button1 = group1.add ("button", undefined, "B1");
    var button2 = group1.add ("button", undefined, "B2");
  var group2 = w.add ("group");
    var button1 = group2.add ("button", undefined, "B3");
    var button2 = group2.add ("button", undefined, "B4");
  set_font (w, "Tahoma:18");
w.show();
```



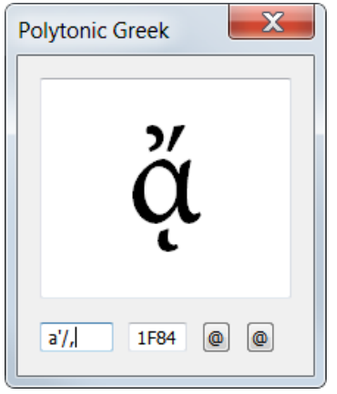
Or to just one group, as in this example:

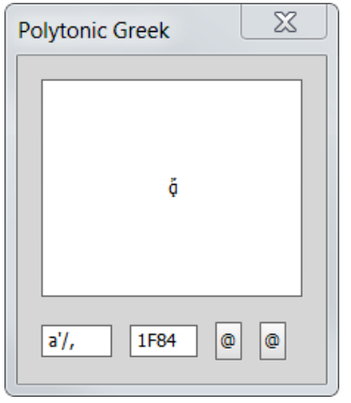
```
set_font (group1, "Tahoma:18");
```



Note: On Windows you can't change the appearance of a window's title.

Unfortunately, everything to do with fonts stopped working in InDesign/ScriptUI CC. This is a mild annoyance in many cases, but it makes some scripts virtually useless. A script I use from time to time to enter polytonic Greek, for example, works by combining keyboard presses to build up a character. In CS6 and earlier that gave you a good idea, as shown on the right. In CC and later, all you get as a preview is a puny little insect, as shown on the far right. (If you're interested in the script, it's here: http://www.kahrel.plus.com/indesign/compose-greek.html.) It is hoped that ScriptUI's font handling will be restored.





Colours

Colours, like fonts, are part of ScriptUI's graphics controls. But whereas fonts are quite well manageable, with colours, brushes, and other graphics features things start to get a bit complicated. We're not greatly helped by the almost complete absence of any clear examples of how to use the various graphics elements. InDesign/the ESTK comes with two scripts that have examples of how to set the foreground and background colours, but that's about it.

The example below shows how to set font, style, size of controls, but also their foreground and background colours. The example was distilled from two sample scripts, ColorSelector.jsx and ColorPicker.jsx (to locate them, search your hard disk; their location differs depending on operating system and InDesign version). These scripts are instructive, and also useful to find colour values. (Another good way of finding colour values is running Rapid ScriptUI; see the resource section, below, for details.)

```
var w = new Window ("dialog");
  var s = w.add ("statictext", undefined, "Static");
  var e = w.add ("edittext", undefined, "Edit");
  var b = w.add ("button", undefined, "Button");
  // The window's backround
  w.graphics.backgroundColor = w.graphics.newBrush (w.graphics.BrushType.SOLID_COLOR,
[0.5, 0.0, 0.0]);
  // Font and its colour for the first item, statictext
  s.graphics.font = ScriptUI.newFont ("Helvetica", "Bold", 30);
  s.graphics.foregroundColor = s.graphics.newPen (w.graphics.PenType.SOLID_COLOR, [0.7,
0.7, 0.7], 1);
  // Font and colours for the second item, edittext
  e.graphics.font = ScriptUI.newFont ("Letter Gothic Std", "Bold", 30);
  e.graphics.foregroundColor = e.graphics.newPen (e.graphics.PenType.SOLID_COLOR, [1, 0,
0], 1);
  e.graphics.backgroundColor = e.graphics.newBrush (e.graphics.BrushType.SOLID_COLOR,
[0.5, 0.5, 0.5]);
```

```
    // Font for the tird control, a button. Can't set colours in buttons
    b.graphics.font = ScriptUI.newFont ("Minion Pro", "Italic", 30);
w.show ();
```

The method that sets the background colours – **newBrush()** – takes two parameters: the type (SOLID_COLOR; the other type, THEME_COLOR, appears not to work if you target InDesign) and the colour as an array of three numbers between 0 and 1 (these are RGB colours). The method for setting the foreground colours here – **newPen()** – takes an additional parameter to set the line width, but as we're using it for applying a colour to a font, line width isn't relevant here (though it must be specified).

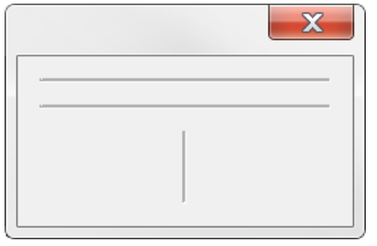
It appears to be impossible to set the colour of buttons. This has been discussed in Adobe's scripting forum, where Dirk Becker provided a work-around.

For a funny and educational application of fonts and colours, see **milligram**'s take on progress bars. For an interesting example of setting some colour interactively, see Xavier's script.

Rules

Rules can be drawn in two ways. You can use ScriptUI's graphics object, which is versatile but complicated, and narrow panels to mimic horizontal and vertical rules, which is much simpler, if rather primitive. You can draw them at a specific point, as in this example:

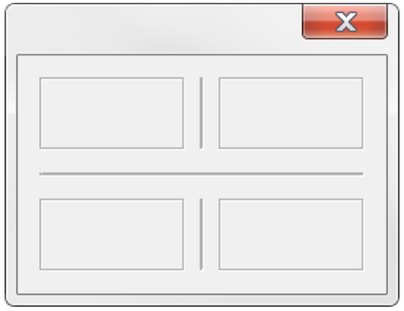
```
var w = new Window ("dialog");
    w.add ("panel", [0,0,200,3]);
    w.add ("panel", [0,20,200,23]);
    w.add ("panel", [100,0,103,50]);
w.show();
```

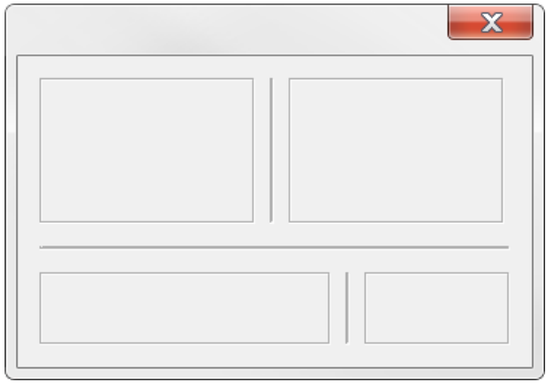


The first line, [0,0,200,3], sets a panel 200 pixels wide and 3 pixels high, so in effect it draws a rule 3 pixels wide; the third line creates a 50-pixel tall and 3-pixel wide rule. Narrow tall panels create vertical rules, shallow wide panels, horizontal ones.

Setting rules with absolute values has a disadvantage in that when you change the window, you almost always have to change the rules as well, and that can be finicky business. To avoid all that you can use ScriptUI's layout manager (always the preferred method): using the alignment attribute **fill**, rules can be made to resize with adjacent panels. Once set up, you have a flexible system. Here is a schematic example.

```
var w = new Window ("dialog");
  w.alignChildren = ["fill","fill"];
  var g1 = w.add ("group");
    g1.alignChildren = ["fill","fill"];
    var p1 = g1.add ("panel");
      p1.preferredSize = [100, 50];      // Use [100,100] to get the second scr. shot
    g1.separator = g1.add ("panel");     // This shows as a vertical line
      // This is the line's width
      g1.separator.minimumSize.width = g1.separator.maximumSize.width = 3;
    var p2 = g1.add ("panel");
      p2.preferredSize = [100, 50];

  w.separator = w.add ("panel");         // This one shows as a horizontal line
    // It says "height", but is again the line's width!
    w.separator.minimumSize.height = w.separator.maximumSize.height = 3;

  var g2 = w.add ("group");
    g2.alignChildren = ["fill", "fill"];
    var p3 = g2.add ("panel");
      p3.preferredSize = [100, 50];      // Use [200,50] to get the second scr. shot
    g2.separator = g2.add ("panel");     // This shows as vertical a line
    g2.separator.minimumSize.width = g2.separator.maximumSize.width = 3;
    var p4 = g2.add ("panel");
      p4.preferredSize = [100, 50];
w.show ();
```





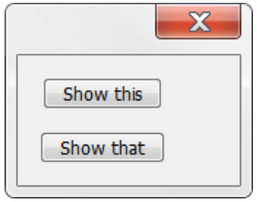
Because the window and the two groups are set to alignment **fill**, when you change an element in a group the rules change accordingly. To see this, change the first bold **[100,50]** in the above script to [100,100], and the second one to [200,50], to get the result in the second screenshot.

(Note: the convoluted w.separator.minimumSize.height = w.separator.maximumSize.height is necessary because objects in ScriptUI don't have separate widths and heights that you can set. So we say that the rules must be at least and at most 3 pixels wide, which is to say exactly 3 pixels.)

Callbacks

Callbacks are built-in methods that monitor events in a dialog: whether buttons are clicked, or a list item is selected, if an edit field is exited, etc. In the CS5 version of the Tools Guide they're listed on pp. 83 and 147. The most frequently used callback is probably **onClick**, which is illustrated in the following script:

```
var w = new Window ("dialog");
  var b1 = w.add ("button", undefined, "Show this");
  var b2 = w.add ("button", undefined, "Show that");

  b1.onClick = function () {$.writeln (this.text + " clicked.")}
  b2.onClick = function () {$.writeln (this.text + " clicked.")}

w.show ();
```

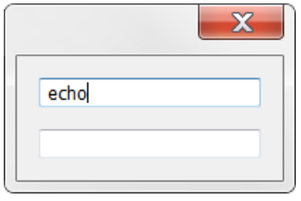


The script displays the dialog in the screenshot; press a button and it prints the name of the button in the console.

The body of the callback's function – which for the b1 button in this example is just {**alert (this.text + " clicked.")**} – can be a function of any complexity. The principle of the other callbacks is essentially the same.

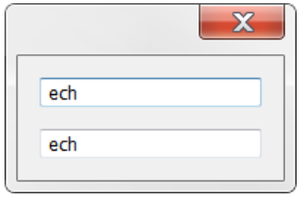
Another frequent callback is **onChange**, which applies to several types of control. The first example shows how you can monitor an edit field.

```
var w = new Window ("dialog");
  var e1 = w.add ("edittext");
  var e2 = w.add ("edittext");
  e1.active = true;
  e1.characters = e2.characters = 20;
  e1.onChange = function () {e2.text = e1.text}
w.show ();
```



The script displays a window with two empty edit fields. Type something in the first field; when you press Enter/Return or Tab, whatever you typed in the first field is (in this example) copied into the second one. Note that onChange is not triggered until you leave the edittext control. To monitor activity while something is entered, you need a different handler, **onChanging**:

```
var w = new Window ("dialog");
  var e1 = w.add ("edittext");
  var e2 = w.add ("edittext");
  e1.active = true;
  e1.characters = e2.characters = 20;
  e1.onChanging = function () {e2.text = e1.text}
w.show ();
```



Now the second edit field is filled while you type.

Lists, too, can be monitored. The following script displays a three-item list, preselecting the first item. Click a list item and it's name is printed in the console.

```
var w = new Window ("dialog");
  var list = w.add ("listbox", undefined, ["one", "two", "three"]);
  list.selection = 0;
  list.onChange = function () {$.writeln (this.selection.text + " selected.")}
w.show ();
```

Clicking anywhere in the list counts as a change in the list. Double-clicking is often useful as well:

```
var w = new Window ("dialog");
  var list = w.add ("listbox", undefined, ["one", "two", "three"]);
  list.selection = 0;
  list.onDoubleClick = function () {$.writeln (this.selection.text + " double-clicked.")}
w.show ();
```

Adding callbacks in loops

Windows can be populated using loops. For example, you could add series of buttons in a loop. Less intuitively, you can add callbacks in loops, too. The following script exemplifies this. It uses a loop to add a bunch of buttons to a window, and then it supplements those buttons with a gaggle of callbacks.

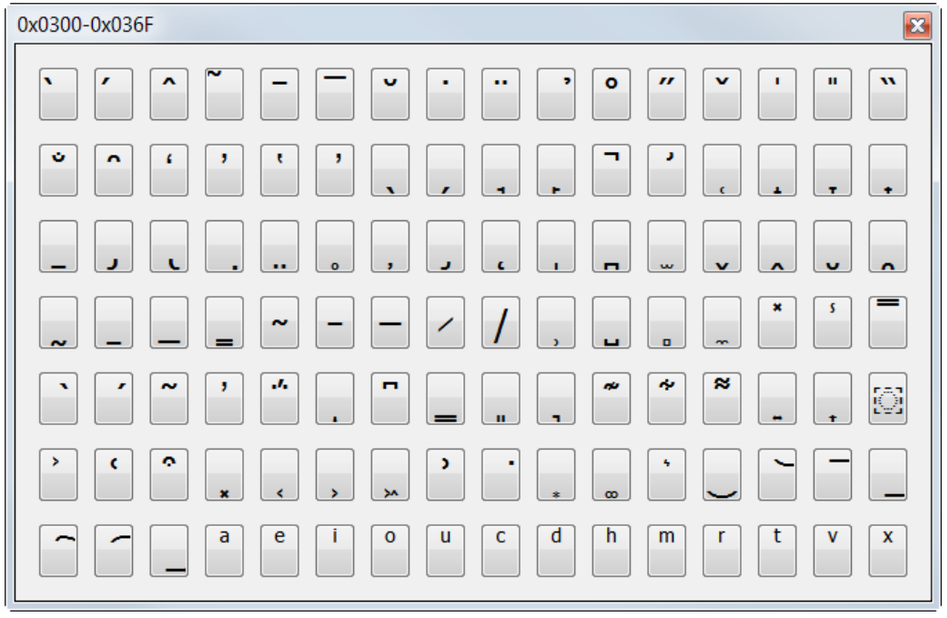
The idea of the script is to facilitate entering accents from the range Combining diacritical marks, and in effect the script is an extension, in a way, of the Glyph panel. The Glyph panel can show several Unicode ranges, but not the combining diacriticals, and as I found myself hunting down those diacritics I thought it useful to have them in a panel.

The first loop creates an object, **buttons**, adding buttons, setting the type size of each button and each button's helptip. The helptip shows the Unicode value associated with a button and of the character inserted when you press the selected button.

The second loop installs a callback for every button. These callbacks are responsible for inserting the character associated with the pressed button. Note that in its current form, the window is dismissed when you click a button. If you would like the window to stay on the screen, all you have to do is to remove **w.close();** in the one-but-last line.

(The example has become pretty useless because the button text can no longer be resized, but it does show how loops can be used to populate a window and to install callbacks.)

```
#targetengine session;

try {
  if (app.selection[0] instanceof InsertionPoint)
    diacritics_panel ();
} catch (_) {
  // Not interested in errors.
};

function diacritics_panel () {
  var w = new Window ("palette", "0x0300-0x036F");
    var bsize = [0,0,40,40]; // Size of the buttons
    var row, key, i, j, buttons = {};
    // The ranges make up the rows
    var ranges = [[0x0300,0x030F],[0x0310,0x031F],[0x0320,0x032F],
      [0x0330,0x033F],[0x0340,0x034F],[0x0350,0x035F],[0x0360,0x036F]];
    for (i = 0; i < ranges.length; i++) {
      row = w.add ("group");
      for (j = ranges[i][0]; j <= ranges[i][1]; j++) {
        key = ("0"+j.toString(16)).toUpperCase();
        buttons[key] = row.add ("button", bsize, String.fromCharCode (j));
        buttons[key].helpTip = key;
      }
    }

    for (i in buttons) {
      buttons[i].onClick = function () {
        app.selection[0].contents = this.text;
        w.close ();
      }
    }
  w.show();
}
```



Note: from CS6, it is no longer necessary to set **resizeable** to **true** in order to display the palette's title. Thus, the window can be created with its title shown as follows:

```
var w = new Window ("palette", "0x0300-0x036F");
```

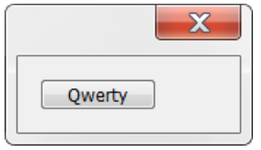
Event handlers

Event handlers are comparable to callbacks in that they monitor what happens in a dialog. They are more flexible, though with this added flexibility comes some complexity. Event handlers are discussed from p. 149 of the Tools Guide. Two examples here for illustration: one that monitors the mouse, another that listens to the keyboard.

Monitoring the mouse

The first example shows how to monitor specific mouse events and some environmental states:

```
var w = new Window ("dialog");
  var b = w.add ("button", undefined, "Qwerty");

  b.addEventListener ("click", function (k) {whatsup (k)});

  function whatsup (p) {
    if (p.button == 2) {$.writeln ("Right-button clicked.")}
    if (p.altKey) {$.writeln ("Alt key pressed.")}
    $.writeln ("X: " + p.clientX);
    $.writeln ("Y: " + p.clientY);
  }
w.show ();
```



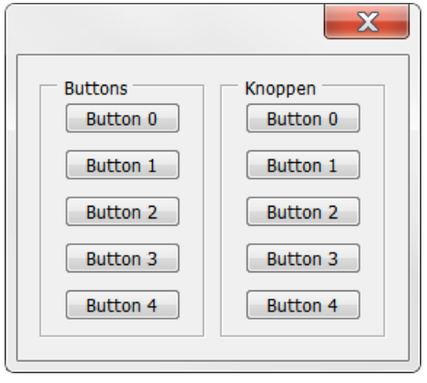
The event handler monitors the mouse and the button and whenever you click the button, it executes the function defined in the handler, and can be seen as **onClick** with some more possibilities.

The four properties whose value the script prints in the console are just a selection; check the Tools Guide (p. 153) for a complete list. The example shows that you can check whether the right button was clicked and whether the Alt key was pressed when you clicked. **clientX** and **clientY** return values that tell where on the control you clicked, so that you can tell, for example, whether you clicked the left or the right half of the button.

Determining which button is pressed

If you have a group of buttons, you don't want to list a battery of onClick callbacks to find which button was pressed. Instead, you can define an event listener which monitors the group and if any of the buttons is pressed, returns some propert of the button.

```
var w = new Window ("dialog");
  var buttongroup = w.add ("group");
  var panel1 = buttongroup.add ("panel", undefined, "Buttons");
    for (var i = 0; i < 5; i++)
      panel1.add ("button", undefined, "Button " + i);
  var panel2 = buttongroup.add ("panel", undefined, "Knoppen");
    for (var i = 0; i < 5; i++)
      panel2.add ("button", undefined, "Button " + i);
```



```
    buttongroup.addEventListener('click', button_pressed);

    function button_pressed (e)
      {
      if (e.target.type == "button")
        $.writeln (e.target.text + " from panel " + e.target.parent.text);
      }

  w.show();
```

The type-check is necessary because you could have clicked the panel just outside a button. If you do that, **ev.target.type** would return **panel**, and the check allows you to ignore everything that's not a button.

For an excellent article on mouse events, see Marc Autret's Note on ScriptUI mouse events.

Listening to the keyboard

To listen to the keyboard, define an event listener using the keyboard event **keydown**. Here is an example that prints some properties of the keyboard event (this doesn't work properly when you target the ESTK):

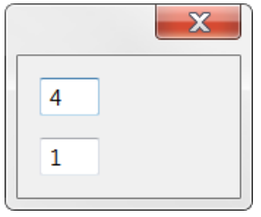
```
var w = new Window ("dialog");
  var edit = w.add ("edittext");
  edit.active = true;
  edit.characters = 30;
  w.addEventListener ("keydown", function (kd) {pressed (kd)});

  function pressed (k) {
    $.writeln (k.keyName);
    $.writeln (k.keyIdentifier);
    $.writeln (k.shiftKey ? "Shift pressed" : "Shift not pressed");
    $.writeln (k.altKey ? "Alt pressed" : "Alt not pressed");
    $.writeln (k.ctrlKey ? "Ctrl pressed" : "Ctrl not pressed");
  }
w.show ();
```

Keys have (among other things) a name (**A**, **B**, **Space**, **Minus**, **Shift**), an identifier (the hex value of the key in the form U+0000 or its name in the case of keys like Shift), and properties that return the state of the Shift, Ctrl, and Alt keys.

Using the up and down arrow keys to change numerical data

In another example we can do something about the paucity of input methods. The **edittext** control accepts just text, but with an event listener you can make this contol behave a bit more like numerical input fields in which you can increment or decrement values by pressing the up and down arrows. Holding the shift key down while pressing an arrow key increases the step value by 10. Here is the script:

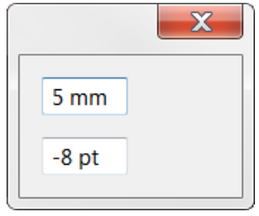
```
var w = new Window ("dialog");
  var e1 = w.add ("edittext", undefined, "1");
  var e2 = w.add ("edittext", undefined, "1");
  e1.characters = e2.characters = 3; e1.active = true;

  function handle_key (key, control)
    {
    var step;
    key.shiftKey ? step = 10 : step = 1;
    switch (key.keyName)
      {
      case "Up": control.text = String(Number(control.text)+step); break;
      case "Down": control.text = String(Number(control.text)-step);
      }
    } // handle_key

  e1.addEventListener ("keydown", function (k) {handle_key (k, this);});
  e2.addEventListener ("keydown", function (k) {handle_key (k, this);});

w.show();
```



The convoluted **String(Number(control.text)+step)** is necessary because we can do arithmatic only with numbers while the edittext control accepts just text, so we'll need to convert between numbers and text all the time. Anyway, the function **handle_key()** that's called by the event listener first checks if the Shift key is pressed; if it is, the step value is set to 10, else it's set to 1.

Measurement controls are just a slight complication of this. The following script adds measurement units to an input field. As you can see, it's now becoming a general JavaScript issue. The script handles both bare numbers and numbers followed by a measurement unit such as **mm** or **pt**.

```
var w = new Window ("dialog");
  var e1 = w.add ("edittext", undefined, "1 mm");
  var e2 = w.add ("edittext", undefined, "1 pt");

  e1.characters = e2.characters = 5; e1.active = true;

  function handle_key (key, control) {
    var step;
    key.shiftKey ? step = 10 : step = 1;
    switch (key.keyName) {
      case "Up": control.text = update (control.text, step); break;
      case "Down": control.text = update (control.text, -step);
      }
    }

  function update (str, incr) {
    try {
      var array = str.split(" ");
      var num = String (Number (array[0])+incr);
      if (array.length == 2) {num += " "+array[1];}
      return num;
    } catch (_) {
      alert ("Illegal input"); return str;
    }
  }

  e1.addEventListener ("keydown", function (k) {handle_key (k, this)});
  e2.addEventListener ("keydown", function (k) {handle_key (k, this)});

w.show();
```



The script simply hard-wires the units mm and pt, it doesn't look at the document. Nor can you enter something like 4cm in the first field, to be displayed as 40 mm. For a more Indesign-like approach, see the example on p. 68.

Selecting items in dropdowns using the keyboard

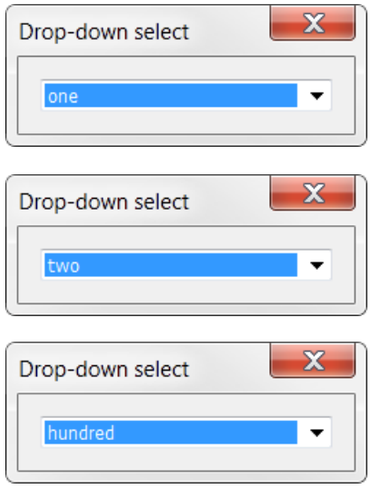
An example of a concrete application is the slight improvement of ScriptUI's **dropdown** control. In most dropdown lists, you can press a key to select an item in the dropdown that starts with that key's corresponding letter (type **t** and the first item starting with a t is displayed in the dropdown's control), but unfortunately that doesn't work in ScriptUI. The remedy is to attach an event

listener to the drop-down that monitors keystrokes and selects the first list item whose first letter matches the key press. Here is an example:

```
var numbers = ["one", "two", "three", "thirty", "hundred"];
var w = new Window ("dialog", "Drop-down select");
  var ddown = w.add ("dropdownlist", undefined, numbers);
  ddown.minimumSize.width = 200;
  ddown.selection = 0; ddown.active = true;

  ddown.addEventListener ("keydown", function (k)
    {
    k = k.keyName.toLowerCase();
    var i = 0;
    while (i < numbers.length-1 && numbers[i].charAt(0).toLowerCase() != k)
      {++i;}
    if (numbers[i].charAt(0).toLowerCase() == k)
      ddown.selection = i;
    }
  );
w.show ();
```

t >

h >



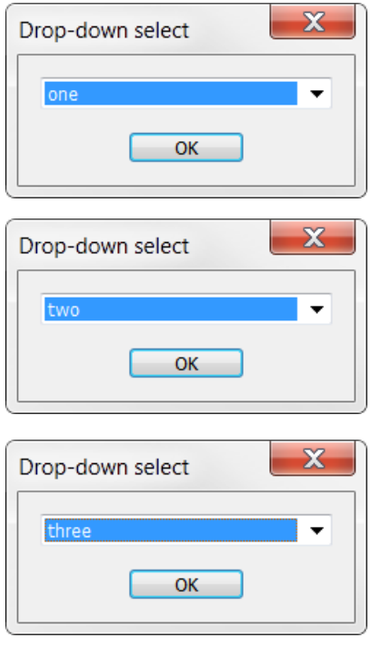
This mimics the behaviour of many dropdowns you encounter in InDesign but also, for instance, on the web. If you type **t**, two is displayed in the list; if you then type **h**, hundred is shown. In other words, any keypress matches just the first letter of an item. **Note**: on Macs you need to define the event handler on the window rather than on the dropdown.

But now that we're into event listeners we can do a bit better and mimic the behaviour of more clever dropdowns, such as that in the font dropdown in InDesign's Character panel. That list is like a type-ahead list, so that successive keypresses match the beginning characters of a list item. You can then type your way to an item, so to speak. Here is the script:

```
var numbers = ["one", "two", "three", "thirty", "hundred"];
var w = new Window ("dialog", "Drop-down select");
  var ddown = w.add ("dropdownlist", undefined, numbers);
  ddown.minimumSize.width = 200;
  ddown.selection = 0; ddown.active = true;

  var buffer = "";
  ddown.onActivate = function () {buffer = ""}

  ddown.addEventListener ("keydown", function (k)
```

t >

h >



```
    {
    buffer += k.keyName.toLowerCase();
    var i = 0;
    while (i < numbers.length-1 && numbers[i].toLowerCase().indexOf (buffer) != 0)
      {++i;}
    if (numbers[i].toLowerCase().indexOf (buffer) == 0)
      ddown.selection = i;
    }
  );

  w.add ("button", undefined, "OK");
w.show ();
```

The script now remembers what we type in the variable called **buffer**. When the script starts, this variable is initialised to an empty string. The event listener adds our input to the buffer every time we press a key. Of course, when you move away from the dropdown (by pressing Tab or clicking somewhere else in the window) and then go back to the dropdown, you want to start afresh. That's what the onActivate callback is for: when the dropdown is activated, the buffer is emptied.

The dropdown's behaviour is now different. If you press **t**, two is highlighted as before. But when you now type **h**, the script selects three in the list, not hundred as before.

There's one problem, though: you can't use the backspace key to correct a typing error. That's because when you press any key, its name is appended to the buffer (except the Tab key). To correct an error, you need to move away from the dropdown, go back to it, and start again. We would therefore like to add something to the event listener that removes the buffer's last character if the Backspace key is pressed, and adds a key's name only if the key's name is a single letter. This last point is relevant because if you press the Shift key, *shift* is added to the buffer. Here is the revised script:

On Macs there appear to be some problems with adding the event handler to the dropdown list. These problems can be avoided by adding the handler to the window, though that in itself is not unproblematic. See https://forums.adobe.com/thread/1705713 for the discussion.

```
var array = ["one", "two", "three", "thirty", "hundred"];
var w = new Window ("dialog", "Drop-down select");
  var ddown = w.add ("dropdownlist", undefined, array);
  ddown.minimumSize.width = 200;
  ddown.selection = 0; ddown.active = true;

  var buffer = "";
  ddown.onActivate = ddown.onDeactivate = function () {buffer = "";}

  ddown.addEventListener ("keydown", function (k)
```

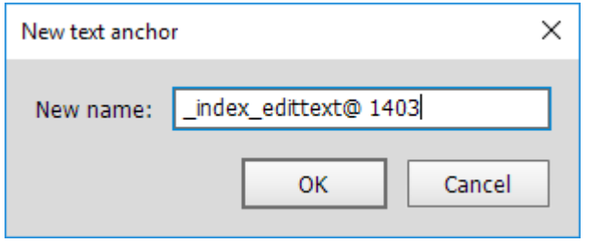
```
    {
    if (k.keyName == "Backspace")
      buffer = buffer.replace (/.$/, "");
    else
      if (k.keyName.length > 0)
        buffer += k.keyName.toLowerCase();
    var i = 0;
    while (i < array.length-1 && array[i].toLowerCase().indexOf (buffer) != 0) {++i;}
    if (array[i].toLowerCase().indexOf (buffer) == 0)
      ddown.selection = i;
    }
  );
  w.add ("button", undefined, "OK");
w.show ();
```

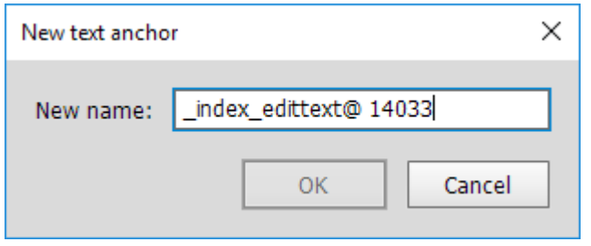
Note that we changed the callback, too, so that the buffer is reset on activation and on deactivation of the dropdown.

Validating input

There are several ways to validate the content of an edittext control. The simplest is simply to disable a window's OK button so that the user can't click OK while the edittext's content is not acceptable. For instance, if you ask the user to enter the name for a new text anchor, the simplest way to make sure that they provide a name that's not already in use is to check at every key press whether a text anchor with that name exists. In the following example, entering the last 3 creates a name that matches an existing name and the OK button is disabled. (This validation method and the one using a change of background colour is based on a script by Bob Stucky.)

```
var w = new Window ('dialog', 'New text anchor');
  w.group = w.add ('group');
    w.group.add ('statictext {text: "New name:"}');
    w.input = w.group.add ('edittext {characters: 20, active: true}');

    w.buttons = w.add ('group {alignment: "right"}');
      w.ok = w.buttons.add ('button {text: "OK", enabled: false}');
      w.buttons.add ('button {text: "Cancel"}');

  w.input.onChanging = function () {
    w.ok.enabled = !app.activeDocument.hyperlinkTextDestinations.item
      (w.input.text).isValid;
  }
```

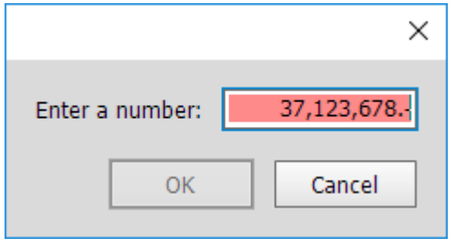




```
if (w.show() == 1 && w.input.text.length > 0) {
  app.activeDocument.hyperlinkTextDestinations.add (app.selection[0],
    {name: w.input.text});
}
```

If a window contains many fields it's often helpful to emphasise an edittext control by changing its background color so that it's easier to spot. This, too, can be done while the user enters data (this no longer works after CS6).

```
var w = new Window ('dialog');
  w.group = w.add ('group');
    w.group.add ('statictext {text: "Enter a number:"}');
    w.input = w.group.add ('edittext {characters: 10, active: true, justify: "right"}');

    w.buttons = w.add ('group {alignment: "right"}');
      w.ok = w.buttons.add ('button {text: "OK", enabled: false}');
      w.buttons.add ('button {text: "Cancel"}');

  w.input.onChanging = function () {
    var valid = /^[\d.,]+$/.test (w.input.text);
    this.graphics.backgroundColor = this.graphics.newBrush (this.graphics.BrushType.
      SOLID_COLOR, valid ? [1, 1, 1, 1] : [1, 0.5, 0.5, 1]);
    w.ok.enabled = valid;
  }

w.show();
```



As soon as the user enters a character that's not a digit, dot, or comma, the background is changed to red. When the illegal character is removed, the window goes back to normal.

Three-state checkboxes: Sprites

ScriptUI's checkbox controls are two-state: on or off. But in many cases – in InDesign anyway – a text selection, for example, can be in three states: on, off, or mixed. ScriptUI doesn't provide anything natively for three-state boxes, but with some (considerable) effort those three-state checkboxes can be added to a ScriptUI panel.

The idea behind this is to create a so-called sprite, as was shown by Marc Autret (see his blog on the subject). The technique is a viewport approach (see also "Scrolling panels and groups" on page 66): create a graphic that shows the three states, add an image control in a ScriptUI window that's (in our case)

The red frame represents the ScriptUI image object. The graphic is moved horizontally. Because the image object is a third of the graphic's width, only one state is ever shown.

a third of the width of the image (that's in fact the viewport), and add an event listener that slides the image horizontally.

The script, below, relies heavily on Marc's method. For the sprite I use three graphics: one for CS6, one for the CC light interface, and the third for the CC dark interface. They must be separate because the CS6 boxes are bigger. The CC graphics could be combined but the way it is now, it's easy to add graphics for other shades of interface. (For myself I use strings instead of graphic files, but the strings are much too long to use here in this guide.)

The script's window is very small. It can be used in the same way that InDesign's three-state checkboxes are used: it shows the state of the current selection and can be used to change the current selection's nobreak state.

```
#targetengine nobreak;

(function () {

  var windowName;
  var nobreakWindow;

  //--------------------------------------------------------------
  // Slightly adapted from Marc Autret's
  // http://www.indiscripts.com/post/2011/04/sprite-buttons-in-scriptui
  // All comments in this section are Marc's.
  // ScriptUI Image Offset Fixer in InDesign CS6 and earlier
  // (this bug was solved in CC i.e. ScriptUI 6.2.x)

  var CC_FLAG = parseInt(app.version) >= 9 ? 1 : 0;
  var FIX_OFFSET = CC_FLAG ? 0 : 1;

  // Force an Image widget to repaint (= onDraw trigger)
  // CC: we need to temporarily change the size
  // (layout.layout(1) does not work anymore in CC)
  // CS4-CS6: just reassign this.size

  Image.prototype.refresh = function() {
    if (CC_FLAG) {
      var wh = this.size;
      this.size = [1+wh[0], 1+wh[1]];
      this.size = [wh[0], wh[1]];
      wh = null;
```

```
    } else {
      this.size = [this.size[0], this.size[1]];
    }
  }

  // Three different sprites: CS6, light CC, and dark CC ----------------------------------------
  function threeWayCheckboxImage() {
    if (parseInt (app.version) < 9) {
      return File ('/D/icons/off_on_mixed_CS6.png');
    } else if (app.generalPreferences.uiBrightnessPreference == 1) {
      return File ('/D/icons/off_on_mixed_CC_light.png');
    } else {
      return File ('/D/icons/off_on_mixed_CC_dark.png');
    }
  }

  // Check if there's a selection and if there is, if it's valid ---------------------------------

  function noBreakApplies () {
    return app.windows.length > 0 && app.documents.length > 0
      && app.selection.length > 0 && app.selection[0].hasOwnProperty('noBreak');
  }

  // Return the nobreak state of the current selection -------------------------------------------

  function getNoBreakState () {
    var ret;
    if (!noBreakApplies()) {
      ret = 0;
    } else if (app.selection[0] instanceof InsertionPoint) {
      ret = app.selection[0].underline ? 1 : 0;
    } else {
      var txt = app.selection[0].textStyleRanges.everyItem().underline.join('');
      if (txt.indexOf ('true') > -1 && txt.indexOf ('false') > -1) {
        ret = 2;
      } else if (txt.indexOf ('true') > -1) {
        ret = 1;
      } else {
        ret= 0;
      }
    }
    return ret;
  }
```

```
function updateNobreak () {
  var w = Window.find ('palette', windowName);
  w.cbox.state = getNoBreakState();
  w.cbox.refresh();
}

// Create a window ---------------------------------------------------------------------
function defineNobreakWindow (name) {
  var w = new Window ('palette');
    w.text = name;
    var checkBoxGroup = w.add ('group');
      var boxWindow = checkBoxGroup.add ('group');
        w.cbox = boxWindow.add ('image', undefined, threeWayCheckboxImage());
        w.cbox.state = 0;
        checkBoxGroup.add ('statictext {text: "No break"}');

    // Three-way checkbox handlers ---------------------------------------------------------
    var iSize = w.cbox.image.size;
    var spriteWidth = iSize[0] / 3;

    w.cbox.size = [spriteWidth, iSize[1]];

    w.cbox.onDraw = function(){
      var dx = this.state * spriteWidth + FIX_OFFSET;
      this.graphics.drawImage (this.image, -dx, 0);
    };

    var mouseEventHandler = function(){
      if (noBreakApplies()) {
        switch (this.state) {
          case 0: case 2: app.selection[0].noBreak = true; this.state = 1; this.refresh();
break;
          case 1 : app.selection[0].noBreak = false; this.state = 2; this.refresh(); break;
        }
      }
    };

    w.cbox.addEventListener ('click', mouseEventHandler);

  return w;
}
```

```
    //--------------------------------------------------------------------------
    windowName = 'No break';
    nobreakWindow = Window.find ('palette', windowName);
    if (nobreakWindow === null){
        nobreakWindow = defineNobreakWindow (windowName);
        app.addEventListener ('afterSelectionChanged', updateNobreak);
    }
    updateNobreak();
    nobreakWindow.show();
}());
```

Size and location

There are two ways to determine the size and the location of a window and all its children: automatically and manually. The automatic method uses ScriptUI's layout manager and, since it does a pretty good job most of the time, is by far the easiest way to create windows. The manual method involves specifing the size and position of all controls, which can turn into a nightmare if you decide to change the type and typesize of one or more controls, or when you change the window's layout. In this section, therefore, we deal almost exclusively with the way the layout manager goes about its business. Here and there we'll use bits of the manual method when it's convenient, but that doesn't happen very much.

Much of what we describe here is well documented in the Guide, but several aspects remain unclear there.

Size

By default, windows are placed in the centre of the screen and its controls are centred in their containers, both horizontally and vertically. Size and position can be stated in several different formats, which are mere notational variants:

```
    var w = new Window ("dialog");
        w.preferredSize = [300, 400];
        w.preferredSize = {width: 300, height: 400}
        w.preferredSize = "width: 300, height: 400";
        w.preferredSize.width = 300;
        w.preferredSize.height = 400;
        w.preferredSize[0] = 300;
        w.preferredSize[1] = 400;
        w.preferredSize = [300, ""];          // set just the width
        w.preferredSize = ["", 400];          // set just the height
w.show()
```

As with all JavaScript arrays and objects, the array notation is ordered ([width, height]), the object notation is not. The format of last two are not very useful here, but will come in handy later on when we deal with alignments.

preferredSize tells the layout manager to aim at a certain size. To fix the size, use **size**. But the size can (often) be set only when the window is being drawn, so that **size** must be used inside an **onShow** callback:

```
var w = new Window ("dialog");
w.onShow = function () {w.size = {width: 300, height: 400}}
w.show();
```

All the notations available for **preferredSize** can be used for **size**, too:

```
  w.size = [300, 400];
  w.size = {width: 300, height: 400};
  w.size = "width: 300, height: 400";
  w.size.width = 300;
  w.size.height = 400;
  w.size[0] = 300;
  w.size[1] = 400;
```

Location

A window's location is set with the **location** property. This one, too, can be used with the several notational variants:

```
var w = new Window ("dialog");
  // either of the following three:
  w.location = [100, 200];
  w.location = {x:100, y:200};
  w.location = "x:100, y:200";
w.show();
```

The **w.location.x** = and **w.location**[**0**] = formats are possible only in **onShow** callbacks:

```
var w = new Window ("dialog");

  w.onShow = function () {
    w.location.x = 100;
    w.location.y = 200;
  }

w.show();
```

Bounds

Size and location together make up a contol's **bounds**, which are presented as a four-element array:

```
var w = new Window ("dialog");
  // either of the following three
  w.bounds = [100, 200, 300, 400];
  w.bounds = {left: 100, top: 200, right: 300, bottom: 400}
  w.bounds = "left: 100, top: 200, right: 300, bottom: 400";
w.show();
```

As with all controls, using the array notation, bounds can be included in the control's definition:

```
var w = new Window ("dialog", "Title", [100, 200, 300, 400]);
w.show();
```

A window's bounds exclude the frame, so when you place a window at the top left of a screen you should allow for the frame. The window's bounds including the frame are returned by **frameBounds**. This script prints a window's bounds and frame bounds in the console:

```
var w = new Window ('dialog', '', [100, 200, 300, 400]);
  $.writeln ('Bounds: ' + w.bounds)
  $.writeln ('Frame bounds: ' + w.frameBounds)
w.show();
```

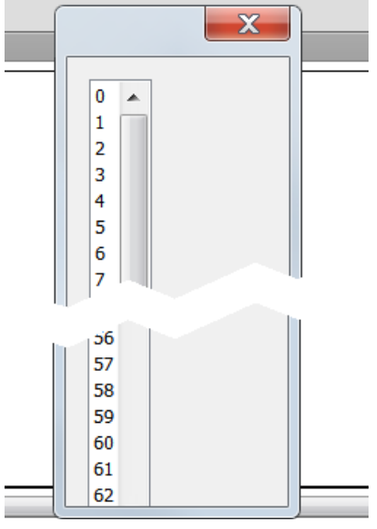
```
Bounds: 100,200,300,400
Frame bounds: 97,170,303,403
```

The output shows that the left, right, and bottom frame are 3 units wide, and that the top of the frame is 30 units.

Maximum size

Maximum size is needed to prevent controls from disappearing off the screen. This could happen, for example, with long lists. The following script displays a list of 100 items in a list box, but the screen is clearly not tall enough to accommodate that list: it manages 61 items on my screen.

```
var w = new Window ('dialog');
  var list = w.add ('listbox');
  for (var i = 0; i < 100; i++)
    list.add ('item {text: ' + i + '}');
w.show();
```

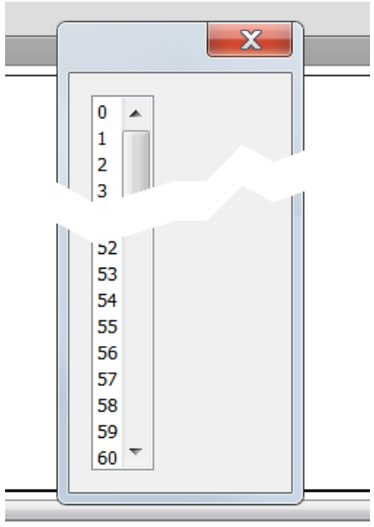


What we want is to set a maximum height to the list. But, naturally, we first need to know the maximum size of the dialog. To find that height, run the following script:

```
var w = new Window ('dialog');
  $.writeln (w.maximumSize.height);
```

On my PC this prints **1150** to the console. You can use this value to set the list's maximum height, but you'll see that the list still spills over the window because of the top and bottom frameBounds. After some experimentation, 1100 turns out to work well for me. To keep the script general, rather than hard wiring a value, we now say that the lists's maximum height should the the window's maximum height minus 50:

```
var w = new Window ('dialog');
  var list = w.add ('listbox');
  for (var i = 0; i < 100; i++)
    list.add ('item {text: ' + i + '}');
  list.maximumSize.height = w.maximumSize.height - 50;
w.show();
```



Predictably, apart from **height**, **maximumSize** has a property **width** as well. In my experience, **maximumSize.width** isn't needed as much as **maximumSize.height** – I use it only for resizeable windows.

But if you use a dual-monitor system, you may be in for a surprise: **maximumSize.width** considers both screens as one. For example, I have two 24-inch 1920 × 1200 screens next to each other. The window property **maximumSize.height** correctly returns 1150 (1200 minus 50 for system overhead), but **maximumSize.width** tells me that my window is 3790 pixels wide. In itself this is correct because you could drag a window across two screens, but you need to be aware of this if, for instance, you want to position a window at the right-hand side of the screen on the left.

To get the correct value for the maximum width of a screen, you need to use **$.screens**, which returns an array of objects representing screen coordinates. Note that this is part of the ESTK, not ScriptUI. The following script lists the coordinates of all screens, in this case, two:

```
for (var i = 0; i < $.screens.length; i++) {
  $.writeln ("Screen " + i);
  $.writeln ("top: " + $.screens[i].top);
  $.writeln ("left: " + $.screens[i].left);
  $.writeln ("bottom: " + $.screens[i].bottom);
  $.writeln ("right: " + $.screens[i].right);
}
```

```
Screen 0
top: 0
left: 0
bottom: 1200
right: 1920

Screen 1
top: 0
left: 1920
bottom: 1200
right: 3840
```

So the maximum width of windows placed on the first screen is about 1900.

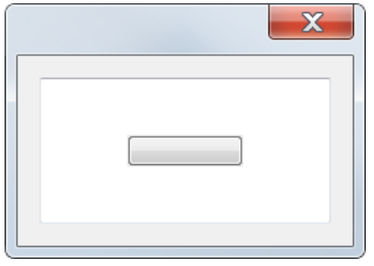
Minimum size

Use **minimumSize** to set a control's minimum width and height. Its values are set in the same way as **preferredSize** and **maximumSize**. I use minimumSize especially in resizable windows; for an example, see "Resizing windows" on page 103.

Orientation

Containers can have either of three orientations: **row**, **column**, or **stack**. Windows and panels default to column, groups to row orientation. These are entirely straightforward. Stack orientation, on the other hand (in InDesign), has suffered from an inconsistency between Macs and Windows PCs up to and including CS6. In that environment, when you stack two controls, the second object drawn is placed behind the first control, but on Macs the second control is placed in front of the first control. When you use **stack** you should therefore do a platform check and on Windows, also do a version check. The following script illustrates:

```
var w = new Window ('dialog');
  w.orientation = 'stack';
  if ($.os.indexOf ('Windows') >-1 && parseInt (app.version) < 9)  {
    // On Windows, before CC
    w.add ('button');
    w.add ('edittext {preferredSize: [200, 100]}')
  }  else {// On Mac or on Windows from CC
    w.add ('edittext {preferredSize: [200, 100]}')
    w.add ('button');
  }
w.show();
```



Margins and spacing

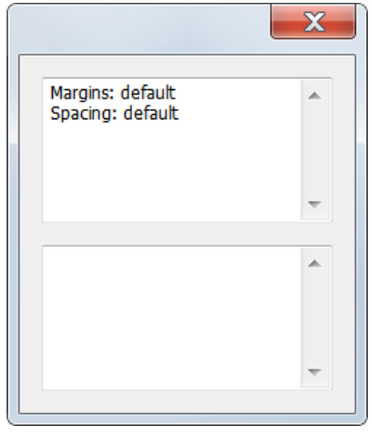
Margins define the space between a container's edges and its children. Margins can be set in the familiar way:

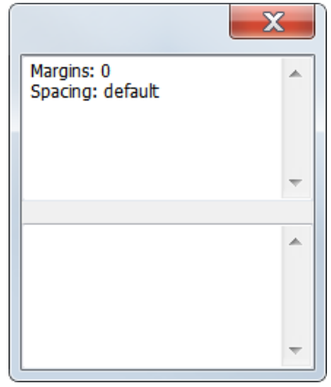
```
w.margins = [0, 0, 0, 0];
w.margins = 0;          // same effect as previous line
w.margins.left = 0;
```

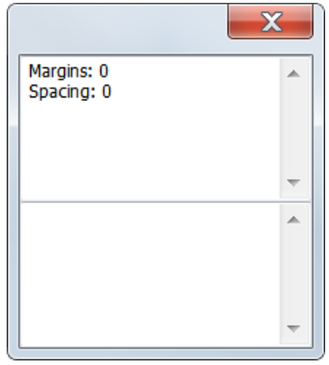
The order of the four values in the **margins** array is [left, top, right, bottom].

Spacing defines the space between a container's children. The scripts below illustrate:

```
var w = new Window ('dialog');
  p1 = w.add ('edittext {preferredSize: [200, 100], properties: {multiline: true}}');
  p2 = w.add ('edittext {preferredSize: [200, 100], properties: {multiline: true}}');
  p1.text = 'Margins: default\rSpacing: default'
w.show ();

var w = new Window ('dialog');
  w.margins = 0;
  p1 = w.add ('edittext {preferredSize: [200, 100], properties: {multiline: true}}');
  p2 = w.add ('edittext {preferredSize: [200, 100], properties: {multiline: true}}');
  p1.text = 'Margins: 0\rSpacing: default'
w.show ();

var w = new Window ('dialog');
  w.margins = 0;
  w.spacing = 0
  p1 = w.add ('edittext {preferredSize: [200, 100], properties: {multiline: true}}');
  p2 = w.add ('edittext {preferredSize: [200, 100], properties: {multiline: true}}');
  p1.text = 'Margins: 0\rSpacing: 0'
w.show ();
```



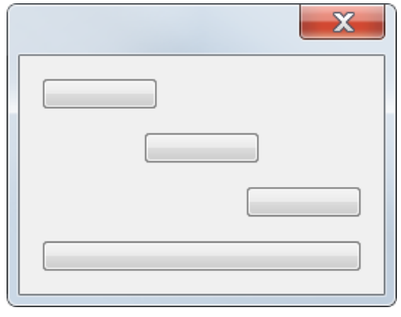




alignment

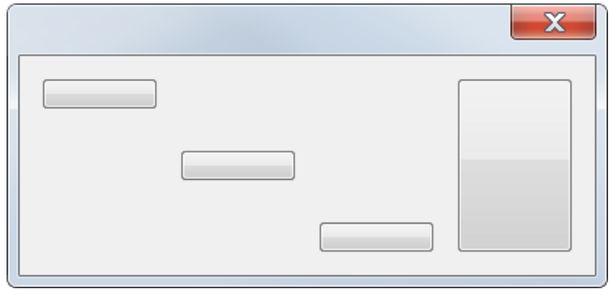
Alignment is set the horizontal or vertical alignment of a single control. Horizontal alignments are **left**, **right**, **center**, and **fill**.

```
var w = new Window ('dialog {preferredSize: [250, ""]}');
  var b = w.add ('button'); b.alignment = 'left';
  var b = w.add ('button'); b.alignment = 'center';
  var b = w.add ('button'); b.alignment = 'right';
  var b = w.add ('button'); b.alignment = 'fill';
w.show();
```



Vertical alignments are **top**, **center**, **bottom**, and **fill**.

```
var w = new Window ('dialog {preferredSize: ["", 150], orientation: "row"}');
  var b = w.add ('button'); b.alignment = ["", 'top'];
  var b = w.add ('button'); b.alignment = ["", 'center'];
  var b = w.add ('button'); b.alignment = ["", 'bottom'];
  var b = w.add ('button'); b.alignment = ["", 'fill'];
w.show();
```

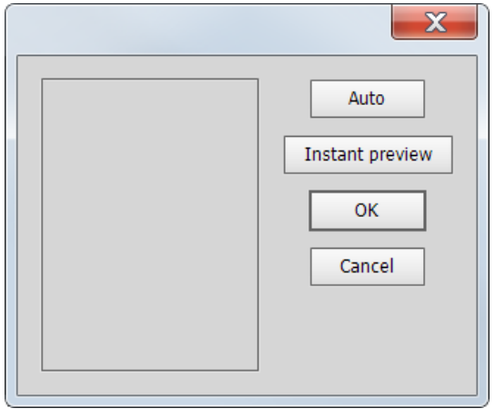


Notice that vertical alignment is set by using the second element of the alignment array. If you want to set just the vertical alignment, leave the horizontal alignment – the first element – undefined, as in this example.

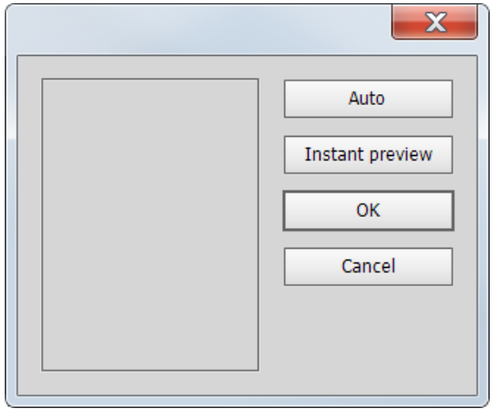
alignChildren

alignment sets the alignment of a single control. To set the alignment of all children in a container (a window, group, or panel), use **alignChildren**. This is useful, for instance, to make all buttons in a group the same width, as in the following script.

```
var w = new Window ('dialog {orientation: "row", alignChildren: ["", "top"]}');
  w.add ('panel {preferredSize: [150, 200]}');
  var g = w.add ('group {orientation: "column"}');
    g.alignChildren = "fill";
    g.add ('button {text: "Auto"}');
    g.add ('button {text: "Instant preview"}');
    g.add ('button {text: "OK"}');
    g.add ('button {text: "Cancel"}');
w.show();
```



Without g.alignChildren = 'fill'



With g.alignChildren = 'fill'

Instead of setting each button's horizontal alignment to **fill**, here we set the group's **alignChildren** value, which transfers the alignment to all the group's children.

As with **alignment**, to set a horizontal value for **alignChildren** we can simply give that value as a string ('left', 'center', 'right', 'fill'), as in the above example; and again, to set just a vertical value for alignChildren we need to use the second element in an array, as in

```
g.alignChildren = [ "", "fill"];
```

Resizing windows

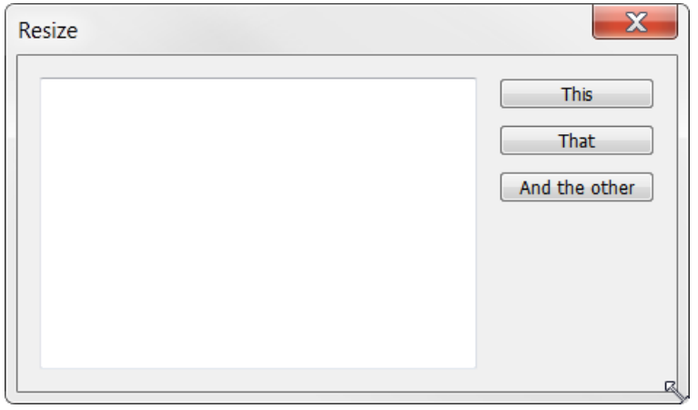
Moving a window is easy: just grab the window's title bar and drag it to somewhere else on the screen. This is useful when the window covers what you were looking at.

It can also be useful to make windows resizable, but that's a bit more complicated. The script below shows a simple window which can be resized. It draws a window with an edit field and a group with three buttons in it. The user can resize the window in the usual way by clicking on a corner or a side of the window's frame and dragging it left, right, up, or down.

The window has two main alements, the edittext control and the group with three buttons. The size of the button group not specified, and by setting the alignment of the edittext control to **["fill", "fill"]** we say in effect that it should always take up all the available space of the window – that is, the space in the window minus the space taken by the button group.

The first thing to do to make a window resizeable is to include the creation property **{resizeable: true}**. In addition, the script needs the callback **onResizing**, which monitors whether the window is being resized; if it is, the window is redrawn. (Marc Autret reports that on a Mac you need onResize as well.)

```
var w = new Window ("dialog", "Resize", undefined, {resizeable: true});
   w.orientation = "row";

  var e = w.add ('edittext')
     e.alignment = ["fill", "fill"];
     e.minimumSize = [300, 200];

   var g = w.add ('group {orientation: "column"}');
     g.alignment = ['right', 'top'];
     g.alignChildren = 'fill';

     g.add ('button {text: "This"}');
     g.add ('button {text: "That"}');
     g.add ('button {text: "And the other"}');

  // onResize needed on Mac OS X

  w.onResizing = w.onResize = function () {this.layout.resize ();}

  w.onShow = function () {
     w.minimumSize = w.size;
  }

w.show ();
```



Another important step is the calculation of the minimum size of the window. We do this in the **onShow** callback. **onShow** determines the layout by calculating it, and at that point we can say that the window's minumum size is the size it has when it was first drawn.

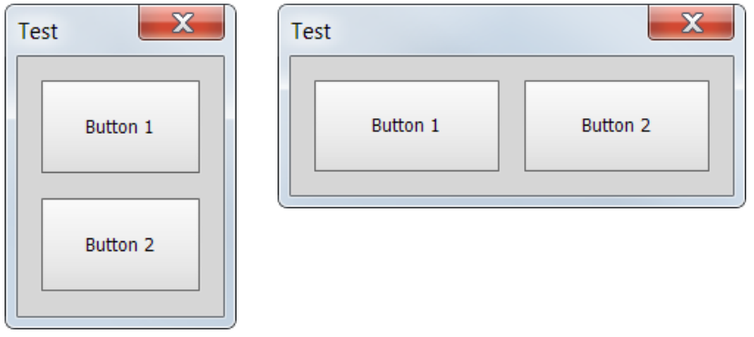
Note: resizing appears not to work too well using the above script in CS6 on Windows 8.1. A remedy was suggested by Lukas Sommer, which works fine on Windows 7 too. The change is a different formulation of the onShow callback:

```
w.onShow = function () {
  w.minimumSize = w.size; // define the initial standard size as minimum size
  for (var i = 0; i < w.children.length; i++) {
    w.children[i].minimumSize = w.children[i].size;
  }
}
```

It amounts to setting the minimum size not only of the container but also of all its children.

The orientation of a window (or group, panel) can be changed dynamically when a window is resized. The following script arranges the two defined buttons horizontally or vertically depending on the shape of the window:

```
w = new Window ('dialog', 'Test', undefined, {resizeable: true})
  w.alignChildren = ['fill', 'fill'];
  w.add ('button {text: "Button 1"}');
  w.add ('button {text: "Button 2"}');

  w.onResizing = w.onResize = function () {
    w.orientation = w.size.width > w.size.height ? 'row' : 'column';
    w.layout.resize ();
  }

  w.onShow = function () {
    w.layout.resize ();
  }

w.show();
```



All that's needed is to check the shape of the container and change its orientation accordingly.

For a flowchart and a draft outline of ScriptUI's layout manager, see Marc Autret's outline.

Coding styles: resource string and code based

The coding style in the examples given so far is sometimes called code based. This is the style I use most of the time because I think it's the more convenient one. There are however two other, distinct, styles, and the choice between them appears to be informed mainly by taste.

Resource string

The first alternative is the so-called resource string, in which the whole window is presented as a single string that defines the window's characteristics as an object. The script below is the script in p. 8 recast using a resource string.

Proponents of this style make the point that using resource strings encourages the coder to separate the window's design from its functional component. This is clear in the example: the resource string states only how the window looks, not what happens when you press a button or type something in a field. Whether this is a valid point is a matter of taste. A clear advantage, however, is that resource strings make for very compact code (though in this example that's not entirely clear).

A disadvantage of resource strings is that they can be very hard to debug if you make an error, mainly because the ESTK's error messages aren't particularly helpful. In addition, I find it easier to change a window's layout using the code-based style.

As mentioned earlier, which style you choose is determined mainly by taste. Here is our previous example using the resource-based style:

```
function myInput () {
  var winResource =
    "dialog {text: 'Form', \
      myInput: Group {\
        n: StaticText {text: 'Name:'},\
        myText: EditText {text: 'John', characters: 20, active: true}\
      },\
      myButtonGroup: Group {alignment: 'right',\
        b1: Button {text: 'OK'},\
        b2: Button {text: 'Cancel'}\
      }\
    }";
  var myWindow = new Window (winResource);
  if (myWindow.show () == 1) {
    return myWindow.myInput.myText.text;
  }
}
```

Since the resource string defines a complex object, you refer to any property as you would any JavaScript property, as in this line:

```
return myWindow.myInput.myText.text;
```

Note that capitalisation is slightly different: use **Button** and **StaticText**, not **button** and **statictext** – the names of controls and properties are case-sensitive in different ways depending on the coding style.

Code-based object

The second alternative, which I dub 'object code' for lack of a better term, is in a way a mix of the two other styles: it doesn't use a resource code, but it does define the window as a complex object. This is illustrated in the following example, which is again the script on p. 8, with just the variable names abbreviated to keep things manageable. And this is the main reason why I don't use this style: it can get terribly verbose. An advantage of this style may be that you use just one variable, namely, the one to create the window.

```
function myInput () {
  var win = new Window ("dialog", "Form");
    win.inpGrp = win.add ("group");
      win.inpGrp.add ("statictext", undefined, "Name:");
      win.inpGrp.txt = win.inpGrp.add ("edittext", undefined, "John");
      win.inpGrp.txt.characters = 20;
      win.inpGrp.active = true;
    win.btnGrp = win.add ("group");
      win.btnGrp.alignment = "right";
      win.btnGrp.add ("button", undefined, "OK");
      win.btnGrp.add ("button", undefined, "Cancel");
  if (win.show () == 1) {
    return win.inpGrp.txt.text;
  }
  exit ();
}
```

If you want to refer to very deeply embedded controls, you can set their name property and use **findElement()** to get a quick reference. For instance, if you change the edittext control in the above script as follows:

```
win.inpGrp.txt = win.inpGrp.add ("edittext", undefined, "John", {name: "personname"});
```

you can then get a quick reference to that control with this line:

```
var bloke = win.findElement ("personname");
```

An alternative to this style is to use with statements, as in the forrowing example:

```
function myInput () {
  var win = new Window ('dialog', 'Form');
```

```
    with (win) {
      with (add ('group')) {
        add ('statictext {text: "Name:"}');
        add ('edittext {text: "John", characters: 20, active: true, properties: {name:
"person"}}');
      }
      with (add ("group")) {
        add ('button {text: "OK"}');
        add ('button {text: "Cancel"}');
      }
    }
  if (win.show () == 1) {
    return win.findElement('person').text;
  }
  exit ();
}
```

Mixing styles

The two styles can be mixed. The following example is still the same script, but this incarnation is essentially code-based with a few resource strings thrown in:

```
function myInput () {
  var win = new Window ("dialog", "Form");
    var inpGrp = win.add ("group");
      inpGrp.add ("statictext {text: 'Name:'}");
      var myText = inpGrp.add ("edittext {text: 'John', characters: 20, active: true}");
    win.btnGrp = win.add ("group {alignment: 'right'}");
      win.btnGrp.add ("button {text: 'OK'}");
      win.btnGrp.add ("button {text: 'Cancel'}");
  if (win.show () == 1) {
    return myText.text;
  }
  exit ();
}
```

I find myself using this hybrid form more and more because it is compact and the code is still easy to debug. In some cases the mixed method is easier to read. For example, in a code-based approach the properties of a slider are set as follows:

```
var slider = w.add ("slider", undefined, 50, 0, 100);
```

You have to remember the order in which the start, minimum, and maximum values are given (50, 0, and 100, respectively, in the example). An much clearer alternative to this is the following, still code-based, version:

```
var slider = w.add ("slider");
  slider.minvalue = 0;
  slider.maxvalue = 100;
  slider.value = 50;
```

The mixed style, on the other hand, is more compact and just as easy to understand:

```
var slider = w.add ("slider {value: 50, minvalue: 0, maxvalue: 100}");
```

As with all JavaScript objects, the order in which the properties are given is immaterial.

What has eluded me a long time is how to set a single preferred size in a resource string. You can't do any of the following:

```
e = w.add ('edittext {preferredSize.width: 150}')   // All these are wrong
e = w.add ('edittext {preferredSize[0]: 150}')
e = w.add ('edittext {preferredSize {width: 150}}')
```

By chance I found that -1 can be used as the default value for a control. In the following example the control's width is set to 150, the height is set to the default:

```
e = w.add ('edittext {preferredSize: [150, -1]}')
```

Other controls allow 'undefined' (see "Setting the size of controls" on page 110); **edittext** allows just -1.

Creation properties

In code-based styles, creation properties are included in the line that creates an object, other properties are stated after the control's creation (see "Creation properties and other properties" on page 10 for further details). Example:

```
var w = new Window ("dialog");
  var b = w.add ("button", undefined, "Continue", {name: "ok"});
  b.alignment = "fill";
w.show();
```

In mixed style, the button could be defined as follows:

```
var w = new Window ("dialog");
  var b = w.add ('button {text: "Continue", alignment: "fill", properties: {name: "ok"}}');
w.show();
```

Setting the size of controls

Sometimes you want to set just one dimension of a control. To set a control's width, for example, and let its height be determined by ScriptUI's layout manager, you would use **myControl.preferredSize.width = 200**, as in the following script:

```
var w = new Window ("dialog");
  var panel = w.add ("panel");
    panel.preferredSize.width = 200;
  panel.add ('button');
  panel.add ('button');
w.show();
```

In a mixed style, however, you can't use **.width**, and you have to use the array notation and leave one element unspecified. In our example we want to set the width, which is the first element in the two-element array, so we leave the second element, which defined the height, undefined (we could have used -1 here, too, or leave it out altogether):

```
var w = new Window ("dialog");
  // All three variants work:
  var panel = w.add ("panel {preferredSize: [200, undefined]}");
  var panel = w.add ("panel {preferredSize: [200, -1]}");
  var panel = w.add ("panel {preferredSize: [200, ]}");
  panel.add ('button');
  panel.add ('button');
w.show();
```

Displaying properties and methods

The properties and methods associated with all controls are listed in the Tools Guide but the guide has not always been accurate, presumably because of changes to ScriptUI after the Guides had been finished. In addition, ScriptUI has its own entry in the ESTK's object-model viewer – press F1 in the ESTK to access it. This is more reliable than the PDF.

For a convenient way to get a quick overview of which properties and methods are available for a particular control, you can use the two functions in the following script. The first few lines of the script define a simple dialog with

just one edittext control. A call to **prop()** prints all properties and their current values. A call to **meth() prints** a similar list for all methods (bold added).

```
var w = new Window ("dialog");
  var e = w.add("edittext", undefined, "Cats");
  prop(e);
  meth(e);
//~ w.show ();

function prop (f)
  {
  $.writeln (f.reflect.name);
  var props = f.reflect.properties;
  var array = [];
  for (var i = 0; i < props.length; i++)
    try {array.push (props[i].name + ": " + f[props[i].name])} catch (_){}
  array.sort ();
  $.writeln (array.join ("\r"));
  }

function meth (m)
  {
  var props = m.reflect.methods.sort();
  $.writeln ("\rMethods");
  for (var i = 0; i < props.length; i++)
    $.writeln (props[i].name);
  }
```

EditText
active: false
alignment: undefined
bounds: undefined
characters: undefined
children:
enabled: true
graphics: [object ScriptUIGraphics]
helpTip:
indent: undefined
justify: left
location: undefined
maximumSize: 3790,1150
minimumSize: 0,0
parent: [object Window]
preferredSize: 35,20
properties: undefined
shortcutKey: undefined
size: undefined
text: Cats
textselection:
type: edittext
visible: true
window: [object Window]
windowBounds: 0,0,100,30

Methods
addEventListener
dispatchEvent
hide
notify
removeEventListener
show

It's a simple list without any clarification. Nevertheless, I find it useful to print a quick overview of a control's properties (you can use the script to display properties and methods in the object models of all CS and CC applications, not just ScriptUI).

Some of the properties are objects themselves, and these can be displayed, too. For example, change **prop(e)** in the third line of the script as follows to print a list of just the ScriptUIGraphics object:

```
prop (e.graphics);
```

ScriptUIGraphics
BrushType: [object Object]
PenType: [object Object]
backgroundColor: undefined
currentPath: [object ScriptUIPath]
currentPoint: undefined
disabledBackgroundColor: undefined
disabledForegroundColor: undefined
font: Tahoma:12.0
foregroundColor: undefined

Resources

Virtually the only real comprehensive ScriptUI resource is the dedicated chapter in the Tools Guide that comes with the ESTK; you can find it in the ESTK's Help menu. This is a PDF file called **JavaScript Tools Guide CSx/CC.pdf**.

Using the ESTK's object-model viewer you can browse the ScriptUI object model. And Jongware's fabulous CS object browsers include a version for ScriptUI (see http://www.jongware.com/idjshelp.html). A more recent alternative (based on JongWare's browser) is Georg Fellenz's browser (https://www.indesignjs.de/extendscriptAPI/indesign14/#about.html). These are versioned: *indesign14* is CC2019; to access a different version, change 14 to the appropriate number.

Another source of information is the collection of sample scripts that comes with the ESTK, and which are listed on the first page of the ScriptUI chapter in the Tools Guide. To find these scripts, search your hard disk for a file with the name one of them (e.g. ColorPicker.jsx): their location depends on your operating system and CS version.

A more interesting collection of scripts are those that can be found in the ESTK's **Required** subdirectory. The scripts there form the basis of the ESTK's interface and use ScriptUI for their dialogs and other windows. This is a fascinating collection, with many instructive techniques. Do **not** on any account change these scripts. The location of the scripts depends on your OS and CS/CC version. To find the folder, search your disk for a script called 35OMVui.jsx.

Bob Stucky has collected a number of educational scripts here: http://www.creativescripting.net/freeStuff/ScriptUI_Validation.zip.

For help, go to Adobe's scripting forum (http://forums.adobe.com/community/indesign/indesign_scripting), where there's always someone at hand to help with problems. There is also a dedicated ScriptUI forum at http://forums.scriptui.com/.

Apart from that (and the present text), information can be found in some blogs and forums, some of which I list below (If you know of any useful blogs and links, please let me know).

Blogs

The following blogs feature several items and script examples dealing with ScriptUI:

– Marc Autret's **Indiscript**: http://www.indiscripts.com/
– Loïc Aigon's **Scriptopedia** at http://www.scriptopedia.org/ and his more personal http://www.loicaigon.com/blog/ (both are in French and English and cover InDesign, Illustrator, Photoshop, and Acrobat).
– Davide Barranca writes about Photoshop and Bridge. See especially http://www.davidebarranca.com/2012/10/scriptui-window-in-photoshop-palette-vs-dialog/ and http://www.davidebarranca.com/2012/11/scriptui-bridgetalk-persistent-window-examples/ for persistent windows in Photoshop and

Bridge, respectively. http://www.davidebarranca.com/2015/06/html-panel-tips-17-cc2015-survival-guide/ is basically about HTML panels, but has some comments about ScriptUI as well (see especially point 8).

– Joonas Pääkkö (who created a dialog builder, see "Resources"), has what is in effect a blog on GitHub (https://github.com/joonaspaakko/ScriptUI-Dialog-Builder-Joonas). He'll take feature requests there as well.

– Gerald Singelmann's site at http://indesign-faq.de/, a blog with several interesting scripts with ScriptUI solutions.

– Marijan Tompa's **InDesign Snippets**: http://indisnip.wordpress.com/ (last updated years ago, but still useful).

Useful forum topics

Some useful topics are floating around here and there, many of them in Adobe's scripting forum (http://forums.adobe.com/community/indesign/indesign_scripting), but Marc Autret's blog (www.indiscripts.com) has a growing number of instructive articles on ScriptUI. In the list below I've grouped a number of ScriptUI topics dealt with in the forums. (Though the list is out of date by now in 2019, it's still useful.)

Differences between Mac OS and Windows: palettes and windows

http://forums.adobe.com/thread/1103569?tstart=0

Change the colour of the background of buttons

http://forums.adobe.com/message/2335096#2335096

Resize windows

http://forums.adobe.com/message/2280793#2280793
http://forums.adobe.com/message/2741942#2741942
http://forums.adobe.com/thread/858153?tstart=0
http://forums.adobe.com/thread/957816?tstart=0
http://www.hilfdirselbst.ch/gforum/gforum.cgi?post=540693#540693

Icon buttons (see also Appendix 1 in this guide)

http://forums.adobe.com/message/1111746#1111746
http://forums.adobe.com/message/2326630#2326630
http://forums.adobe.com/message/2899148#2899148

Scrollable panels

http://forums.adobe.com/message/2899148#2899148
http://indesign-faq.de/de/ScriptUI_scrollbar

Progress bar

https://forums.adobe.com/thread/2089372

http://forums.adobe.com/message/3152162#3152162
https://gist.github.com/966103

edittext
http://forums.adobe.com/thread/964339?tstart=0

fonts
http://www.indiscripts.com/post/2012/05/scriptui-fonts-facts

colour
https://forums.adobe.com/thread/2159810

Miscellaneous
http://forums.adobe.com/thread/1260079?tstart=0

Interactive dialog builders

Interactive dialog builders are applications that allow you to construct a ScriptUI window without any knowledge of ScriptUI. The best one is Joonas Pääkkö's. You can find it at https://scriptui.joonas.me/. The link is to the on-line version, an off-line version can be downloaded from https://github.com/joonaspaakko/ScriptUI-Dialog-Builder-Joonas. You can contact Joonas on that GitHub address, and he'll discuss feature requests. The dialog builder is easy to use, is very capable, and is really the application of choice.

Slava Boyko's DialogBuilder is available at https://github.com/SlavaBuck/DialogBuilder. It was last updated in August 2014, and seems available only inRussian (I thought there was an English version as well, but I can't find it).

A slightly different approach is David Van Brink's Omino dialog builder. This is not an interactive dialog builder like the two apps mentioned above, but rather a kind of interface for ScriptUI. See http://omino.com/pixelblog/2008/09/21/35/

Acknowledgements

I would like to thank Mark Francis of Adobe for sharing his knowledge of ScriptUI and also the following people, most of whom are (or have been) regulars in Adobe's scripting forum: Marc Autret, Dirk Becker, Gabe Harbs, Mikhail Ivanyushin, Uwe Laubender, Kasyan Servetsky, Bob Stucky, Marijan Tompa, Vlad Vladila, and Ariel Walden for providing ScriptUI details and examples.

Appendix: Embedding graphic files in a script

Earlier we saw that the following line adds an icon from a file to an iconbutton in a window **w**:

```
w.add ("iconbutton", undefined, File ("/d/test/icon.png"));
```

In itself this is convenient, but you should really check if the icon file is present and provide some alternative just in case the file can't be found. So it would be convenient if we could embed those icons in the script – and we can.

The method outlined here was first described (to my knowledge) by Bob Stucky in Adobe's scripting forum; see Some useful forum topics, below. The method works for png, idrc, and jpg files; possibly for other formats as well, but I haven't tried them.

To embed a graphic in a script file we must convert the contents of the binary file to a string. The following script does that:

```
var infile = File ("/d/test_jpg/icon.png");
var outfile = File ("/d/test_jpg/icon.txt");
infile.open ("r");
infile.encoding = "binary";
var temp = infile.read();
infile.close();
outfile.open ("w");
outfile.write (temp.toSource ());
outfile.close ();
```

This results in a text file with a single, usually very long, line, which has the following form:

```
(new String(". . ."))
```

To use the string in a script, open the file in a plain text editor and strip (new String(from the beginning and)) from the end. Copy the string (including the quote marks) and paste it into the script, declaring it as a variable. This variable is used in the script instead of a reference to a file object. Our earlier code would then look as follows:

```
var mylcon = "/* very long string */";
var w = new Window ("dialog");
  w.add ("iconbutton", undefined, mylcon);
w.show();
```

That's basically all there's to it. But the conversion script, above, is a bit unwieldy, and I started using the following script, which is flexible in its input and strips the unwanted overhead away from the converted string.

```
function graphic_to_text (infiles /*array of file objects*/ ) {
  var outfile, s,
    re1 = /^\(new String\(/,
    re2 = /\)\)$/;

  for (var i = 0; i < infiles.length; i++) {
    if (infiles[i].exists) {
      outfile = File (infiles[i].fullName.replace (/\.(png|idrc)$/, '.txt'));
      outfile.open ('w');
      infiles[i].encoding = 'BINARY';
      infiles[i].open('r');
      s = infiles[i].read();
      outfile.write('var ' + outfile.name.replace ('.txt', '') + ' = ');
      outfile.write(s.toSource().replace(re1, '').replace(re2, ''));
      outfile.write(';');
      infiles[i].close();
      outfile.close();
    }
  }
}
```

The function's input is an array of file objects. Here are some examples:

```
graphic_to_text (Folder ("/d/test/").getFiles ("*.png"));
graphic_to_text ([File ("/d/test/iconA.png"), File ("/d/test/iconB.jpg")]);
graphic_to_text ([File ("/d/test/iconC.idrc")]);
```

The first example is correct since **getFiles()** returns an array of file objects. In the second example, we name two files ; in the third example we want to convert just one file, so we pass that in the form of a one-element array.

The converted graphics are written to files that use the same name as the graphics, but with the extention **.txt**. Each output file consists of a single line.

Note: when you paste those long strings into a script file in the ESTK you'll notice that the ESTK isn't particularly good at handling them. It may look as if part of the string has disappeared in the ESTK, but don't let that worry you, it's just a display problem.

Revision details – version 2.16

This revision is more a maintenance update than anything else. The only new item is mentioning Joonas Pääkkö's dialog builder, which is an extremely useful tool. For the rest I checked the links and refreshed the text here and there.

Index