The Super Widget Package

Creating serious GUI interfaces!

Introduction to the package

The Super Widget Package (SWP) provides a very easy-to-use mechanism to unleash the power to create Graphical User Interfaces (GUI's) via J/Link. It was originally based on the GUIKit, although since version 4.00 it no longer relies on that technology. It enables you to construct beautiful GUI interfaces to your *Mathematica* programs with the absolute minimum of effort.

The GUIKit is very powerful, in that it gives you access to the entire power of the Java graphical user interface. However, it is essentially a low level tool. It would take a lot of work to construct a serious *Mathematica* GUI application using GUIKit (or J/Link), and a lot of that work would be highly repetitious. It is important to realise that most of the pieces of example code in the GUIKit help information are incomplete, in that they are not linked up with any *Mathematica* code – you would need to add various `BindEvent` structures, which can quickly become quite complicated. The super widgets take all the typical component tasks of creating a GUI – such as displaying a variable in an editable box – and package them as neatly as possible.

The main design considerations for the Super Widget Package are:

• Extreme ease of use – I want even beginner *Mathematica* users to have access to GUI interfaces.

• Very traditional *Mathematica* syntax – e.g. no string option names are used, because you can't look them up with '?'. Likewise, colours are specified in the traditional way – e.g. using RGBColor.

• Absolutely no knowledge of Java should be required, except in order to use a few specialised advanced features.

• A very compact GUI notation.

The Super Widget Package is tested using *Mathematica* 6.0, 7.0, and will be upgraded as necessary to support future versions of *Mathematica*. Support for 5.2 was dropped at version 4.50, please contact the author if this is a problem.

Changes introduced at version 4.00

The SWP used to be based on the GUIKit. SWP code was translated into equivalent GUIKit code, plus some J/Link code and compiled Java. As the SWP progressed, more and more of its functions were performed by Java code – bypassing the GUIKit. At version 4.00 the GUIKit is no-longer used by the SWP. The main consequence of this is that GUI applications operate considerably more efficiently. This change has also opened the way to much further development which would have been difficult or impossible while using the GUIKit. Unfortunately, users should note that it is no-longer possible to mix GUIKit widgets with SWP code. Although this was possible with earlier versions, I have not heard of anybody actually using this facility.

These changes have also made it much easier to extend the SWP in new directions – possibly as part of projects with customers.

Getting started with the Super Widget Package

Installing the package

The SWP is supplied as a ZIP file containing all the files required in their appropriate directories.. It is vital that this directory structure is preserved. Copy the file either to the directory specified by $BaseDirectory, or $UserBaseDirectory (as a general rule, use the former if you are the sole user of the computer, use the latter if several people log on to your machine). Do not use the *Mathematica* directory itself, as was recommended in previous SWP releases, and ensure that you remove any previous version of the SWP located there. Unzip the file SuperWidgetPackage.ZIP using a tool that preserves the directory structure and handles long names correctly, e.g. PKZIP(R) Version 2.50, or WinZip(R). Never add or remove anything from the SuperWidgetPackage directory or any of its subdirectories, as this can disturb the operation of the package.

You can access the help by opening the Documentation Centre, and clicking in the "Installed Addons" link (bottom right of the window).

The examples in this guide are mostly very small – designed to illustrate a single point. A collection of rather larger examples is available at http://www.dbaileyconsultancy.co.uk/swp_examples/swp_examples.html. You are recommended to study these examples to understand the use of the SWP in a more realistic context.

Loading the package

The SWP may be loaded thus:

```
Needs["SuperWidgetPackage`"]
```

Loading this package automatically loads J/Link.

This checks the SWP version number.

```
SuperWidgetVersion[]
```

```
6.24
```

Simple use

Here is a very trivial SWP example:

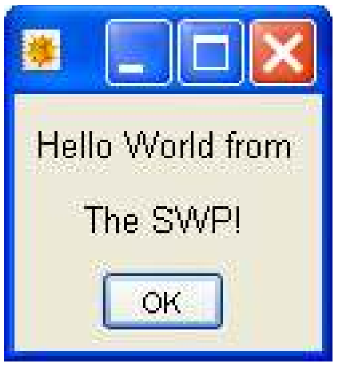
```
Needs["SuperWidgetPackage`"]
```

```
{{"Hello World from"},
  "The SWP!"} // SuperGUIRunModal
```

Alternatively, here is a very slightly more complicated example that looks a lot more attractive because the text and the button are laid out sensibly:

```
Needs["SuperWidgetPackage`"]
```

```
{¶10,
  {¶0, "Hello World from", ¶0},
  ¶10,
  {¶0, "The SWP!", ¶0},
  ¶10,
  {¶0, SuperWidgetButton[Null, "OK", 1], ¶0},
  ¶10} // SuperGUIRunModal
```



Special support for versions 5 and 6

Under versions of *Mathematica* prior to 6, graphics functions such as `Plot` created a visible plot as a side-effect of their operations, while returning a graphics structure as their formal result. This structure could be displayed subsequently using the `Show` command. Under version 6, graphics does not rely on side-effects – a graphics structure is displayed as such by the frontend. Although this manual is mainly concerned with GUI graphics, in a few places the function `V6`Show` is used to smooth over these differences. `V6`Show` is defined in the Super Widget Package, but is really only useful in the context of this manual.

Some basic GUI concepts

This section contains a brief overview of GUI terminology. While this user-guide avoids jargon where possible, various terms with a particular meaning in this context are underlined as they are defined, and will be useful in what follows.

A program is said to have a Graphical User Interface (GUI) if it communicates with the user via a windowed interface so that the user fills data into boxes, and uses buttons, menus, etc. Although the *Mathematica* front end is obviously a GUI application itself, programs written in *Mathematica* code have traditionally not been able to offer a full GUI interface to their users.

A typical GUI window contains many individual components – such as menus, buttons, data-entry boxes, icons (small images), etc. Each of these entities is known as a widget. This terminology is traditional in the Unix world, however Windows programmers typically refer to widgets as 'controls'.

Using the GUIKit, it is possible to assemble collections of widget definitions, together with some layout information to create complete windows, which are displayed using the J/Link Java interface. A complete GUI application may consist of one or more windows, which may be displayed simultaneously, or at different times as the program executes.

GUIKit widget definitions can become quite complicated by the time they are actually ready to use. For example, if you look up the "TextField" widget in the GUIKit help it looks fairly simple to code, but does not actually transmit any user input back to the *Mathematica* program without considerable additional complications! This is partly because the GUIKit reflects the underlying Java classes, which require a Java program to make them do anything useful.

The SWP enables GUI's to be built using super widgets, which are very much easier to construct than GUIKit widgets, and also offer considerable extra functionality. In what follows, the SWP will be explained with very little reference to the underlying Java or J/Link – the technologies that underlie the SWP.

GUI windows can be either modal or modeless. A modal window takes control when it is displayed (the program can be thought of as entering a different mode while the window is visible) and other windows in the same application are rendered temporarily inactive. If you alter a notebook and then try to close it without saving it you will see an example of a modal window – you have to press one of its buttons before you can proceed. Conversely, the *Mathematica* palettes are good examples of modeless windows. A palette can sit about for as long as you like, ready to use when required. When you create a modal window, the window (if any) that 'caused' the event (say by pressing a button) will be disabled for the life-time of the new window. This means that its widgets will not respond and that it cannot overlay the new window. This is, of course exactly what is required by a modal window, because it must be attended to before the program can proceed. From a programming point of view, when you create a modal window, you pass control to that window, and you only gain control when the window closes. This means that a function that creates a modal window can return a result.

Widgets and menu items are not always usable. For example, there is no point in activating a file-save widget if there is nothing to save! While it would be possible to simply ignore such 'user mistakes' – or even generate an error message – this is not very user-friendly. You must anticipate that your user will explore your program, clicking all over the place to see what happens. It is for this reason, that most GUI applications disable widgets or menu options when they can't currently be used. This is also sometimes known as 'greying out', because the relevant item is rendered in a faded fashion, and does nothing if clicked.

All the individual widgets are stored in one big widget, which represents the window itself – think of it as a widget box. There are two possibilities – a frame window or a dialog window. Traditionally, dialog windows tend to be used for modal interactions, but either window can be modal or modeless. The dialog window is actually rather less flexible because it cannot have a menu, and cannot be minimised or maximised. However, if you are putting up a simple query or information box, maximisation and/or minimisation are inappropriate, and a dialog box looks better.

If you look at a typical GUI window – such as a notebook, you will see that it contains two active points. There will be a mouse cursor, which will obviously move as the mouse moves, and the so-called caret – a vertical bar which indicates the point at which keyboard input will appear. The place to which keyboard data will be sent is said to have input focus. The input focus can be moved from widget to widget by clicking with the mouse (but not by mere mouse movements) – thus it is normal to click into an input field in order to adjust its value. The input focus can also be moved using the TAB key. It is instructive to try this. You will see that the input focus cycles round the various widgets (but not all can accept input focus). In particular, the input focus can be moved to a button. In this state the button is emphasized in appearance, and can be 'pressed' by using the ENTER or RET keys.

When constructing a GUI interface, it is important to make it look as much like other simple GUI applications as possible. A lot of the value of a GUI interface is that a user can guess how a program will behave, using his or her experience with other applications. Thus, for example, the file/open/save/print/exit menu in the left-most position is very traditional, nobody will thank you for putting these menu items in another location! Traditional GUI's also frequently supply several ways of doing the same operation. For example, it may be possible to specify a file/save operation via the menus or using the toolbar. This makes sense because people vary greatly in the ways in which they prefer to interact with a GUI.

The more you critically examine a typical GUI, the more complexity you will find. Fortunately, most of this functionality is catered for by the default settings of the super widgets, and the job is less daunting than it might at first appear.

Using Simple Super Widgets

Introduction

The super widgets are all completely inert (analogous to, say, the `RowBox`) until passed to the `SuperGUIRun` or `SuperGUIRunModal` functions. Use `SuperGUIRunModal` unless you specifically require the modeless properties of `SuperGUIRun`. In this case, please read the section on the restrictions that apply to modeless windows.

Every super widget is set up with attribute `HoldFirst`. The first argument of each super widget must be the name of a *Mathematica* variable, or (as of version 4.70) an simply indexed array - such as x[[5]]. This variable, known as the associated variable is used to identify the widget and to hold the data represented by the super widget. Even widgets that do not have associated data, still require an associated variable as the first argument. It is extremely important that a different associated variable is used for each super widget. Sometimes the associated variable is not useful, and in these cases it can be coded as Null, and the SWP will invent a suitable (unique) replacement.

Most super widgets can take a variety of options. These options are traditional symbolic options, and can be examined and adjusted using `Options` and `SetOptions`.

Tip! Many super widget arguments or options are the names of functions. These functions are typically called when some action is required – say in response to a button. Since most of these functions need to be defined with the `HoldFirst` attribute, the SWP does not accept pure functions in these places. There is a syntax available to define a pure function with attributes, but this is quite clumsy.

SuperGUIRun and SuperGUIRunModal

`SuperGUIRun[widgets,opts]`	Displays one or more super widgets as a modeless window, and returns to the program immediately. Modeless windows are exceptional – use SuperGUIRunModal is in doubt.
`SuperGUIRunModal[widgets,opts]`	Displays one or more super widgets as a modal window, and returns when the window is ultimately closed. The return value is Null, or the button number or ReturnAction that closed the window (see below).

Functions to display super widgets

Both these functions take a widgets argument. This can be a single super widget, or a list (usually nested) of super widgets mixed with layout and ancillary information. Alternatively, the widgets argument may consist of one `SuperWidgetFrame` or `SuperWidgetDialog` (which represent whole windows – see below), each of which contains an embedded list of the super widgets inside them.

`ConcealNotebooks`	Option for SuperGUIRunModal. Hides the notebooks while the GUI is on display. Read the notes about this option before use.
`On⌄Display`	Option for SuperGUIRunModal only. Specifies a function with no arguments that is to be called immediately after the GUI has been displayed. Note that this option is not available for SuperGUIRun – where it would serve no purpose.

Options for SuperGUIRun and SuperGUIRunModal

`SuperGUIRunModal` can take the `ConcealNotebooks` option. This hides the currently open notebooks before displaying the window (so the notebooks cannot obscure the window) and restores the notebooks afterwards. This option is best used in fully debugged applications, as there is a chance that if your window hangs, you will have difficulty recovering your notebook (assuming it contained unsaved material).

It is generally much safer and easier to use `SuperGUIRunModal` rather than `SuperGUIRun`, unless you genuinely want to continue doing something while the GUI is on display. If you need to perform a fairly quick task after the GUI is displayed, it is recommended that you use the `On⌄Display` option to supply a startup function. The GUI will not become fully responsive until after any startup function has completed.

SuperWidgetIntegerBox, SuperWidgetRealBox, SuperWidgetStringBox

`SuperWidgetIntegerBox[v,opts]`	Displays the decimal integer in the associated variable v in a text edit box. As the user edits the number, the value of v changes.
`SuperWidgetRealBox[v,opts]`	Displays the machine real in the associated variable v in a text edit box. As the user edits the number, the value of v changes.
`SuperWidgetDecimalBox[v,ndigs,opts]`	Displays the machine real in the associated variable v in a text edit box, but with a fixed number – ndigs – of digits after the decimal point. Exponent form is not permitted. As the user edits the number, the value of v changes.
`SuperWidgetStringBox[v,opts]`	Displays the string in associated variable v in a text edit box. As the user edits the string, the value of v changes.

Basic data input widgets

`ChangeFunction`	1-Argument function to call with the associated variable each time. the user alters the data
`Character⌄Width`	Specifies the number of character positions used for the box.
Editable	Set False to create a data box that can only be modified by code – not by using the mouse/keyboard.
`Select⌄`All	Set True to cause the entire box to be initially selected – this means that typing into the field will cause the selection to be deleted.
`Tool⌄Tip`	Supplies a tool-tip (helpful string) to be displayed when the mouse enters the box.
`Return⌄Action`	Action to be taken if the user presses the ENTER or RET while the widget has focus. Set to an integer to cause the window to close with that as the return value, or supply the name of a function to be called with the widget associated variable as argument. Enabling data input windows to close in this way is very convenient because the user's hands are already on the keyboard.
`Stretch`	Default `{False,False}` – determines if the widget can expand in the horizontal and vertical directions to fill space. For one−line controls of this sort, it can be useful to specify `{True,False}`, but specifying vertical stretch is not useful.

Options for the basic data input super widgets

These four super widgets create an edit box specialised to accept data of a particular kind. They take an associated variable which should start off either with no value, or with an integer/real/string value as appropriate. The widget will use the value of `v` for initialisation, or will start off blank. As the user changes the value, the associated variable, `v` will update, and the option `ChangeFunction` can be used to execute code each time the variable is updated. Note that since the value is automatically transmitted to the associated variable, the `ChangeFunction` option is typically only required for more complex applications.

Note that there are two super widgets for real numbers. `SuperWidgetRealBox` is useful for regular real numbers, and can handle exponents, whereas `SuperWidgetDecimalBox` handles reals that have a specific maximum number of decimal digits – such as currency. Numbers are right-aligned in the `SuperWidgetDecimalBox`.

Remember to load the SWP package before executing the following example. First, we turn on snapshot mode (explained in detail later) so that we can record an image (typically a little larger than the live GUI) of the resulting window:

```
Needs["SuperWidgetPackage`"]
```

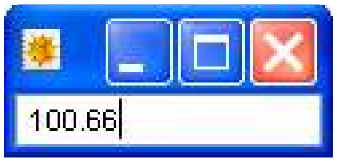
```
SetSnapshotMode[True]
```

The variable x will get passed to the function f each time the user changes the number. A complete program might contain many `SuperWidgetRealBox` widgets, so it is useful to give `f` the attribute `HoldFirst` so that it is possible to determine which widget changed as well as its value.

```
SetAttributes[f, HoldFirst];
f[a_] := Print[HoldForm[a], "=", a];
x = 100.5;
{SuperWidgetRealBox[x, ChangeFunction → f]} // SuperGUIRunModal
```

```
x=100.54

x=100.544
```

```
GetSnapshots[] // V6⌣Show
```



100.66

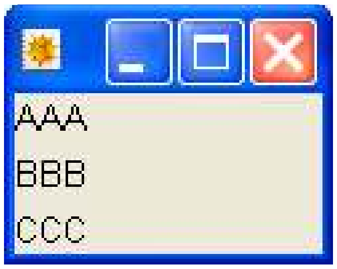
Note the Return⌣Action option. You can give your user a better experience if you use this option. The data supplied by this option is exactly analogous to the third argument of SuperWidgetButton. It enables you to make the ENTER or RET keys mimic the action of a button – typically the button which marks successful completion of the data entry process. If your window contains multiple data-entry super widgets, it is, of course, sensible to use the same value for the Return⌣Action option in each case.

For more advanced manipulation of basic data input widgets see the section "Dynamic manipulation of basic data input widget properties".

Layout

The layout mechanism is derived almost completely from the GUIKit, on which the SWP used to be based. Super widgets are combined by using a list. A simple list of super widgets (or indeed GUIKit widgets) are displayed vertically. For example:

```
Needs["SuperWidgetPackage`"]

{"AAA", "BBB", "CCC"} // SuperGUIRunModal

GetSnapshots[] // Show;
```



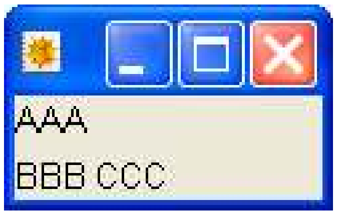
AAA
BBB
CCC

Note that this example uses the fact that simple text strings are converted to SuperWidgetLabel objects and displayed as read-only text.

If you embed super widgets in nested lists, the layout flips between vertical and horizontal. For example:

```
{"AAA", {"BBB", "CCC"}} // SuperGUIRunModal
```

```
GetSnapshots[] // Show;
```



Within the lists, super widgets are spaced out using ¶$_{10}$ (the size of the gap, 10 is arbitrary, but seems to produce good results in most cases – use a bigger or smaller value as required). The spacing will be horizontal or vertical according to the list nesting. While this notation may seem rather cryptic, real GUI's often require a lot of spacing to look right, and the standard GUIKit notation – `WidgetSpace[10]`, (which will also work) can become extremely cumbersome.

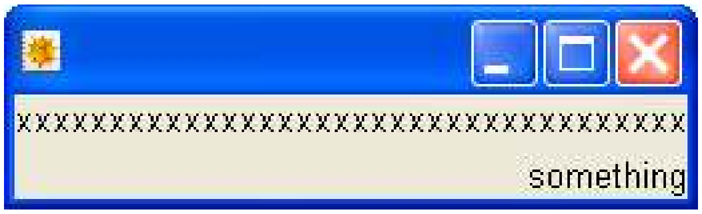
Tip! Your window will look more attractive if you use one, or at most two, different fixed-width space elements. You may achieve this by assigning a variable – such as `sp` – thus:

```
sp = ¶10
```

Using this variable where space is required will ensure consistency and make it easy to adjust the spacing later, if required.

The notation ¶$_0$ has a special meaning, it corresponds to the GUIKit `WidgetFill[]` and supplies filling space as required. Thus, you get right justification by filling from the left:

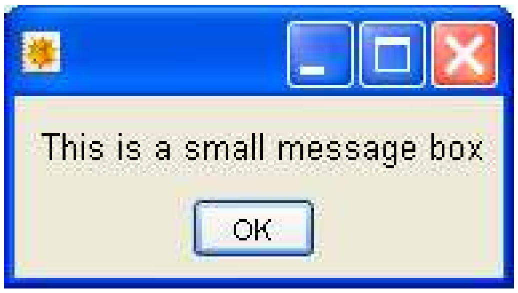
```
{"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx", {¶0, "something"}} // SuperGUIRunModal

GetSnapshots[] // Show;
```



It is particularly useful to fill from both the left and the right to produce a centred widget. This is well illustrated using a button super widget that will be described in full later:

```
{¶10, {¶10, "This is a small message box", ¶10}, ¶10,
  {¶0, SuperWidgetButton[Null, "OK", 1], ¶0}, ¶10} // SuperGUIRunModal

1
```

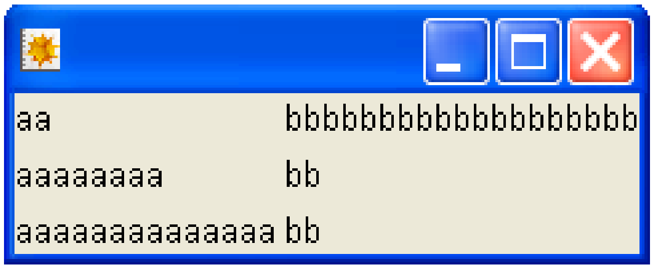
```
GetSnapshots[] // Show;
```



Note that simple message boxes like this are more easily produced using the `ShowMessageBox` function, described later.

Another useful layout feature is the GUIKit `WidgetAlign[]`, which can be represented more neatly using TAB. For example:

```
{{"aa", TAB, "bbbbbbbbbbbbbbbbbbb"}, {"aaaaaaaa", TAB, "bb"}, {"aaaaaaaaaaaaaa", TAB, "bb"}
} // SuperGUIRunModal

GetSnapshots[] // Show;
```



All the 'b' strings have been forced to line up because of the tabs.

By default, some super widget types are stretchable (e.g. graphics panels), while others are not. The option `Stretch` can be used to override this default as required. For example, `Stretch->{True,False}` indicates that the widget in question can be stretched horizontally but not vertically. Note that not all widgets can be meaningfully stretched.

More complex layouts are possible using `SuperWidgetPanel`, `SuperWidgetLabelledBox`, and `SuperWidgetTabPanel`

SuperWidgetHorizontalSeparator / SuperWidgetVerticalSeparator

These two super widgets can be used to 'score' a line into a window in such a way that they act as a separator. These should not be used on windows with a white background.

`SuperWidgetHorizontalSeparator[v,opts]`	Displays a horizontal scored line that acts as a separator. The variable v is not currently used.
`SuperWidgetVerticalSeparator[v,opts]`	Displays a vertical scored line that acts as a separator. The variable v is not currently used.

The separator super widgets

SuperWidgetHorizontalSlider / SuperWidgetVerticalSlider

`SuperWidgetHorizontalSlider[v,min,max,opts]`	Displays a horizontal slider widget to represent real values between min and max. If v has an initial value, it is used to set the slider initially, otherwise it starts at the half−way point. The associated variable v is updated when the slider is moved, and if the ChangeFunction option is used, the supplied, function is called with the v as argument.
`SuperWidgetVerticalSlider[v,min,max,opts]`	Displays a vertical slider widget.

The Slider super widgets

`ChangeFunction`	1-Argument function to call with the associated variable each time. the user alters the data
`Slider␣Labels`	Supplies a list of label values to be used to annotate a slider. Tick marks are also added if this option is used.
`Snap␣To␣Ticks`	Set to True to cause the slider to snap to the nearest tick position when the mouse is released. Also causes the slider to delay reporting until the mouse button is released.
`Tick␣Ratio`	Ratio of major to minor tick spacing – default 5. Set this value to 1 to effectively remove the minor ticks
`Tool␣Tip`	Supplies a tool-tip (helpful string) to be displayed when the mouse enters the box. The displayed tool tip is concatenated with the current value.
Zoom␣Menu	When this option is set to True, the slider can be made to zoom using a right mouse click (or equivalent on non−Windows platforms) to bring up the zoom menu. This option generates its own labels – do not use with the Slider␣Labels option.

Options for the Slider super widgets

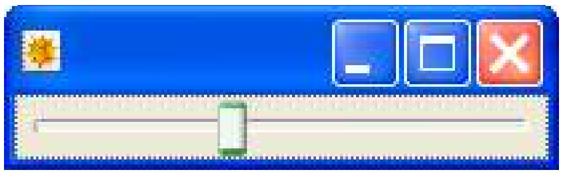
The slider super widgets take a Real-valued associated variable followed by the min and max values. As the slider is moved, the variable updates, and if the `ChangeFunction` option has been used, that is also called:

```
Needs["SuperWidgetPackage`"]
```

```
SetAttributes[foo, HoldFirst];
foo[x_] := Print[Unevaluated[x], "=", x];
y = 0.4;
SuperWidgetHorizontalSlider[y, 0, 1.0, ChangeFunction → foo] // SuperGUIRunModal;
y
```

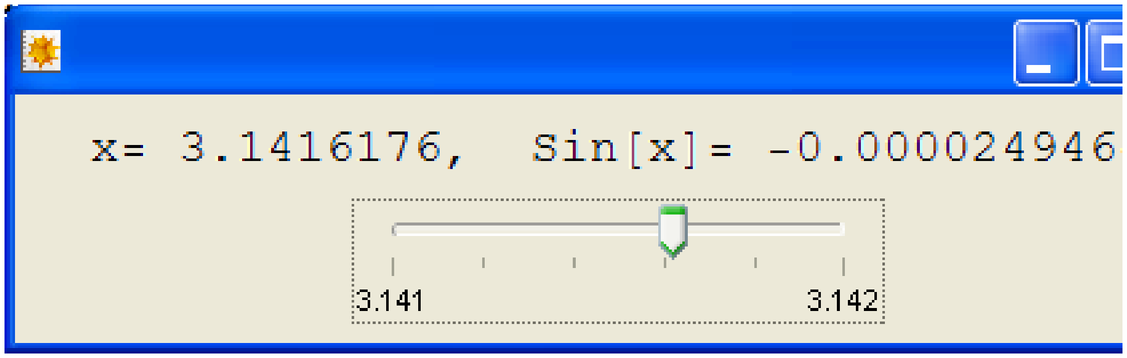
```
    y=0.4

    y=0.4032

    y=0.4194

    y=0.4247

    y=0.4301

    y=0.4355

    y=0.4462

    y=0.457

    y=0.4462

    y=0.4409

    y=0.4301

    y=0.4247

    y=0.4194

    y=0.4086

    y=0.4032

    y=0.4032

0.4032
```

```
GetSnapshots[] // Show;
```



The Zoom⌣Menu option enables you to use sliders in a variety of interesting ways. Here is a very trivial example in which it is used to 'manually' find the root of an equation:

```
Clear[f];
labelText[] :=
 (
  Image⌣Expression[StyleForm[SequenceForm["x= ", NumberForm[x⌣val, {10, 10}],
     ",  Sin[x]= ", NumberForm[Sin[x⌣val], {10, 10}]], FontSize → 18]]
 )

f[x_] := Set⌣Label⌣Contents[xxx, labelText[]];
x⌣val = 3.0;
{
  ¶10,
  {¶0, SuperWidgetLabel[xxx,
    labelText[], Label⌣Alignment → Center, Pixel⌣Width → 450], ¶0},
  ¶10,
  {¶0, SuperWidgetHorizontalSlider[
    x⌣val, 3.0, 3.5, ChangeFunction → f, Zoom⌣Menu → True], ¶0},
  ¶10
 } // SuperGUIRunModal
```



By moving the slider until the value of `Sin[x]` is closest to zero, and then right clicking on the slider to zoom it in, it is possible to home in on the root of `Sin[x]==0` (i.e. π) with steadily increasing accuracy. The image shows the display some way into this process. Although there is not much point in using this method for root finding, zoomable sliders can be very useful in other situations in which the location of interest is less easily characterised – for example, finding the value of a parameter that just makes a fractal disconnected.

Note that in the above example, the text is output as an image because the numbers are displayed in exponent form as the zoom progresses. This image is placed in a `SuperWidgetLabel` with an explicit `Pixel⌣Width` to allow for the numbers to occupy a little more space if necessary.

SuperWidgetButton

`SuperWidgetButton[v,name,fn,opts]`	Displays a button where name can be either a string or an image construct. When pressed, if fn is an integer, the button closes the whole window and returns that integer as an argument, otherwise s call is made to fn[v].

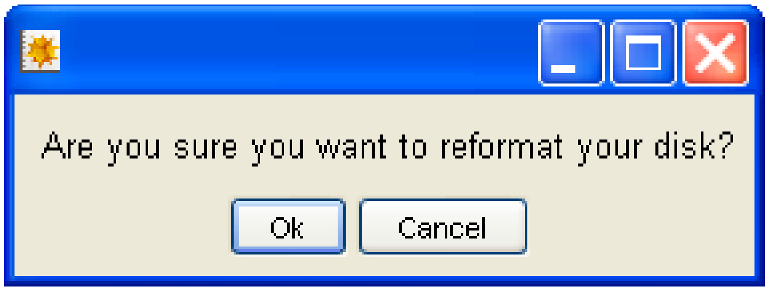
The Button super widget

Press⌄Function	Supplies a function to be called when the button is depressed (the normal action occurs as it is released again). Useful in push and hold applications.
Tool ⌄Tip	Supplies a string to be displayed as a tool tip when the mouse enters . the widget

Options for the Button super widget

This creates a button. The first argument is the associated variable (or Null), used to identify the widget, and the second is the text (or image) of the button. This text string should contain only ordinary characters – not special *Mathematica* characters which cannot be displayed in Java, however, it is easy to convert *Mathematica* expressions as images, so this is no real limitation. The final argument can be either an integer or the name of a function. Using an integer, the button will close the whole window if it is pressed, and SuperGUIRunModal will return this integer as its result. This can be very convenient in situations in which a window can be dismissed by one of several buttons. For example:

```
Needs["SuperWidgetPackage`"]

SuperWidgetFrame[fr1, {
   WidgetSpace[10],
   {WidgetSpace[10], SuperWidgetLabel[l1,
     "Are you sure you want to reformat your disk?"], WidgetSpace[10]},
WidgetSpace[10],
{WidgetFill[], SuperWidgetButton[bb1, "Ok", 1],
    SuperWidgetButton[bb2, "Cancel", 2], WidgetFill[]},
   WidgetSpace[10]
  }
 ] // SuperGUIRunModal

1

GetSnapshots[] // Show;
```


Are you sure you want to reformat your disk?

This will return 1 if the 'Ok' button is pressed, 2 if the 'Cancel' button is pressed, and Null if the window is closed in some other way.

Simple message boxes can be shown using an even simpler method using the function ShowMessageBox. For example:

```
ShowMessageBox["Are you sure you want to reformat your hard disk?",
 "Careful!", {"OK", "Cancel"}]

1
```

You can specify any number of buttons, and these produce return values of 1,2,...

```
SuperWidgetFrame[fr1, {
    ¶10,
    {¶10, SuperWidgetLabel[l1, "Are you sure you want to reformat your disk?"], ¶10},
¶10,
{¶0, SuperWidgetButton[bb1, "Ok", 1], SuperWidgetButton[bb2, "Cancel", 2], ¶0},
    ¶10
   }
 ] // SuperGUIRunModal
```

```
1
```

The third argument to `SuperWidgetButton`, which determines what the button actually does is often made the same as the `Return⌣Action` option used elsewhere in the window. This makes it possible to mimic the pressing of a button by pressing the RET key. Typically you should arrange for this key to produce the same behaviour as pressing the 'OK' button.

The option `Press⌣Function` creates a button with rather different functionality. Ordinary buttons do nothing as they are pressed – the action happens as they are released. Indeed, if you press an ordinary button with the mouse and then move the mouse off the button before you release the key, you will release the button without performing the corresponding action. The `Press⌣Function` is for cases where a button is to be pressed and held for the duration of a process. In this case the function supplied to `Press⌣Function` normally initiates some action, and the normal button function terminates it. The SWP ensures that in this case, if the button is pressed and the mouse leaves the button, the normal mouse click function is called at that point. This ensures that such a button cannot be left stuck on.

The label of a button can also be an image (see Images) or a general HTML string (see Using HTML strings inside widgets) to obtain more exotic effects.

Buttons may also be coloured after they have been created. This is useful to designate which button has most recently been pressed, or for other reasons. Light colours usually look best:

`Set⌣Button⌣Colour[v,colour]`	Sets the colour of the button with associated variable v

The colour is combined with the ordinary button image, and the default is `White`.

SuperWidgetLabel

`SuperWidgetLabel[ v,contents,opts]`	Displays static text or an image, specified by contents. The associated variable may be supplied as Null, if not required.

The label super widget

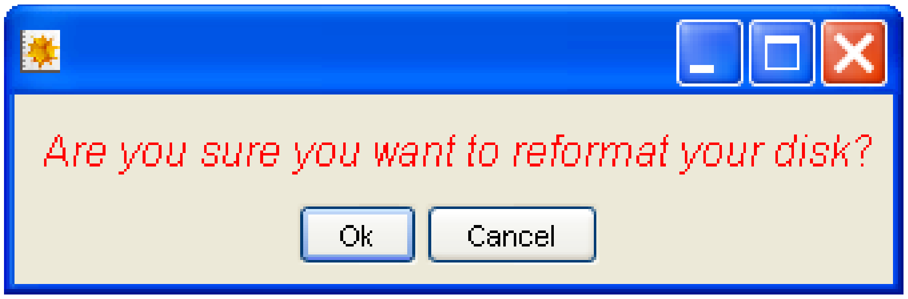
`Font˽Color`	*Mathematica* colour specification (e.g. RGBColor[....] for the text.
`Font`	Font specification in the form {Name,face,size}
Label˽Alignment	Specifies the horizontal alignment of the label contents. Defaults to `Left`, but `Center` or `Right` can be specified.
Pixel˽Width	Specifies the minimum width of the label in pixels – particularly useful if the label contents may be replaced with a longer label.

Options for the label super widget

Most `SuperWidgetLabels` are not coded explicitly. Whenever you include a string (or an image – see below) in a list of super widgets, this is translated into a `SuperWidgetLabel` before it is displayed. However, using the explicit formulation it is possible to adjust their properties using options.

The super widget label is used to place fixed text strings in a GUI. The first argument – the associated variable – is not really used at present, but it seemed more consistent to make this super widget follow the same convention as all the others. Unlike the "label" widget (into which it resolves), the super widget uses the super widget default font (SWDF). This can be adjusted by calling `Set˽Text˽Font`. Alternatively, the font may be set on a per-label basis by using the Font option. By default, the SWDF is a little larger than the default Java font. The `Font˽Color` (or `Font˽Colour`, for UK users!) option can be used to specify the text colour using an `RGBColor` object. For example:

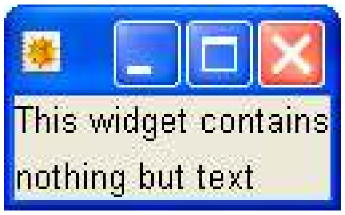
```
Needs["SuperWidgetPackage`"]

SuperWidgetFrame[fr1, {
    ¶10,
    {¶10, SuperWidgetLabel[Null, "Are you sure you want to reformat your disk?",
      Font˽Color → RGBColor[1, 0, 0], Font → {"Helvetica", "Italic", 16}], ¶10},
¶10,
{¶0, SuperWidgetButton[Null, "Ok", 1], SuperWidgetButton[Null, "Cancel", 2], ¶0},
    ¶10
   }
 ] // SuperGUIRunModal

1

GetSnapshots[] // Show;
```


Are you sure you want to reformat your disk?

If no special options are required, a label may be specified as a simple string. For Example:

```
{"This widget contains", "nothing but text"} // SuperGUIRunModal
```

```
GetSnapshots[] // Show;
```



Labels can contain newline characters ("\n"). In this case multiple lines of text are displayed with left justification.

The contents of a label can be changed by calling the function `Set⌄Label⌄Contents`, subject to the following restrictions:

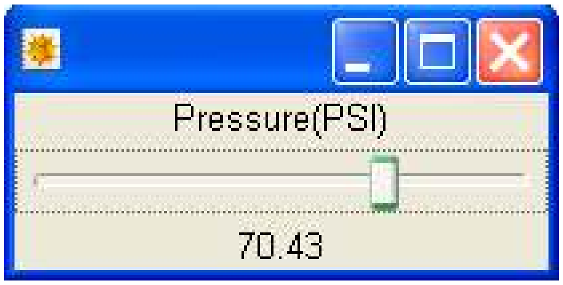
• A text-label cannot be converted into an image-label (or vice-versa) by calling this function.

• Multi-line text labels (i.e. where the text contains newline characters) cannot be updated.

• Labels are not re-sized to fit their new contents (since the window is already on display at this point!) so make sure that the initial contents are large enough, or (as in the example below) that the label will be stretched by the layout rules to an adequate size.

In this example, the change function for the slider updates a label to indicate the value change.

```
xxx = 50.0;
foo[x_] := (
   Set⌄Label⌄Contents[lll, ToString[x]];
  );

SuperGUIRunModal[{SuperWidgetLabel[Null, "Pressure(PSI)", Label⌄Alignment → Center],
  SuperWidgetHorizontalSlider[xxx, 0., 100., ChangeFunction → foo],
  SuperWidgetLabel[lll, ToString[xxx], Label⌄Alignment → Center]
 }]

GetSnapshots[] // Show
```



Labels can also contain HTML strings – see the section on using HTML strings inside super widgets.

Using images

`Image⌄File[file-name]`	Represents an image stored in a file. Use a complete path name rather than a local file name.
`Image⌄Expression[expr]`	Displays static text or an image, specified by contents. The associated variable may be supplied as Null, if not required.
`Image⌄Expression[expr,form]`	Represents an image of the given expression in StandardForm. Use HoldForm if you need to display an expression that would undergo evaluation.
`Image⌄Boxes[boxes]`	Represents an image of the boxes (e.g. created by ToBoxes).
`Image⌄Graphics[g]`	Represents an image consisting of a 2 or 3–D *Mathematica* graphics object.
`Image⌄String[string]`	Represents an image in the form of a string, typically created by ExportString.

Image wrappers

So far the buttons and labels have all been textual. There are a number of places where super widgets can accept images as well as strings. An image can just be a pretty image in a file, but it can also be an image of an expression – so that a button or a label can access the full character set and expression mechanisms of *Mathematica*. Images can be represented in the following ways:

• As a file, e.g. `Image⌄File`["c:\\gifs\\test.gif"]. Jpeg, PNG, and GIF files are accepted, and GIF files can even be animated!

• As an expression, e.g. `Image⌄Expression[Sqrt[x+1]]`

• As an expression in a particular form, e.g. `Image⌄Expression`[`Gamma`[x],`TraditionalForm`]

• As a box expression (typically created by `ToBoxes`), e.g. `Image⌄Boxes[ToBoxes[Sqrt[x+1]]]`

• `As a graphics object (typically created by Plot, Plot3D, etc.).`

• As a string, as generated by `ExportString`, e.g. `Image⌄String[x]`

These 'functions' do not evaluate in themselves, they merely represent an image in a super widget. For example:

```
Needs["SuperWidgetPackage`"]

Clear[x]; SuperWidgetLabel[pp, Image⌄Expression[Sqrt[x^2 + 1]]] // SuperGUIRunModal
```

Here is a more complicated example in which the value of both a text label and an image label are updated as the slider is adjusted.

```
Clear[f];
aaa = 0.0;
do⌣plot[] := ParametricPlot[{Cos[5 t], Sin[(1 + aaa) t]},
   {t, 0, 2 π}, AspectRatio → Automatic, DisplayFunction → Identity];
xxx = do⌣plot[];
f[_] := (
   xxx = do⌣plot[];
   Set⌣Label⌣Contents[lll, ToString[aaa]];
   UpdateWidgetValue[xxx];
  );

{SuperWidgetGraphicsPanel[xxx],
  SuperWidgetHorizontalSlider[aaa, 0,
   10, ChangeFunction → f, Slider⌣Labels → {0, 2, 4, 6, 8, 10}],
  SuperWidgetLabel[lll, ToString[aaa], Label⌣Alignment → Center],
  ¶10
 } // SuperGUIRunModal
```

Using fonts

A `SuperWidgetLabel` can be defined to use a particular font (see above). However, it is important to realise that these are Java fonts, not *Mathematica* fonts. It is usually better to set up a `StyleBox` object using ordinary *Mathematica* fonts, and then wrap the result in `Image⌣Boxes`.

SuperWidgetComboBox

`SuperWidgetComboBox[ v,contents,opts]`	Displays a combo box using the list of strings contents. The selected string is placed in the associated variable.

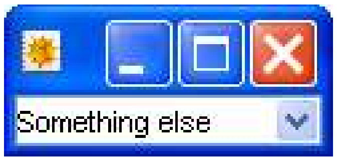
The combo box super widget

`ChangeFunction`	Function that is called (with the argument v) each time the user makes a new selection or edits the selection.
`CharacterWidth`	Specifies the number of character positions used for the box. This is a minimum size – if the list contains long strings, the box may be wider.
`Editable`	Specifies if the user can type in a value or if he can only select from a predetermined list (default).
`ReturnAction`	Action to be taken if the user presses the ENTER or RET while the widget has focus. Set to an integer to cause the window to close with that as the return value, or supply the name of a function to be called with the widget associated variable as argument. Enabling data input windows to close in this way is very convenient because the user's hands are already on the keyboard.
`ToolTip`	String to be used as a tooltip.

Options for the combo box super widget

This creates a combo-box – a small box that can drop down to display a set of alternatives (which must be strings). The first argument is the associated variable. It is updated each time the user selects an item. Its initial value is not used. If the option `Editable->True` is used, the user can also type in a value, which may or may not be on the list. The `ChangeFunction` option can be used to monitor the changes. The supplied function is called with the controlling variable as argument. For example:

```
Needs["SuperWidgetPackage`"]

SetAttributes[myFunc, HoldFirst];
myFunc[x_] := Print[Unevaluated[x], "=", x];
x =.;
SuperWidgetComboBox[x, {"Alpha", "Beta", "Gamma"},
   Editable → True, ChangeFunction → myFunc] // SuperGUIRunModal;
x

   x=Beta

   x=Beta

Beta

GetSnapshots[] // Show;
```



At version 3.40, the SuperWidgetComboBox acquired additional capabilities supplied by the following functions:

`ComboBox⌣Index[var]`	Returns the index of the selected item in the combobox super widget associated with var – only useful if Editable->False.
`Select⌣All[var]`	Selects all the text in the edit box of the combobox super widget associated with var – only useful if Editable->True.
`Set⌣ComboBox⌣List[var,list]`	Replaces the list of strings in the combobox super widget associated with var.
`Last⌣Focus⌣Time[var]`	Returns a representation in miliseconds of the time when the widget associated with variable vvar last acquired focus.

Functions to manipulate combobox super widgets

The contents of combo boxes can be defined by HTML strings – which provide a way to use images in such boxes. For convenience, the ChangeFunction of a combo box can call `ComboBox⌣Index` to determine the item selected as a number rather than a string. This function should not be called at other times or when the `Editable` option has been set.

SuperWidgetCheckBox

`SuperWidgetCheckBox[v,description,opts]`	Displays a check box with the given description. The associated variable reflects the value as True or False.

The check box super widget

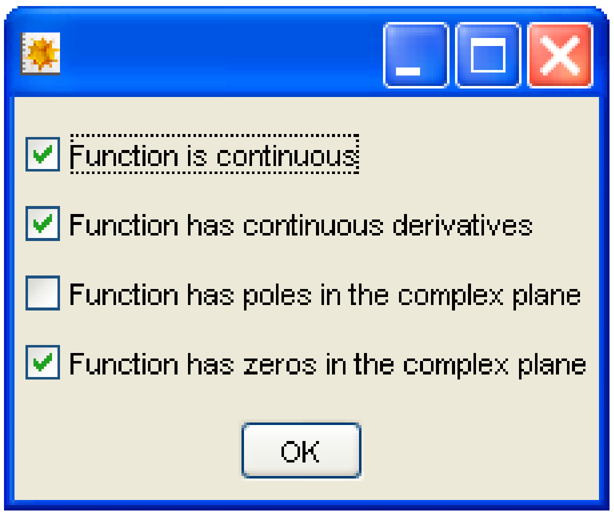
`ChangeFunction`	Function to call (with v as argument) each time the user changes the state of the widget.
`Tool⌣Tip`	String to be used as a tooltip.

Options for the check box super widget

This creates a check box with some associated text. It is used to obtain a simple boolean value. For example:

```
Needs["SuperWidgetPackage`"]

v1 = True;
v2 = True;
v3 = False;
v4 = True;
 {¶10,
   SuperWidgetCheckBox[v1, "Function is continuous"],
   SuperWidgetCheckBox[v2, "Function has continuous derivatives"],
   SuperWidgetCheckBox[v3, "Function has poles in the complex plane"],
   SuperWidgetCheckBox[v4, "Function has zeros in the complex plane"],
   ¶10,
   {¶0, SuperWidgetButton[Null, "OK", 1], ¶0},
   ¶10
  } // SuperGUIRunModal

1
```

```
GetSnapshots[] // Show;
```


☑ Function is continuous
☑ Function has continuous derivatives
☐ Function has poles in the complex plane
☑ Function has zeros in the complex plane

SuperWidgetRadioButtonGroup

`SuperWidgetRadioButtonGroup[v,contents,opts]`	Displays a group of radio buttons with names taken from the list contents. The associated variable is an integer indicating which button is currently pressed.

The radio button group super widget

`ChangeFunction`	Function to call with v as argument, each time the user clicks one of the buttons.

Options for the radio button group super widget

This creates a group of radio buttons stacked vertically. Exactly one element of this group will be 'on' at a time. The first argument is the associated variable, which should be initialised to an integer in the range of the number of alternatives. The second argument is a list of strings which label these alternatives. As usual, the variable is updated as the user changes the selection. The option `ChangeFunction` can be used to supply a function to obtain immediate feedback:

```
Needs["SuperWidgetPackage`"]

SetAttributes[myFunc, HoldFirst];
myFunc[x_] := Print[Unevaluated[x], "=", x];
x = 2;
SuperWidgetRadioButtonGroup[x,
  {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"},
  ChangeFunction -> myFunc] // SuperGUIRunModal; x

   x=4

4
```

If a list of just one string is supplied, this super widget operates slightly differently. In this case the variable takes on the values 0 and 1, depending on whether the corresponding button it on or off. However, note that by convention, it is normal to use a checkbox to represent this situation, since there are no mutually exclusive alternatives.

SuperWidgetProgressBar

`SuperWidgetProgressBar[ v,max,min,str,opts]`	Displays a progress bar with values between min and max – current value in v. The str argument is a string, whose use is described in the text.

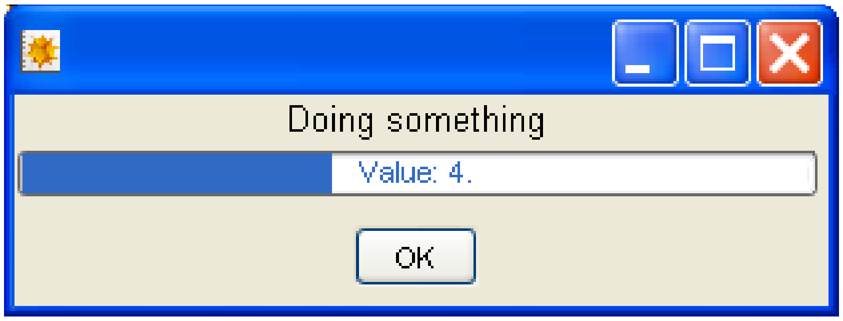
The progress bar super widget

`Tool⌣Tip`	Specifies a toop tip string for the progress bar.
Pixel⌣Width	Specifies the width in pixels of the bar (default 150). The height of the bar is not adjustable.

Options for the progress bar super widget

This creates a progress bar that can be updated by changing the value of the control variable and calling `UpdateWidget⌣Value`. The string argument will be used is a call to StringForm[str,v] to generate a string to appear in the bar. Use an empty string to create an empty bar. Here is a simple example:

```
v = 0.;
{
   {¶0, "Doing something", ¶0},
   {¶0, SuperWidgetProgressBar[v, 0., 10.,
     "Value: `1`", Tool⌣Tip -> "Gibber", Pixel⌣Width → 300], ¶0},
   ¶10,
   {¶0, SuperWidgetButton[Null, "OK", 1], ¶0},
   ¶10
  } // SuperGUIRun;
While[v ≤ 10 && Active⌣WidgetQ[v],
  v = v + 1;
  UpdateWidgetValue[v];
  Pause[1];
 ];
Close⌣Frame[v];
```



In this example, the 'task' whose progress is being monitored is simply simulated using `Pause`. When the progress bar is complete, the entire window is removed using `Close⌣Frame`. The argument to `Close⌣Frame` could have been any variable controlling a widget in the window, `v` just happens to be convenient. In this case, the window containing the progress bar can be closed before the task is complete, so the code checks that `v` is still controlling an active widget and exits early if necessary.

More about associated variables

As you know, every super widget has an associated variable. This association has many uses. It is used to hold the data corresponding to a super widget (where appropriate), and is passed to the `ChangeFunction` as an argument. If you specify that function with `HoldFirst` attribute, you can identify which super widget was responsible (so that you can use the same change function for several super widgets).

Super widget definitions can become quite cluttered, so options can be associated ahead of time with the corresponding associated variable using the `Set⌣Variable⌣Options` function. This can also be convenient because, for example, the setting for the `ChangeFunction` option is really associated with the program logic, rather than the GUI itself. If an option is specified ahead of time and in the actual super widget definition, the latter is used in preference. Here is a trivial example:

```
Needs["SuperWidgetPackage`"]

p1 = 42.5;
Set⌣Variable⌣Options[p1, ChangeFunction → fp1];
SetAttributes[fp1, HoldFirst];
fp1[x_] := Print[Unevaluated[x], "=", x];
SuperWidgetFrame[fr1, {SuperWidgetRealBox[p1]}] // SuperGUIRunModal

   p1=42.54
```

Options are actually checked at the time the corresponding super widget definition is processed. It is also possible to change the default options for one particular type of super widget using the standard *Mathematica* `SetOptions` function:

```
SetOptions[SuperWidgetLabel, Font⌣Color → RGBColor[1, 0, 0]];
{{¶10, SuperWidgetLabel[ll1, "Some red text"], ¶10}} // SuperGUIRunModal
```

Return to default for the sake of the rest of the examples.

```
SetOptions[SuperWidgetLabel, Font⌣Color → RGBColor[0, 0, 0]];
```

Associated variables also carry the enabled/disabled state of a super widget. By default, all super widgets start in their enabled state. However, by calling `Set⌣Enabled⌣Status`, you can switch between the enabled and disabled states. By calling this function on an associated variable that has yet to be used, it is possible to set the initial state of a super widget. Ideally, your program will ensure that no part of your GUI interface is enabled if the corresponding action would be inappropriate.

Interactive graphics

Introduction

The `SuperWidgetGraphicsPanel` lets you create genuinely interactive *Mathematica* graphics applications.

Traditional *Mathematica* graphics is essentially non-interactive and offers only rather crude animation possibilities. This reflects the fact that early versions of *Mathematica* were created at a time when CPU performance was such that a single plot could take many seconds to render. The `SuperWidgetGraphicsPanel` displays conventional 2-D or 3-D *Mathematica* graphics objects, such as those produced by `Plot,Plot3D`, etc. and offers a variety of extensions to make them interactive and animated. In particular, 2-D graphics objects can be augmented with additional graphics primitives – `Graph⋱ics⌄Region` structures – which control the interaction with the mouse. In the simplest case, a program will completely replace the graphic (including any `Graphics⌄Region` structures) each time it needs to update the display. On a modern (say 2GHz) machine, this generally requires about a second – depending on the size of window used. While this is fast enough for some purposes, it is also possible to add one or more arbitrary shaped 'icons' (sometimes referred to as 'sprites') that float on top of the main graphics and can be moved about for very little cost. For example, the corresponding points on two curves might be represented by a pair of moving icons.

`SuperWidgetGraphicsPanel[v,opts]`	Displays graphics panel, which initially contains the plot stored in the control variable v.
`SuperWidgetGraphicsPanel[v,plot,opts]`	Obsolete form, retained for compatibility. Displays graphics panel, which initially contains plot, the given graphics object.

Interactive graphics super widget

`MouseMotionFunction`	Supplies a function of three arguments var,x,y which is called each time the mouse is moved. This should be used sparingly, as it is easy to impede the motion of the mouse by excessive computation.
`ResizeFunction`	Supplies a function of three arguments var,sx,sy which is called each time graphics area is re-sized. The arguments sx and sy represent the new data for ImageSize – you can use them to deduce how much space you have to draw into. The function must refresh the graphics with ReCalibrate->True.
`Dynamic⌄Tool⌄Tip`	Specifies that the graphics panel will support a tool tip whose contents are computed as a function of the mouse position whenever they are needed. This function of three arguments var,x,y is called when needed, and should return a string result – see dynamic tool tips below.

Options for SuperWidgetGraphicsPanel excluding some obsolete options

Because interactive examples are inevitably a little more complicated, let us start with a display of a graph without mouse interaction. Since we are not interested in using the mouse, the plot might as well be 3-dimensional:

```
Needs["SuperWidgetPackage`"]

vv = Plot3D[Sin[x y], {x, 0, 4}, {y, 0, 4},
   DisplayFunction → Identity, ImageSize → {400, 400}];
SuperWidgetGraphicsPanel[vv] // SuperGUIRunModal
```

The variable vv is set up to contain the plot and also acts as the control variable. This is, of course, precisely analogous to other super widgets such as `SuperWidgetIntegerBox`. The size of the resulting graphics panel (in pixels) is determined by the ImageSize option used in the command (`Plot3D` in this case) used to create the graphic. Note the use of `DisplayFunction→Identity` since we (presumably) don't want the plot to appear in the notebook as well as in the GUI.

At its very simplest, some interaction can, of course, be achieved by changing the graphic in response to other widgets in the window. For example, in this example the 3-D graph changes in response to the menu:

```
f⌣sin[] := Module[{},
   vv = Plot3D[Sin[x y], {x, 0, 4},
     {y, 0, 4}, DisplayFunction → Identity, ImageSize → {400, 400}];
   UpdateWidgetValue[vv];
  ];
f⌣cos[] := Module[{},
   vv = Plot3D[Cos[x y], {x, 0, 4},
     {y, 0, 4}, DisplayFunction → Identity, ImageSize → {400, 400}];
   UpdateWidgetValue[vv]; (* This has no effect if
    executed before GUI is launched *)
  ];
f⌣sin[];

SuperWidgetFrame[Null, {
   SuperWidgetGraphicsPanel[vv]},
  Menu → {{"Function", {{"Sin", f⌣sin}, {"Cos", f⌣cos}}}},
  Title -> "3-D function plotter"] // SuperGUIRunModal
```

Take care that each plot uses the same settings for `ImageSize` unless you are refreshing the plot within a `ResizeFunction`. In the latter case, you should set the new `ImageSize` to equal the size parameters passed to that function. If you set the `ImageSize` to any other value, the result will be either clipped or not fill the window.

Note that `SuperWidgetGraphicsPanel` used to take the plot as its second argument, rather than as the value of the control variable. This was changed for consistency, but the obsolete form has been retained for compatibility and should not be used in new code.

Graphics regions

Because a mouse is a 2-D input device, 2-D graphics can enjoy a very high level of interactivity. This functionality is organised around the concept of the `Graphics⌣Region`.

`Graphics⌣Region[v, {{x1,y1},{x2,y2},...},opts]`	Functions as a graphics primitive in 2-D graphics prepared for GUI display. The variable v is the control variable for the region. The second argument is a list of the vertices of a polygon defining the shape of the region. These coordinates should not be defined using Scaled or Offset constructs. Graphics⌣Region constructs are inert except when interpreted as a graphic for a SuperWidgetGraphicsPanel.

Graphics⌣Region construct

Action⌄Function	Supplies a function of three or five arguments which is called each time the mouse performs an action as specified by the Mouse⌄Mode. The first argument will be the control variable of the region, followed by x,y or x1,y1,x2,y2 depending on the drag mode.
Mouse⌄Mode	Specifies mouse behaviour in the region. Values include: CliClick⌄Action – Take action on mouse clicks (default). Line⌄Drag⌄Action – Show dragging operations with an XOR'ed line and take action when the mouse key is finally released. Rectangle⌄Drag⌄Action – Show dragging operations with an XOR'ed rectangle and take action when the mouse key is finally released. Region⌄Drag⌄Action – Dragging operations drag a copy of the entire region across the graphics area and take action when the mouse key is finally released. (Imagine, say, dragging electronic circuit symbols or musical notes across the graphics surface).
Leave⌄Dragged⌄Image	Used in conjunction with Mouse⌄Mode->All to achieve a smoother visual effect in certain situations – see below.
Menu	Supplies a menu that appears as a popup menu when the right mouse button is clicked (Windows) or an equivalent operation is performed in another environment. The menu structure uses the same scheme as used in the SuperWidgetFrame. Each menu function should accept one argument – the control variable of the region.
Mouse⌄Cursor	Specifies the name of a cursor to be used when the mouse is in the given region. Cursor names include "DEFAULT_CURSOR", "CROSSHAIR_CURSOR", "WAIT_CURSOR", "TEXT_CURSOR", "HAND_CURSOR".
Tool⌄Tip	Specifies a string to be used as a tool tip – displayed as the mouse cursor enters the region.

Options for Graphics⌄Region construct

A Graphics⌄Region can be any polygonal shape and can be of any size up to the size of the entire graphic. Graphic⌄s⌄Region constructs can be geometrically nested, but should not merely overlap, as this would make it impossible to unambiguously assign a region to every point. Overlaps are not faulted, however, as in some situations it may be hard to avoid a slight overlap of regions, and the resultant ambiguity does not really matter. Each region can have its own cursor and context menu, and responds to the mouse in a way that is determined by the Mouse⌄Mode. Line and Rectangle dragging operations must begin and end in the same region, but for Mouse⌄Mode->Region⌄Drag⌄Action a copy of the entire region will be dragged across the window.

The control variable of a Graphics⌄Region is closely analogous to the control variable of a super widget. In particular, it is possible to set up options for a Graphics⌄Region ahead of time using Set⌄Variable⌄Options.

With Mouse⌄Mode->Click⌄Action mouse clicks are simply reported to the Action⌄Function for the region in which they occur, however the other Mouse⌄Mode options result in temporary changes to the window contents. Normally these are removed just before the Action⌄Function is called to respond to the situation. In many cases, the Action⌄Function will replace the image in a way that corresponds to the temporary changes. In these situations, setting Leave⌄Dragged⌄Image->True will reduce the flicker by leaving the line, rectangle, or graphic at the end of the drag operation ready for the Action⌄Function to replace the graphic. Note that it is vital that if this option is used the graphic is indeed updated (using UpdateWidgetValue) otherwise inconsistent results will be observed.

If you do not require any explicit mouse interaction in a region, simply leave the Mouse⌣Mode at its default of Click⌣Action, and do not supply an Action⌣Function.

All of this is best illustrated with an example. We start with a list (fx) of {x,y} data points and define a function build⌣graph that sets up the global variable graph with a modified ListPlot. Every point in the ListPlot is surrounded by a tiny Graphics⌣Region with its own cursor and menu. This means that as you move the mouse cursor over the graph it will change into a hand cursor near to each point. When this happens, a right click (or equivalent in non-Windows environments) will display a 1-element context menu. Clicking on this will delete the point in question.

```
Needs["SuperWidgetPackage`"]

fx = Table[{N[x], x + Random[] - 0.5}, {x, 0, 10}];

build⌣graph[] := Module[{},
graph = ListPlot[fx, DisplayFunction → Identity,
         ImageSize → {600, 600}, PlotRange → {{0, 10}, {0, 10}}]
        /. Point[pp : {ll_List, ___}] :> Sequence @@ Map[Point, pp]
      /. Point[{x_, y_}] :> {Point[{x, y}], Graphics⌣Region[
          Evaluate[Unique[]], Table[{x + 0.2 Cos[t], y + 0.2 Sin[t]}, {t, 0, 2 π, π / 8}],
          Mouse⌣Cursor -> "HAND_CURSOR", Menu → {{"Delete point", dp}}]};
UpdateWidgetValue[graph];
];

dp[v_] := Module[{s},
s = Cases[graph, {_Point, Graphics⌣Region[v, ___]}, ∞][[1]];
    fx = DeleteCases[fx, {s[[1, 1, 1]], _}];
    build⌣graph[];
];

build⌣graph[];
SuperWidgetGraphicsPanel[graph] // SuperGUIRunModal
```

This is a simple example of genuinely interactive graphics. Note in particular that the PlotRange option has been used on ListPlot. This is important because it is vital that the plot axes are the same from plot to plot (even if you delete an end-point) otherwise the plot will jump about as the user manipulates it. If, for any reason, the PlotRange cannot be fixed, it is vital to add the option ReCalibrate->True to the call to UpdateWidgetValue. This is expensive, which is why it is not set by default, but it is vital that the SWP is told if the graphic scale changes in some way.

Since the control variables for the Graphics⌣Region constructs in the above example are never given a value, it was not necessary to give the function dp the HoldFirst attribute.

In the following, slightly more realistic example, points may be added with a left mouse button click or deleted (as above) using the context menu.

```
Clear[myPlot];

addingPoints = True;
myData = {{Unique[], 1, 1}, {Unique[], 4, 4}};

dp[v_] := Module[{s},
myData = DeleteCases[myData, {v, _, _}];
pp1 = myPlot[];
    UpdateWidgetValue[pp1];
];

myPlot[] := Module[{m, c, x, fitting = False, tmp, p1, p2},
    If[Length[myData] > 1,
     fitting = True;
     tmp = FindFit[Map[Drop[#, 1] &, myData], c + m x, {c, m}, x];
     p1 = {0, c} /. tmp;
     p2 = {10, 10 m + c} /. tmp;
    ];
    Graphics[{
      Map[{Point[Drop[#, 1]], Graphics‿Region[Evaluate[#[[1]]],
           Table[{#[[2]] + 0.2 Cos[t], #[[3]] + 0.2 Sin[t]}, {t, 0, 2 π, π / 8}],
           Mouse‿Cursor -> "HAND_CURSOR", Menu → {{"Delete point", dp}}]} &, myData],

      If[fitting, {RGBColor[1, 0, 0], Line[{p1, p2}]}, {}],
      Text["Use the mouse to add points,\nright click on a point to remove it.",
        {5, 16}, {0, 0}, TextStyle → {FontSize → 17}]
     },
     DisplayFunction → Identity, ImageSize → {600, 400},
     AspectRatio -> 0.6, Axes -> True, PlotRange → {{0, 10}, {-5, 18}}]
   ];

myExit[] := Close‿Frame[fr1];
mouseClick[_, x_, y_] := Module[{pt},
    Print["Adding data point: ", {x, y}];
    myData = Append[myData, {Unique[], x, y}];
    pp1 = myPlot[];
    UpdateWidgetValue[pp1];
   ];
myMenu = {{"File", {{"Exit", myExit}}}};

pp1 = myPlot[]; SuperWidgetFrame[fr1,
   {SuperWidgetGraphicsPanel[pp1, MouseClickFunction -> mouseClick]},
   Title -> "Simple interactive graph", Menu → myMenu] // SuperGUIRunModal;
```

```
    Adding data point: {8.9157, 9.81602}

    Adding data point: {6.78779, 8.40922}

    Adding data point: {6.64826, 5.52864}

    Adding data point: {4.39826, 6.60049}
```

This program displays a simple straight-line graph with just two points. As you add or delete points using the mouse, the best-fit straight line is re-drawn on the fly. You can achieve all this using the relatively tiny amount of code above – nothing is hidden!

Bear in mind that the above example is still extremely simple because the code has been kept easy to understand. You should consult the examples at my website for more extensive examples: `http://www.dbaileyconsultancy.co.uk/swp_examples/swp_examples.html`.

The need for calibration

Ordinary *Mathematica* graphics is quite unlike most computer graphics in that you can create a plot or a low-level graphical object without any thought about the scale necessary to place it on the screen or paper. The system chooses a scale for you just before the image is created. You can create a plot of part of the Milky Way (or part of a molecule) using units of meters if you wish – the system will simply take care of the necessary scaling. The `SuperWidgetGraphicsPanel` determines what this scale is by calculating (this process is not visible to the user, unlike in earlier versions) a slightly modified version of your plot with certain colours adjusted and two small spots added. The resultant image is then read back into *Mathematica* as an array of pixels, and used to determine the relationship between mouse position and graphics coordinates. Because calibration of a large image can be quite time consuming, by default it is only done once, and the SWP 'assumes' that subsequent images will require the same calibration. It is important that your graphics satisfy this assumption, otherwise the mouse coordinates returned by the system will be wrong. For example, if you initially display `Plot[Sin[x],{x,0,10}]` and subsequently use `UpdateWidgetValue` to display `Plot[x^2,{x,0,10}]`, it is vital to use the option `ReCalibrate->True` on the call to `UpdateWidgetValue`. If necessary it may be useful to bound your graphics with a box (and not draw outside the box!) to ensure successive images use the same scale.

Re-sizing a graphic

When a graphics panel is re-sized, the system will by default re-scale and re-calibrate the image automatically. However, sometimes this is not what is required – for example it may be necessary to display more of some diagram as the image expands, rather than merely scale up the existing diagram. If the ResizeFunction option is used, the replacement of the image is left to you. You must generate a new plot with the new ImageResolution parameters (arguments 2 and 3) and update using `ReCalibrate->True`. If you fail to update and re-calibrate the plot inside your `ResizeFunction` the image will not fill the space and the mouse coordinates will not be properly calibrated.

Mouse modes Line⌣Drag⌣Action, and Rectangle⌣Drag⌣Action

Sometimes, as above, we want to interact with a graphic by means of clicks – single points – but sometimes it is more meaningful to think in terms of lines or rectangles. Think of a simple line drawing program. When the mouse is pressed and dragged we want a line or box to appear on the drawing surface and track the motion of the mouse. Software packages normally do this by drawing the extra lines using XOR mode (which makes it fast to remove the line and re-draw it as the mouse moves). Since none of this corresponds to normal *Mathematica* graphics operations, which would, in any case, be too slow to track mouse movements, this XOR line drawing is performed by the `SuperWidgetGraphicsPanel` itself. These options can be selected using the `Mouse⌣Mode` option to the `Graphics⌣Region`, and can be changed on the fly using the `Set⌣Mouse⌣Mode` function.

The following example is a little longer than most in this user guide, but it does illustrate something of the power that is possible using this super widget:

```
Needs["SuperWidgetPackage`"]
```

```
drawing = {};
drawing⌣history = {};
style = 1;
Set⌣Line⌣Style[] := (Set⌣Mouse⌣Mode[panel1, gr0, Line⌣Drag⌣Action]; style = 1);
Set⌣Box⌣Style[] := (Set⌣Mouse⌣Mode[panel1, gr0, Rectangle⌣Drag⌣Action]; style = 2);
myExit[] := Close⌣Frame[fr1];

do⌣drawing[] := Module[{},
   Graphics[{drawing, Graphics⌣Region[gr0, {{0, 0}, {0, 10}, {10, 10}, {10, 0}},
      Mouse⌣Mode → Line⌣Drag⌣Action, Action⌣Function → updateDrawing]},
    PlotRange → {{0, 10}, {0, 10}}, AspectRatio → 1, ImageSize → {600, 600}]
  ];

SetAttributes[newLine, HoldFirst];
updateDrawing[pvar_, x1_, y1_, x2_, y2_] := Module[{},
   drawing⌣history = Append[drawing⌣history, drawing];
   Switch[style,
    1,
    drawing = Append[drawing, Line[{{x1, y1}, {x2, y2}}]],

    2,
    drawing = Append[drawing, Line[{{x1, y1}, {x1, y2}, {x2, y2}, {x2, y1}, {x1, y1}}]]
   ];
   panel1 = do⌣drawing[];
   UpdateWidgetValue[panel1];
  ];

undo⌣update[] :=
  If[drawing⌣history =!= {},
   drawing = Last[drawing⌣history];
   drawing⌣history = Drop[drawing⌣history, -1];
   panel1 = do⌣drawing[];
   UpdateWidgetValue[panel1];
  ];

SuperGUIRunModal[
 SuperWidgetFrame[fr1, {
   panel1 = do⌣drawing[];
   SuperWidgetGraphicsPanel[panel1],
   WidgetSpace[10],
   {WidgetFill[], SuperWidgetButton[bb1, "OK", 1], WidgetFill[]},
   WidgetSpace[10]
  }, Title -> "Example drawing program",
  Menu → {
    {"File", {{"Exit", myExit}}},
    {"Edit", {{"Undo", undo⌣update}}},
    {"Objects", {{"Line", Set⌣Line⌣Style}, {"Box", Set⌣Box⌣Style}}}
   }
 ](* ,ConcealNotebooks→True *)
]
```

```
1
```

When you run this program you are presented with a big white drawing surface. As you drag the mouse (i.e. move it with the left button depressed) a line will appear. This line will disappear when you release the mouse button, but the `updateDrawing` function is then called and refreshes the graphic with the extra line. By using the menu option it is possible to change to rectangle drawing by changing the mouse mode of the graphics region and recording the change of object in the style variable.

It is important to realise that it is your *Mathematica* code which ultimately draws the permanent line, rectangle, or whatever. Thus, for example, if this were an electronic circuit wiring application, it might be desirable to spot line endpoints that were close to components and move the line slightly to close the gap. You can do what you like with the coordinate information that you get from the mouse – you don't need to just draw a line between the end-points.

The above code also includes an (infinite!) undo menu option. This is achieved by simply storing a list of drawings created so far and backing off to the previous one in response to the menu item. I hope I have illustrated that the `SuperWidgetGraphicsPanel` opens up a whole new realm of *Mathematica* graphics!

Finally, if you put back the option `ConcealNotebooks→True` into the code, the notebooks will be hidden while the GUI is displayed. This is recommended for many finished applications because prevents the user inadvertently clicking on the notebook and obscuring the window to which he should be attending. This unfortunate problem occurs because the Java GUI application is executing as a separate process from *Mathematica* – linked together by *MathLink*.

Mouse mode Region‿Drag‿Action

When you start a drag operation in this mode, the contents of the whole region are dragged as a picture across the graphic. One of the examples at my website exploits this in a simple electronic circuit drawing program. The circuit symbols (such as transistors) are each drawn in a `Graphics‿Region`, and can be dragged about the screen.

In the following code a red arrow is placed on a graph, and a `Graphics‿Region` is placed round the arrow and set up so it can be dragged across the screen. The `test` function is called at the end of the drag and re-positions the arrow, ignoring the y value. It also computes the root of the equation `f[x]==0` using `FindRoot`, and places a (passive) blue arrow at this point. This process can be repeated as desired. Notice that if you position the red arrow well away from a root, it does not always find the nearest root.

```
Needs["SuperWidgetPackage`"]
```

```
Clear[f];
results = {};
x⌣pos = 0;
f[x_] := 8 Sin[x];
test[_, x1_, y1_, x2_, y2_] := Module[{ans},
   x⌣pos = x2;
   ans = x /. FindRoot[f[x] == 0, {x, x⌣pos}];
   If[ans > -10 && ans < 10,
    results = Append[results, blueArrow[x /. FindRoot[f[x] == 0, {x, x⌣pos}], 0, 0.5]]];
   p = myPlot[];
   UpdateWidgetValue[p];
  ];
redArrow[x_, y_, scale_] :=
  {Red, Polygon[{{x, y}, {x - scale / 2, y + scale}, {x + scale / 2, y + scale}}],
   Polygon[{{x - scale / 5, y + scale}, {x - scale / 5, y + 3 scale},
     {x + scale / 5, y + 3 scale}, {x + scale / 5, y + scale}}],
   Graphics⌣Region[gr0, {{x, y}, {x - scale / 2, y + scale}, {x - scale / 5, y + scale},
     {x - scale / 5, y + 3 scale}, {x + scale / 5, y + 3 scale}, {x + scale / 5, y + scale},
     {x + scale / 2, y + scale}}, Mouse⌣Mode -> Region⌣Drag⌣Action,
    Action⌣Function → test, Mouse⌣Cursor -> "HAND_CURSOR"]};

blueArrow[x_, y_, scale_] :=
 {Blue, Polygon[{{x, y}, {x - scale / 2, y + scale}, {x + scale / 2, y + scale}}],
  Polygon[{{x - scale / 5, y + scale}, {x - scale / 5, y + 3 scale},
    {x + scale / 5, y + 3 scale}, {x + scale / 5, y + scale}}]}

myPlot[] := Plot[f[x], {x, -10, 10}, PlotRange → {{-10, 10}, {-10, 10}},
   ImageSize → {600, 600}, AspectRatio → Automatic,
   DisplayFunction → Identity, Epilog → {redArrow[x⌣pos, 8, .5], results}];
p = myPlot[];
SuperWidgetGraphicsPanel[p] // SuperGUIRunModal
```

Animation using icons

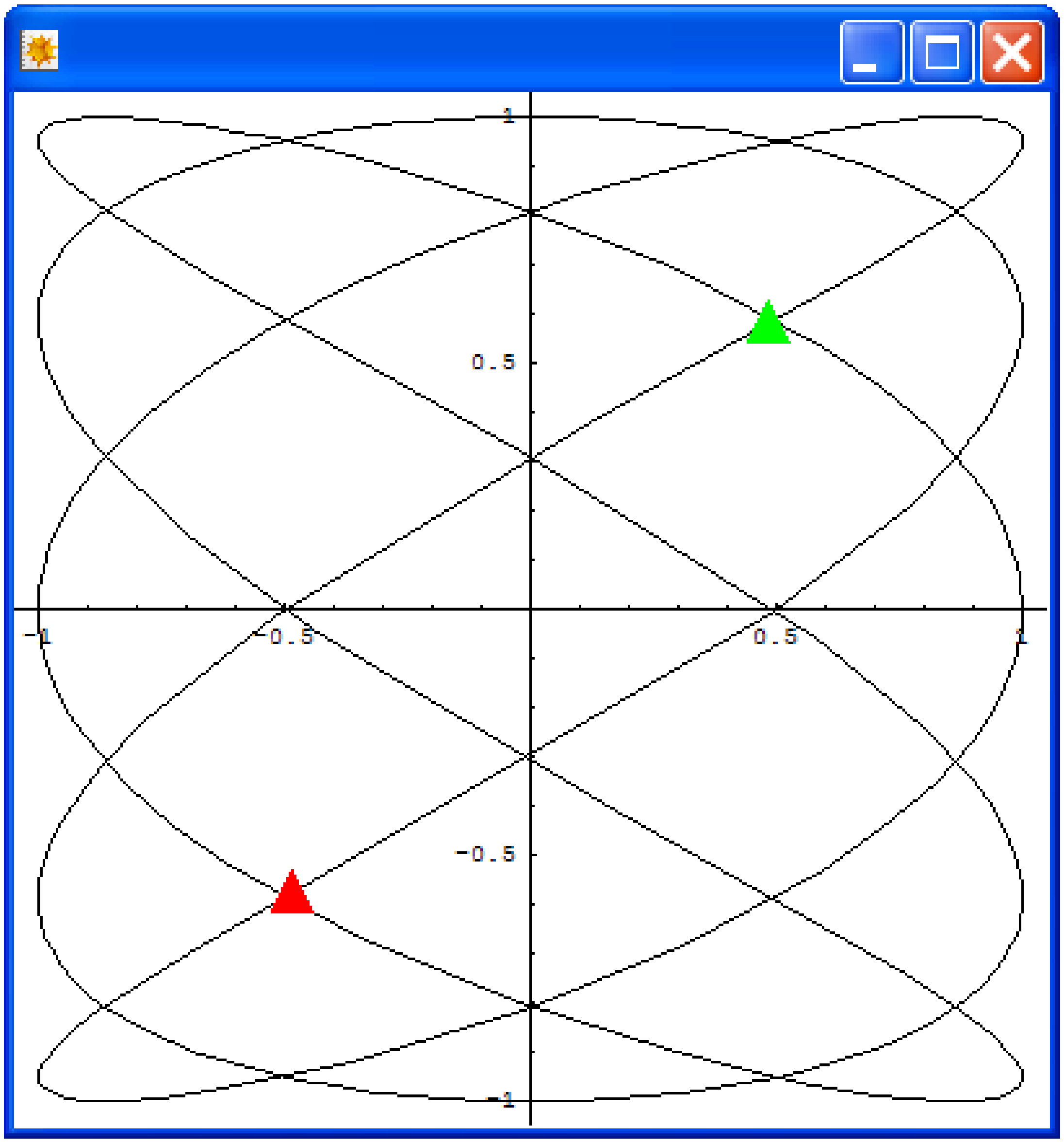
An 'icon' is a small 2-D graphics that is not, in general rectangular. These are created using Graphics objects, and one colour is designated as representing the 'colour' transparent. Sprites may be added, moved and removed from a scene for very little cost, and so offer an effective way to implement certain types of animation. Icons can only be attached to 2-D graphics.

Add⌣Icon[var,var1,x,y,opts]	Adds a icon to the SuperWidgetGraphicsPanel associated with variable var. The variable var1 holds the Graphics object, and is used to refer to the icon in subsequent operations. The icon is initially positioned at the point (x,y) in *Mathematica* graphics coordinates. See below for details of the options.
Move⌣Icon[var,var1,x,y]	Moves the icon labelled by var1 in the SuperWidgetGraphicsPanel associated with variable var to the specified point.
Delete⌣Icon[var,var1]	Deletes the icon labelled by variable var1 from the SuperWidget-GraphicsPanel associated with variable var.
Delete⌣All⌣Icons[var]	Removes all icons from the SuperWidgetGraphicsPanel associated with variable var.
Service⌣GUI[]	This should be called reasonably frequently within an animation to ensure that other GUI controls are processed in a timely fashion. This function can be useful in any situation in which a lengthy calculation might prevent a speedy GUI response.

Graphics animation using icons

Here is a very simple – but instructive – example of an animation using icons:

```
my⌣icon1 = Graphics[{Green, Polygon[{{0, 0}, {1, 0}, {0.5, 1}}]},
    AspectRatio → Automatic, ImageSize → 20];
my⌣icon2 = Graphics[{Red, Polygon[{{0, 0}, {1, 0}, {0.5, 1}}]},
    AspectRatio → Automatic, ImageSize → 20];
vvv = ParametricPlot[{Cos[5 t], Sin[3 t]}, {t, 0, 2 π},
    AspectRatio → Automatic, DisplayFunction → Identity];
SuperWidgetGraphicsPanel[vvv] // SuperGUIRun;
Add⌣Icon[vvv, my⌣icon1, 0, 0];
Add⌣Icon[vvv, my⌣icon2, 0, 0];
time = 0;
While[Active⌣WidgetQ[vvv],
  Move⌣Icon[vvv, my⌣icon1, Cos[5 time], Sin[3 time]];
  Move⌣Icon[vvv, my⌣icon2, Cos[5 (time + π)], Sin[3 (time + π)]];
  Service⌣GUI[];
  Pause[0.1];
  time = Mod[time + 0.05, 2 π];
 ];
```



One instant from the above animation

- Observe that the speed of this animation is controlled by the length of the pause and the size of the time step. Removing the pause demonstrates that the maximum update rate is very high. Animations should, wherever possible, be controlled by a pause, rather than than simply running flat out, because this makes the display independant of subsequent increases in processor performance.

- Note that the `While` loop executes until the variable `vvv` is no-longer associated with a widget – i.e. until the window is closed. The calls to the function `Service˽GUI` are desirable whenever a lengthy calculation (not just an animation) is performed while a super widget is on display – they ensure that actions associated with other widgets are processed in a timely manner.

- The size of the icons is controlled by the `ImageSize` option to the `Graphics` structure – omitting this will result in absurdly large icons. The automatic aspect ratio ensures that the shape of the icons are not squashed.

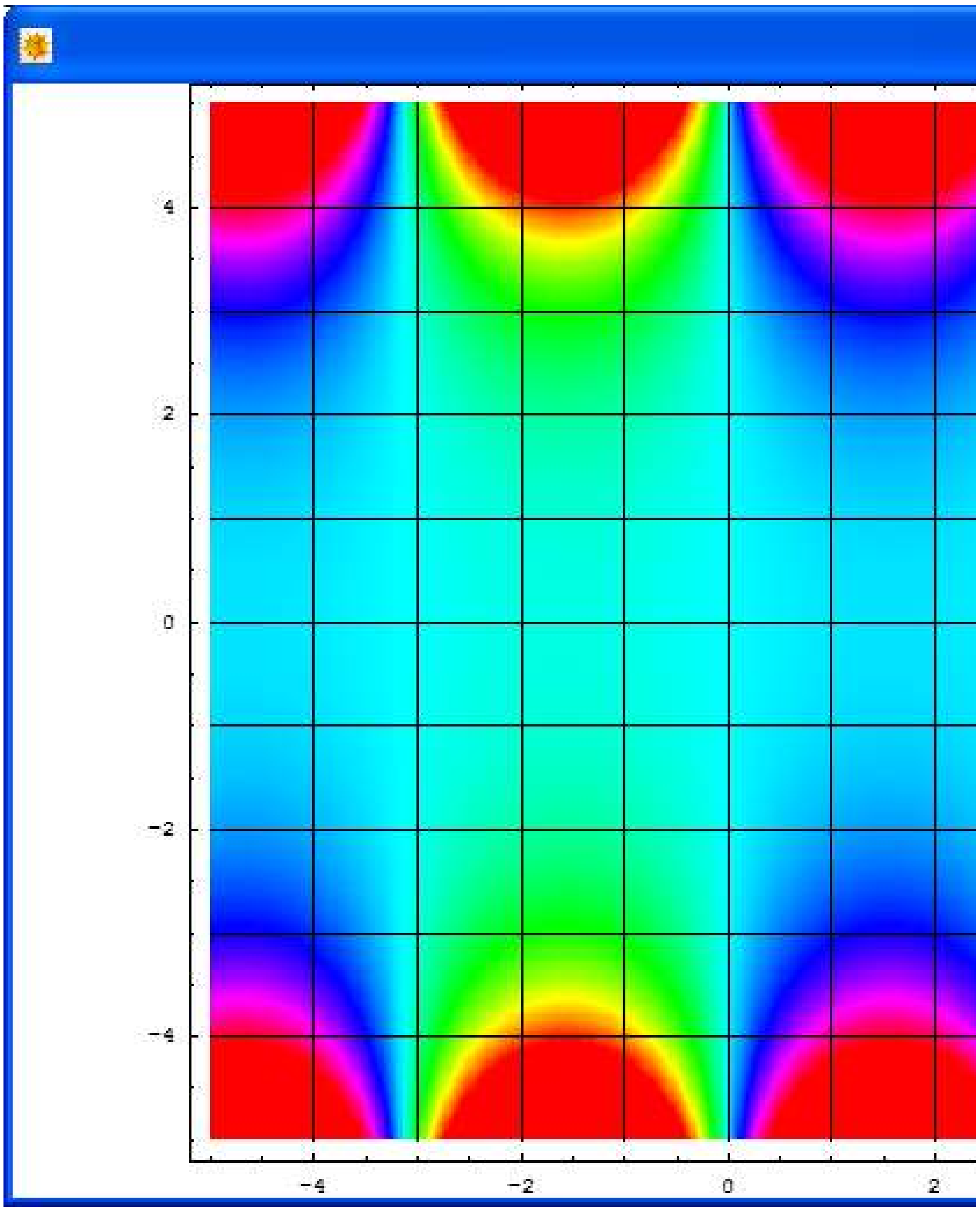
- Although the above example does not require this, it would be possible to update the main graphic within the animation loop by calling `UpdateWidgetValue` on the variable `vvv`. This would be a much slower operation than moving the icons, and the animation would need to be designed with this in mind.

By default, icons created by Add⌣Icon render white parts of the icon as transparent. Since this is the default background colour, this is often a good choice, however the option `Transparent⌣Color` can be used to select another colour to play this role. You need this option if part of the icon is to display as white. Also by default, icons are centred on a point in the middle of their rectangle. This point is known as the 'hot spot' because it is the point to which the icon refers. Sometimes it is more convenient to select a different hot spot – e.g. in the case of an icon shaped like an arrow – using the `Hot⌣Spot` option. This defaults to `{0.5,0.5}` and represents the position of the hot spot as a fraction of the total icon dimensions. Thus `{0,0}` would represent bottom left, `{1,1}`, top right, etc.

Dynamic tool tips

A tool-tip is normally used to provide fixed some help information to the user. Many of the super widgets take a `Tool⌣Tip` option for exactly this purpose. However, within ia graphics panel, it is usually more convenient to use a tool-tip that is a function of the (x,y) position. This can be particularly useful for displaying information about a complex function. The idea is that each time the mouse moves to a new location on the graphics surface, a function of three arguments is called to recompute the tool-tip string. For example:

```
ToExpression["{plot}"];
plot = DensityPlot[Re[Sin[x + I y]], {x, -5, 5}, {y, -5, 5}, ColorFunction → Hue,
    DisplayFunction → Identity, ImageSize → {600, 480}, PlotPoints → 40,
    Mesh → False, Epilog → {Table[Line[{{-5, k}, {5, k}}], {k, -4, 4}],
      Table[Line[{{k, -5}, {k, 5}}], {k, -4, 4}]}];
HoldFirst[f];
f[_, x_, y_] := Module[{},
    "    f[" <> ToString[SetAccuracy[x + I y, 4]] <>
     "]=" <> ToString[SetAccuracy[Sin[x + I y], 4]]
   ];
SuperWidgetGraphicsPanel[plot, Dynamic⌣Tool⌣Tip → f] // SuperGUIRunModal
```



```
- Graphics -
```

Although the above image does not actually show the tool-tip (because I had to us the mouse to take the picture!), an animated GIF that shows the tool-tips can be found on the SWP website here.

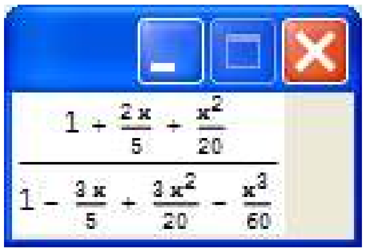
If the mouse cursor enters a graphics region with its own tool-tip, this takes precedence over the dynamic tool-tip.

Notice that the mesh is not useful if, as here, enough points are selected to make a smooth image, therefore the above code disables it and draws a grid instead.

Displaying other objects using SuperWidgetGraphicsPanel

Although SuperWidgetGraphicsPanel is normally used to display graphics, it can be used to display any object. For example:

```
xxx = PadeApproximant[Exp[x], {x, 0, {2, 3}}];
SuperWidgetGraphicsPanel[xxx] // SuperGUIRunModal
```



$$\frac{1+\frac{2x}{5}+\frac{x^2}{20}}{1-\frac{3x}{5}+\frac{3x^2}{20}-\frac{x^3}{60}}$$

Graphics regions can be introduced by using the Annotation mechanism introduced at 6.0. For example:

```
xxx = a + Annotation[b, Graphics⌣Region[ggg, Null, Tool⌣Tip -> "This is the variable b"]]
```

This object would display as a+b, and a tool tip would appear when 'b' was moused over. All the features of graphics regions are available using this mechanism.

Tip! This is just one of the many interesting possibilities opened up by the introduction of `Annotation` in *Mathematica* 6.0 because it allows data to be associated with an expression invisibly.

LiveGraphics3D

Introduction to LiveGraphics3D

LiveGraphics3D is a Java applet written by Martin Kraus which has been used extensively by *Mathematica* programmers interested in displaying and manipulating 3-dimentional objects. An 'applet' is a piece of Java code that is designed to run inside a web page. It is called from HTML with various parameters that control it. This was the original idea of LiveGraphics3D, which has been used to create many interactive mathematical displays, for example here. These displays can be rotated by the mouse. In the later versions it is also possible to add points that can be dragged using the mouse and which cause the rest of the object to reconfigure accordingly. The LiveGraphics3D applet can be downloaded here together with its documentation, and you should ensure you have version 1.83 or later.

A good place to put the file Live.jar to ensure it is found by the system is inside the Java subdirectory of the SuperWidgetPackage – which will be found at `$InstallationDirectory<>"/addons/Applications/superwidgetpackage/Java"`. This is where the SWP will put it if you rely on SWP to download the applet from the internet the first time you use it.

More recently, LiveGraphics3D has acquired a J/Link interface so that a *Mathematica* program can create a window containing LiveGraphics3D and control it directly – without using a browser. Using the SWP, it is possible to treat a LiveGraphics3D object as a widget and control it with other widgets on the same window.

This user guide does not attempt to describe how to use the LiveGraphics3D applet, only how to interface it with the SWP.

SuperWidgetLiveGraphics3D

`SuperWidgetLiveGraphics3D[v,opts]`	Displays Graphics3D object stored in v using LiveGraphics3D

LiveGraphics3D super widget

`Background`	Background colour for the applet – defaults to white.
`ChangeFunction`	One-argument function to be called when a user mouse action might have changed one of the independent parameters or rotated the image.
Full`ChangeFunction`	One-argument function to be called when a user mouse action might have changed one of the independent parameters or rotated the image. The function is called repeatedly during dragging operations – see text for details.
`Pixel⌣Width`	Width in pixels for the applet.
`Pixel⌣Height`	Height in pixelts for the applet.
`Independent⌣Variables`	Specifies a list of independant variable specifications of the form var->real-value. See the LiveGraphics3D documentation for full details.
`Dependent⌣Variables`	Specifies a list of dependant variable specifications of the form var->real-expression. See the LiveGraphics3D documentation for full details.
`Mouse⌣Dragable`	Determines whether the mouse can rotate the image. This defaults to True, but can usefully be set false in cases where the mouse is used to control the variables..

Options for SuperWidgetLiveGraphics3D

First you must prepare some input suitable for the LiveGraphics3D applet. A simple 3-D plot will do, but there are issues regarding overlapping polygons – so read the LiveGraphics3D documentation for full details. This data is then used in the control variable:

```
Needs["SuperWidgetPackage`"]

v = Graphics3D[Plot3D[Sin[x y], {x, -2, 2}, {y, -2, 2}, DisplayFunction → Identity],
   ViewPoint → {1, 1, 1}, ViewVertical → {0, 0, 1}];

{SuperWidgetLiveGraphics3D[v], ¶10, {¶0, SuperWidgetButton[Null, "OK", 1], ¶0}, ¶10} //
 SuperGUIRunModal

1
```

Note that the image can be rotated with the mouse, or even set spinning.

The LiveGraphics3D applet is capable of working interactively. The Graphics3D object can contain variables which can be controlled by dragging the mouse or used to produce animation. It is possible to interact with this process using the `Change⌣Function` or `FullChangeFunction` options. The supplied function will be called (with the associated variable as argument) if a user mouse action might have changed an independent variable or rotated the image. The function that you supply must tolerate the fact that it will sometimes get called when no change has taken place. This is due to the rather loose connection between the SWP and the applet. Passing the associated variable to the `LiveGraphics3DVariables` function will return a complete set of variable values as a set or rules. It is particularly important that these variables are never given actual values within your *Mathematica* code.

Using the `FullChangeFunction` it is possible to get continuous feedback while a drag operation is being performed. While this can be useful, it may overload the processor and cause the dragging operation to become erratic in some situations.

By altering the graphical data stored in the associated variable and calling `UpdateWidgetValue` it is possible to completely change the image on display.

Although the above may sound a little confusing, the concept of parameterized graphics using LiveGraphics3D is well described at Martin Kraus' site – so a simple example of its use in the super widget should suffice:

```
Circle3D[{px_, py_, pz_}, r_] := With[{eps = 0.1},
   Line[Table[{px + r Cos[t], py + r Sin[t], pz}, {t, 0, 2.0 π + eps, eps}]]];

Text3D[x_, pos_] := Text[StyleForm[x, FontSize → 35, FontWeight -> "Bold"], pos * 1.095];
xxx = 1 / Sqrt[2.0];
yyy = 1 / Sqrt[2.0];
gg = Graphics3D[{
     Thickness[0.003],
     Text3D["A", {-1, 0, 0}], Text3D["B", {1, 0, 0}], Text3D["C", {xx, yy, 0}],
     Circle3D[{0, 0, 0}, 1], Line[{{-1, 0, 0}, {1, 0, 0}}],
     PointSize[0.015], Point[{xx, yy, zz}],
     Line[{{-1, 0, 0}, {xx, yy, 0}, {1, 0, 0}}]
   }, ViewPoint → {0, 0, 4}, ViewVertical → {0, 1, 0}, Boxed → False];

indepvar = {xx → xxx, yy → yyy};
depvar = {zz → 0, theta -> ArcTan[xx, yy], xx → Cos[theta], yy → Sin[theta]};
```

```
Clear[f];
f[_] := Module[{s},
    s = LiveGraphics3DVariables[gg];
    {xxx, yyy} = {xx, yy} /. s;
    AC = Norm[{xxx, yyy} - {-1, 0}];
    BC = Norm[{xxx, yyy} - {1, 0}];
    res = Sqrt[AC^2 + BC^2];
    UpdateWidgetValue[AC];
    UpdateWidgetValue[BC];
    UpdateWidgetValue[res];
   ];

SuperWidgetFrame[Null, {
     {¶10,
      {
       {"AC", TAB, "=", SuperWidgetRealBox[AC]},
       {"BC", TAB, "=", SuperWidgetRealBox[BC]},
       {Image⌣Expression[HoldForm[Sqrt[AC^2 + BC^2]]],
         TAB, "=", SuperWidgetRealBox[res]}
      },
      SuperWidgetLiveGraphics3D[gg, Pixel⌣Width → 600, Pixel⌣Height → 450,
       Mouse⌣Dragable → False,
       Background → RGBColor[0.8, 0.8, 1],
       Independent⌣Variables → indepvar,
       Dependent⌣Variables → depvar,
       FullChangeFunction → f]}, ¶10,
     {¶0, SuperWidgetButton[Null, "OK", 1], ¶0}, ¶10}, Title -> "Euclidean Geometry"
   ] // SuperGUIRun;
```

This very simple example illustrates a well known theorem in geometry that a triangle inscribed in a circle so that one side is a diameter will be right angled. If you move the mouse over point C, you will see it respond – meaning it is draggable. Try dragging the point round the circle and watch the data in the rest of the GUI adjust accordingly. Note that the dependent variables list contains rules that constrain the point C to lie on the circle. Because all the coordinates in this example have z=0, and because it is viewed from directly above (positive z) the diagram is essentially 2-dimensional. For this reason, the `Mouse⌣Dragable->False` option is used to prevent the 'paper' being rotated by the mouse.

JavaView

About JavaView

JavaView is another freely available Java program that displays graphics in a Java window and lets you manipulate the image with the mouse. It was written by Konrad Polthier and is available here. In many ways it is similar to LiveGraphic-s3D, but the graphics are generally considered to be superior, and it is more extensively programmable via J/Link (but you need to know a little Java). JavaView can be used in many environments (not just *Mathematica*) but it comes with some interface code for *Mathematica*, which you should install. You can download JavaView for free, but you have to register the software before it can be used sensibly. Note that the registration step is also free.

Tip! When you register JavaView, you will be sent a file called jv-lic.lic, to be placed in the JavaView\rsrc directory. When installed on a machine with *Mathematica*, there are two such directories – one in the *Mathematica* AddOns tree, and one in the main software directory tree (typically c:\Program Files under Windows). It is important that a copy of this license file is placed in both the JavaView\rsrc directories.

In order to control some of the more advanced features of JavaView – such as the ability to map textures on to 3-D objects – it is necessary to use J/Link, so some of the examples in this section assume a knowledge of J/Link and Java, although this is not necessary in order to use JavaView as a simple viewer.

SuperWidgetJavaView

`SuperWidgetJavaView[v,opts]`	Displays *Mathematica* graphics (usually 3-D) stored in v. The graphics can be rotated by dragging with the mouse.

JavaView super widget

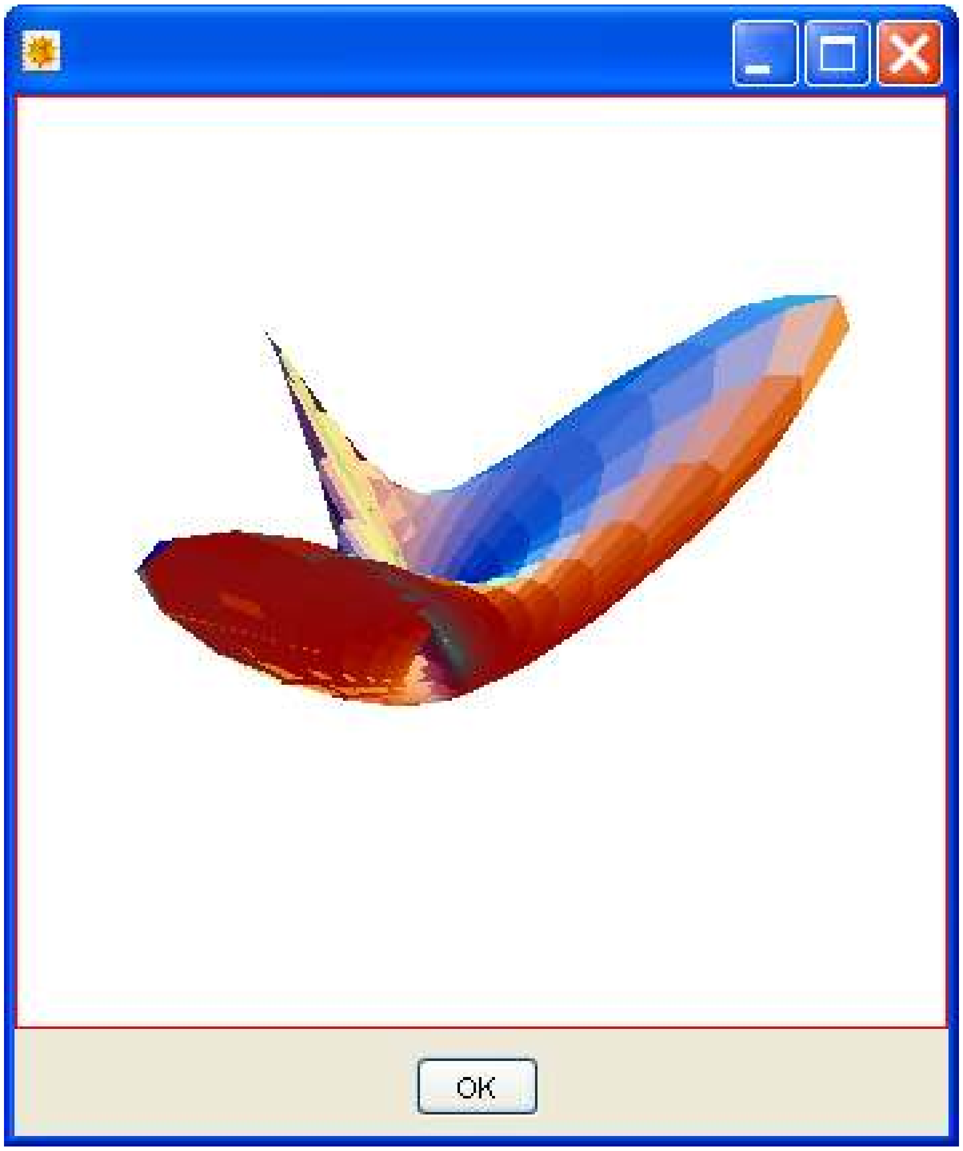
`Pixel⌣Width`	Width in pixels for the applet.
`Pixel⌣Height`	Height in pixelts for the applet.
`Pick`⌣Camera⌣Listener	Specifies a function to be called when a camera pick event occurs (see below).
Drag⌣`Camera`⌣`Listener`	Specifies a function to be called when a camera drag event occurs (see below).

Options for SuperWidgetJavaView

Using the `SuperWidgetJavaView`, it is possible to embed a JavaView window (or windows!) in a SWP GUI:

Here is a simple example:

```
zzz =
  ParametricPlot3D[{(√2 Cos[v]^2 Cos[2 u] + Cos[u] Sin[2 v]) / (2 - √2 Sin[3 u] Sin[2 v]),
    (√2 Cos[v]^2 Sin[2 u] + Cos[u] Sin[2 v]) / (2 - √2 Sin[3 u] Sin[2 v]),
    3 Cos[v]^2 / (2 - √2 Sin[3 u] Sin[2 v])}, {u, -π / 2, π / 2},
   {v, 0, π}, DisplayFunction → Identity, Axes → False, Boxed → False];
zzz[[1]] = Prepend[zzz[[1]], EdgeForm[]];
{SuperWidgetJavaView[zzz],
   ¶10,
   {¶0, SuperWidgetButton[Null, "OK", 1], ¶0}, ¶10} // SuperGUIRun;
```



Tip ! Normally, `ParametricPlot3D` produces an image in which the edges of each triangle are drawn in black.

To suppress these, the above code splices in the EdgeForm[] directive ahead of the rest of the graphics. Of course, this trick can be used generally, not just in the SWP/JavaView context.

JavaView objects can be manipulated using J/Link in many ways. Typically these operations use the geometry and display objects belonging to the JavaView widget.

Tip ! Under *Mathematica* 6.0 the above example issues a warning about a package. This does not seem relevant to the use of JavaView inside the SWP, and will doubtless be fixed in a subsequent version of the software.

Camera listeners

JavaView supplies several 'listener classes'. The most important one is the camera listener, which can be hooked up directly using options to SuperWidgetJavaView. The function should accept two arguments – the event class, and a second argument which seems to always contain the string "hello". Perhaps a future version of JavaView will remove this argument, so it may be best to code the function to take an arbitrary number of arguments.

For full details of camera events, consult the JavaView documentation, but pick events happen at the start of a drag, followed by a sequence of drag events.

To avoid a buildup of unwanted Java objects, the first argument (the event) should be removed using RemoveJavaObject when no longer required.

Manipulating JavaView objects

`Get⌣JavaView⌣Objects[v]`	DReturns a list of Java objects belonging to the JavaView widget associated with variable v. Currently this list contains the geometry object and the display object. Further objects may be appended to this list in future versions.

Get⌣JavaView⌣Objects function

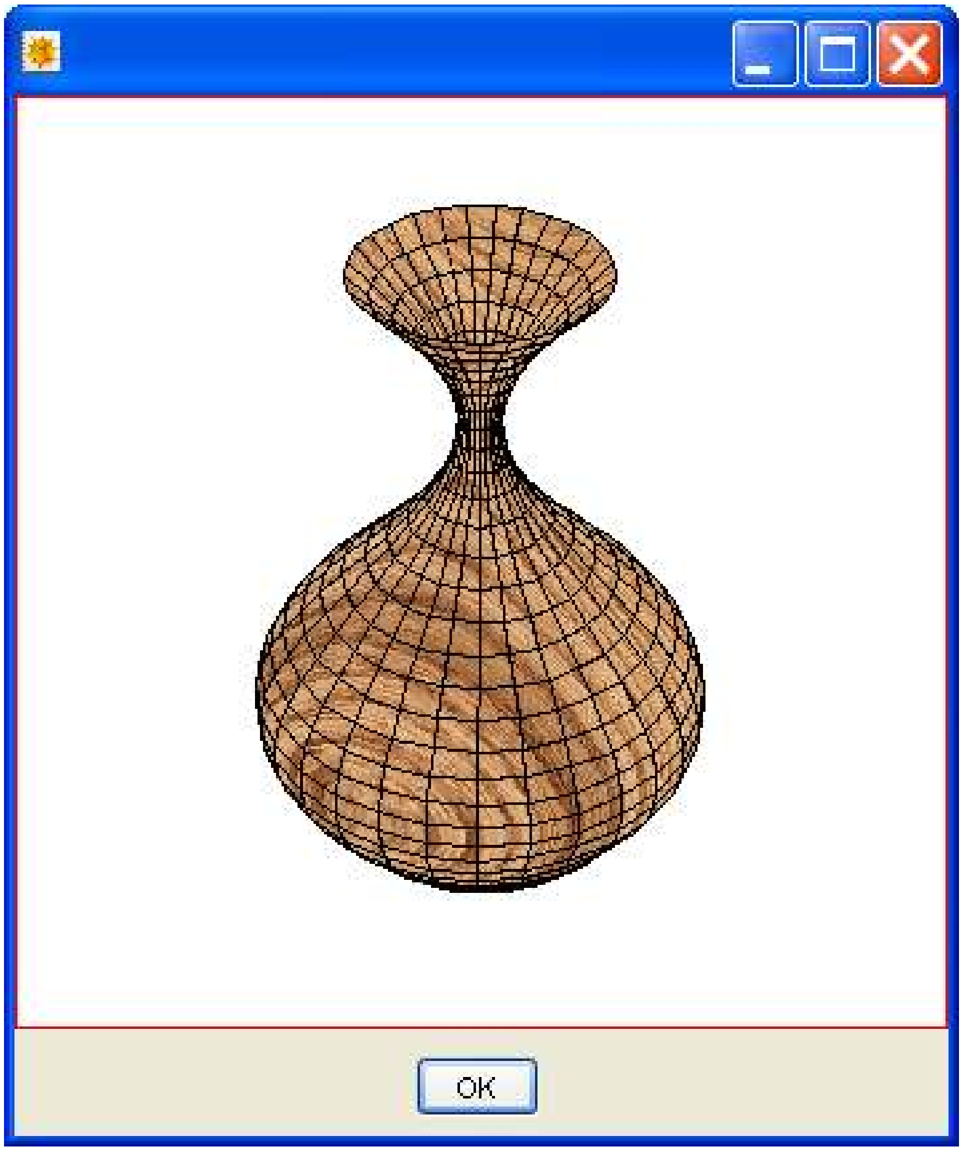
A JavaView display is controlled by a number of Java objects. The traditional way to use JavaView is to manipulate these objects directly using J/Link. Although this is not usually necessary in simple examples, it is sometimes useful to obtain these objects – as in the texture mapping example that follows.

Using texture mapping

This example exploits some of the more advanced features of JavaView to perform texture mapping on a 3-D structure. Consult the JavaView manual for more details of the J/Link calls that are used here.

Tip ! Note also, that because the pathname is consumed by Java code and not by *Mathematica*, the correct separator characters must be used to form the path to the texture file. This is obtained from `$PathnameSeparator`.

```
<< JavaView`JLink`;
<< Graphics`SurfaceOfRevolution`;

texturise[] := Module[{geom, tex},
   geom = Get⌣JavaView⌣Objects[zzz];
   tex = TextureImage[SuperWidgetPackagePath[] <>
       $PathnameSeparator <> "ExampleFiles" <> $PathnameSeparator <> "wood.gif"];
   geom[[1]]@setTexture[tex];
   geom[[1]]@makeVertexTextureFromBndBox[0, 1];
   geom[[1]]@showVertexTexture[True];
   geom[[1]]@update[geom[[1]]]
  ];




zzz = SurfaceOfRevolution[{1.2 + Sin[x], x},
   {x, 0, 2 Pi}, DisplayFunction → Identity, Axes → False, Boxed → False];

SuperGUIRunModal[{SuperWidgetJavaView[zzz],
  ¶10,
  {¶0, SuperWidgetButton[Null, "OK", 1], ¶0}, ¶10}, On⌣Display → texturise]
```



Observe that the surface of revolution is displayed in the normal way, and then the `On⌄Display` option to `SuperGUIRun⌄Modal` executes a function that modifies the image to apply a wood texture.

Frames, Dialog boxes, and menus

Introduction to Frames and Dialogs

`SuperWidgetFrame[ v,contents,opts]`	Displays a frame with the given contents. The contents should be a list (typically nested) of super widgets, widgets, and widget layout operations. SuperGUIRun and SuperGUIRunModal each supply a frame if none is supplied, however by providing one explicitly, the Menu, Title, Tool˷Bar, and CloseFunction options can be supplied. The associated variable, v may also be used in the Close˷Frame function.
`SuperWidgetDialog[ v,contents,opts]`	Displays a dialog box with the given contents. The contents should be a list (typically nested) of super widgets, widgets, and widget layout operations. Dialog boxes cannot accept a menu and do not have maximise/minimise buttons in their caption bars.
Replace˷Menu[v,menu]	Replaces the menu if a SuperWidgetFrame. This can be used to create dynamic menus – analogous to the *Mathematica* Windows menu which changes as notebooks are opened or closed.

Frame and Dialog super widgets

SuperWidgetFrame or `SuperWidgetDialog` supply the window that surrounds the various widgets that it contains. A `SuperWidgetFrame` is supplied implicitly, where required, but by providing it explicitly you can customise it using its various options:

Menu	Option for SuperWidgetFrame only – specifies a menu that appears immediately beneath the caption bar. See below for details regarding the construction of a menu.
Tool⌣Bar	Option for SuperWidgetFrame only – specifies a tool bar that appears immediately beneath the menu (if present). See below for details regarding the construction of a tool bar.
Title	Specifies a title (as a string) for the window. The title will appear in the caption bar.
CloseFunction	Specifies a function which is called with the associated variable v when the window closes.
Window⌣Position	Specifies the position of the top left hand corner of the frame or dialog with respect to the top left of the screen.
Bare⌣Window	Option for SuperWidgetFrame only – specifies that the window appear without title or borders. This is typically used for very temporary information windows, and care should be taken to ensure that such a windo is removed when not required, because the user cannot do this explicitly.
Background⌣Wallpaper	Specifies an image file to use as a background pattern to the frame.
Palette⌣Style	Used together with Desktop opton to indicate that the new frame should appear as a floating palette.

Options for Frame and Dialog super widgets

Menus

Menus are typically attached to the main window of an application, and are one of the main ways of presenting the functionality of the program. A menu consists of a list of menu-items, which have one of three possible structures:

{Name,action-function}

{Name,action-function,accelerator}

{Name,sub-menu}

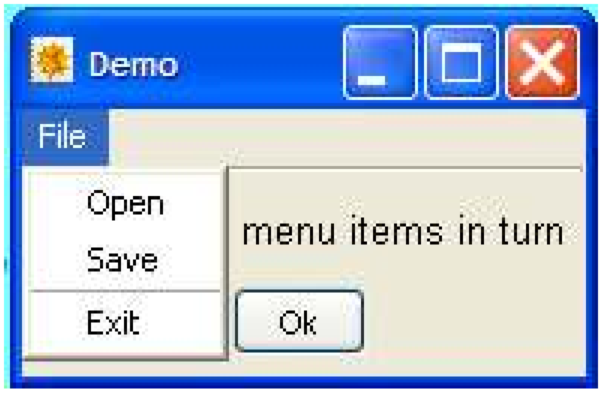
Menus can also contain Menu⌣Separator items to split a menu into sections separated by a line.

For example, here is a menu consisting of just one item – the "File" item. This item is defined , not by a function, but by a sub-menu containing three menu items with the third separated from the other two. Menu functions take no arguments.

```
Needs["SuperWidgetPackage`"]

{{"File",
  {{"Open", open⌣func}, {"Save", save⌣func}, Menu⌣Separator, {"Exit", exit⌣func}}}}
```

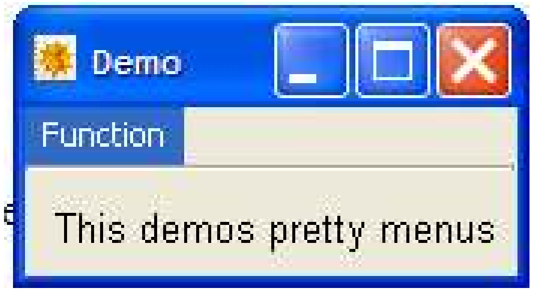
Here is a complete example:

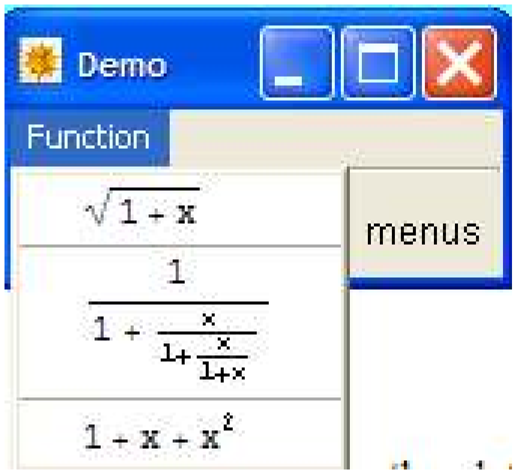
```
open⌣func[] := Print["Open"];
save⌣func[] := Print["Save"];
exit⌣func[] := Print["Exit"];
SuperWidgetFrame[fr, {
    ¶10,
    {¶10, SuperWidgetLabel[ll1, "Select the menu items in turn"], ¶10},
    ¶10,
    {¶0, SuperWidgetButton[bb1, "Ok", 1], ¶0},
    ¶10
  },
  Title -> "Demo", Menu -> {{"File",
     {{"Open", open⌣func}, {"Save", save⌣func}, Menu⌣Separator, {"Exit", exit⌣func}}}}
 ] // SuperGUIRunModal

    Save

1
```



Notice how the "File" menu contains a sub-menu rather than a function name. Menus can be nested to an arbitrary depth. Because the function names are also used to refer to menus in certain situations (analogous to the first argument of a super widget), the functions must be specified as actual function names – not as pure functions. It is also possible to create menus with images – which is particularly convenient for mathematical expressions, which are usually unprintable as Java strings:

```
Clear[x];
SuperWidgetFrame[fr, {
    ¶10,
    {¶10, "This demos pretty menus", ¶10},
    ¶10
  },
  Title -> "Demo",
  Menu -> {{"Function", {{Image⌣Expression[Sqrt[x + 1]], fn1}, Menu⌣Separator,
       {Image⌣Expression[1 / (1 + x / (1 + x / (1 + x)))], fn2},
       Menu⌣Separator, {Image⌣Expression[1 + x + x^2], fn3}}}}
 ] // SuperGUIRunModal
```





To understand the real power of menus, it is worth looking carefully at a specific example – say *Mathematica*'s Cell>Convert to>InputForm (this refers to version 5.2, menu details sometimes vary between versions). Although this menu can be accessed using the mouse, many people find it more convenient to use the keyboard. This particular menu item can be accessed using the keyboard in two different ways. If you open the menu, you will see that it can be accessed using Shift-Ctrl-I – this is known as a keyboard accelerator. Alternatively, on some platforms (including Windows), it is possible to open menus progressively using menu mnemonics. In this case, if you hold the Alt key down, you will see that the 'C' of the cell menu is underlined. Keeping the Alt key down and pressing 'C' will open the cell menu, where you will see that each option also has an underlined letter - 'C' takes us to the sub-menu required, etc. The sequence Alt-CCI is known as a keyboard mnemonic. Mnemonics are very heavily used on platforms that support them, and are very easy to specify using the SWP

The accelerator, if present, consists of a (case insensitive) string that defines a key with modifiers that will operate the menu function directly. Here are a few examples of accelerators:

"Ctrl-A"

"Shift-Ctrl-Z"

"Alt-Q"

"Meta-Q"

Note that the Meta key does not apply to Windows platforms.

To specify a mnemonic, simply include an '&' character in the menu name (e.g. "&File") – the '&' will be removed, and on suitable platforms, the next character ('F' in this case) will be underlined as part of the mnemonic. No special action is required to support platforms such as the Mac which do not support mnemonics – the mnemonic will simply be ignored.

Tip ! The entire menu bar for *Mathematica* is defined in the file MENUSETUP.TR using the same '&' notation to specify mnemonics as is used here.

There is no problem if your keyboard accelerators/mnemonics clash with those of *Mathematica*. If your GUI application is in focus, its menu is in effect – not that of *Mathematica*.

Tip ! In complex GUI programs, not all menu options make sense in all situations. For example, many menu options may not make sense until the user has actually opened a file – say by using the 'file/open' menu item. Menu items can ge disabled (which gives them a washed out appearance) by passing the name of their associated function to `Set⌣Enabled⌣Status`.

Toolbars

A toolbar is a strip of widgets positioned at the top of a window underneath the menu (if there is one). Typically it contains small image buttons and combo boxes. The `Tool⌣Bar` option can be used on `SuperWidgetFrame`, and you simply specify a list of super widgets to place on the bar.

```
Needs["SuperWidgetPackage`"]

new⌣func[___] := Print["New"];
open⌣func[___] := Print["Open"];
save⌣func[___] := Print["Save"];
exit⌣func[___] := (Print["Exit"]; Close⌣Frame[fr]);

SuperWidgetFrame[fr, {
   ¶10,
   {¶10, "Select the menu items in turn", ¶10},
   ¶10,
   {¶0, SuperWidgetButton[bb1, "Ok", 1], ¶0},
   ¶10
  },
  Tool⌣Bar → {
    SuperWidgetButton[tb1,
     Image⌣File[SuperWidgetPackagePath[] <> "/ExampleFiles/new.gif"], new⌣func],
    SuperWidgetButton[tb2, Image⌣File[SuperWidgetPackagePath[] <>
       "/ExampleFiles/open.gif"], open⌣func],
    Menu⌣Separator,
    SuperWidgetButton[tb3,
     Image⌣File[SuperWidgetPackagePath[] <> "/ExampleFiles/save.gif"], save⌣func]
   },
  Title -> "Demo",
  Menu -> {{"File", {{"New", new⌣func}, {"Open", open⌣func}, {"Save", save⌣func},
       Menu⌣Separator, {"Exit", exit⌣func}}}}
 ] // SuperGUIRunModal
```

Leaving a shadow

We have already seen a number of programs that use `Close⌣Frame` to destroy a window explicitly. The funstion `Close⌣Frame` can take the option `Leave⌣Shadow->True.`

Using this option, you destroy the window, but leave a non-functional copy of the window visible on the screen until the SWP puts up the next window. This has a number of practical uses:

• If you want to replace one window with another of the same size, but containing some different information, closing the first using this option will create the illusion that you have somehow updated the window, not replaced it – i.e. there will be no glitch.

• Many serious programs must do considerable work before they can display their first window. By greating a window – possibly containing some relevant graphics – and then closing it with the Leave⌣Shadow option, you will create a splash screen that will keep your users focussed until your real GUI fires up. This reproduces the behaviour of many GUI programs – including *Mathematica* itself.

• If you provide a button that initiates a task taking more than a second or so (remember that your user's may not all be using fast processors), you can use a shadowed window to tell them to wait for a moment. Longer waits may be better handled using a window with a progress bar, but this is inevitably more complex to program.

For example, the following code will display a simple splash screen while a Pause simulates a complex startup procedure:

```
SuperWidgetFrame[splash⌣screen, {Image⌣File[
      SuperWidgetPackagePath[] <> "/ExampleFiles/consultancy.gif"]}] // SuperGUIRun;
Close⌣Frame[splash⌣screen, Leave⌣Shadow -> True];
Pause[5];
ShowMessageBox["The rest of the program", "", {"OK"}]
```

It is also possible to fade the shadow by mixing it with another colour by adding the option `Fade⌣Colour->Blue` (say, use any colour specification here). This can sometimes be helpful to remind users that the controls on the shadow are inactive!

SuperWidgetTextEditor

`SuperWidgetTextEditor[v,opts]`	Implements a text editor, where v contains the string of text.The text may contain newlines, and will flow across multiple lines as required (so it could consist of several paragraphs).The string must only contain normal Java characters.The ChangeFunction option can be used to monitor the changes made by the user.

Text editor super widget

`ChangeFunction`	Function to call when the user alters the text.
`Font`	Font specification in the form {Name,face,size}
`Tool`Tip	String to use as the tool tip.
Panel Margins	Pixel margin to surround editing area – default {12,12,12,12}
Pixel Width	Pixel width for the widget.
Pixel Height	Pixel height for the widget.
Editable	Specifies whether the text can be altered (default True).

This creates a simple text editing panel. The control variable is set to the initial value of the text string, and is updated as the user makes changes. The text string can contain newlines, and will flow across multiple lines, with scroll bars as required. Thus large quantities of text can be handled. The option `ChangeFunction` can be used to specify a 1-argument function to be invoked each time a change is made by the user. The options `Pixel Width` and `Pixel Height` can be used to override the default size of this widget. Also, it is possible to change the value of the control variable, and call `UpdateWidgetValue` to change the contents programmatically. Here we read a file from the ExampleFiles directory and manually replace most of the text with the name of a vegetable:

```
Needs["SuperWidgetPackage`"]

Module[{str, v},
 str = OpenRead["SuperWidgetPackage/ExampleFiles/TextToEdit"];
 v = Read[str, Record, RecordSeparators → {}];
 Close[str];
 SuperWidgetTextEditor[v, Pixel Width → 300, Pixel Height → 200] // SuperGUIRunModal;
 v
]
```

```
The SWP is supplied as a ZIP file containing all the files required in their
  appropriate directories.. It is vital that this directory structure is
  preserved. Copy the file to the Mathematica directory, e.g. C:\program
  files\wolfram research\mathematica\5.1 (the exact directory depends on the
  version of Mathematica installed). Unzip the file SuperWidgetPackage.ZIP
  using a tool that preserves the directory structure and handles
  long names correctly, e.g. PKZIP(R)  Version 2.50, or WinZip(R).
Finally, start Mathematica, click on the 'Help' menu, and select 'Rebuild Help
  index'. This will integrate the SWP documentation with the rest of Mathematica.
  You will find the SWP help in the 'Addons' section of the help browser.
```

Note that by setting `Editable->False` you can display a portion of text (with scrolling if necessary) but prevent the user from modifying it.

Exploiting HTML

`SuperWidgetHTMLPanel[v,opts]`	Displays HTML string stored in v. Can respond to hyperlinks by loading other pages (e.g. from the internet), or be executing *Mathematica* code.

HTML super widget

This creates an HTML panel. The first argument is the associated variable, and should be set to a text string containing the HTML. The result is laid out in the window, and if the HTML contains URL links, these are clickable and will start the browser. The options `Pixel⌣Width` and `Pixel⌣Height` can be used to set the size of the panel. The option `Pixel⌣Margins` can be set to an array of four integers (left, top,bottom,right) representing the margin size in pixel. A 15 pixel margin is used by default. A single integer can be used to set all four margins to the same value. The Background option may also be specified to set an overall background colour for the panel. Each of the examples in the examples section has a Help/About box constructed with this super widget.

This super widget also recognises a special form of 'URL', beginning mathcommand:// – for example: mathcommand://Print[42] – which can be used to execute a command when the link is pressed. Since the syntax of HTML would preclude such a command containing a double quote symbol, the sterling symbol '£' can be used instead of a double quote.

Although the text cannot be edited by the user, it can be changed by altering the value of the associated variable and using `UpdateWidgetValue`. This could be particularly useful in 'Wizard' - like applications where you want to display some explanatory text which changes as things progress.

Here is a small illustration of what is possible. Obviously, some knowledge of HTML (but none of cooking!) is necessary to interpret this example:

```
Needs["SuperWidgetPackage`"]

page1 = "<html><body>" <>
   "<h1 align=\"center\">Baking a jalopeno cake: part 1.</h1>" <>
   "<p>Start with a TESCO sponge cake mix, and prepare it according to
     instructions until it is ready to be put in the oven.</p><p>Add
     1 ounce of chopped jalopenos (or more, if desired).</p> <p><a
     href=\"mathcommand://Replacepage[]\">Next step</a></p>" <>
   "</p></body></html>";
page2 = "<html><body>" <>
   "<h1 align=\"center\">Baking a jalopeno cake: part 2.</h1>" <>
   "<p>After thorough mixing, bake your cake in the oven according
     to TESCO instructions. </p><p>While your cake is cooling, you
     may wish to purchase some indigestion tablets just in case
     you find the result a little tough on the stomach!</p> <p><a
     href=\"mathcommand://Replacepage[]\">Previous step</a></p>" <>
   "</p></body></html>";
this⌣page = page1;
Replacepage[] := (If[this⌣page == page1, this⌣page = page2, this⌣page = page1];
   UpdateWidgetValue[this⌣page]);
SuperWidgetHTMLPanel[this⌣page, Background → RGBColor[0.8, 0.8, 1]] // SuperGUIRunModal
Null
```

• Note: as you can see above, HTML text often contains quoted text. As is usual with *Mathematica*, to obtain a quote character inside a string you must preceded it with a backslash character. Some care is required to do this correctly – particularly in larger examples. You may wish to either read the HTML text from a file, and display it with `FullForm`, which will show all the escape characters correctly. The result could then be pasted into your program. Alternatively, you could read a file (or even something off the Internet, using GetURL) as your program executes.

• Observe that you could produce your entire application using links within HTML to drive it.

HTML strings can also be inserted in many other places where super widgets require text. This can be used to produce extremely fancy buttons, combo boxes, tool tips, etc. See the section on using HTML strings inside super widgets.

SuperWidgetTable

`SuperWidgetTable[v,opts]`	Displays a table represented by the structure stored in the associated variable v, which should contain a rectangular array of reals.

Table super widget

`Table⌣Headings`	Can be set to an array of strings. The array should contain as many elements as there are columns in the array.
`ChangeFunction`	Function that will be called each time the data is changed by the user. Takes one argument – the associated variable v.
`Columns⌣Editable`	A list of boolean values or All or None - indicating which columns can be edited.
`Tool⌣Tip`	Specifies a tooltip string.
`Pixel⌣Width`	Width in pixels, default 300. (scrolling will be used if the table size is not set large enough).
`Pixel⌣Height`	Height in pixels, default 200 (scrolling will be used if the table size is not set large enough).
Digits⌣After⌣Point	Defaults to Automatic, can be set to a list (one per column) of integers specifying the number of digits after the decimal point used to display Real data.

Options for Table super widget

This creates a grid of data values, which may optionally be edited. The control variable holds the array of data, which can be real, integer, string, or boolean (True/False) and the columns can have title information. Here is a simple example:

```
Needs["SuperWidgetPackage`"]

z = IdentityMatrix[5] // N;
zinv = Inverse[z] // N;
tfn[_] := Module[{},
    zinv = Inverse[z] // N;
    UpdateWidgetValue[zinv];
   ];
hd = Table["Col " <> ToString[i], {i, 1, 5}];
{
   {¶0, "Input table", ¶0}, ¶10, SuperWidgetTable[z,
    Table⌣Headings → hd, Columns⌣Editable → All, ChangeFunction → tfn],
   ¶10,
   SuperWidgetTable[zinv, Table⌣Headings → hd, Columns⌣Editable → None],
   ¶10, {¶0, SuperWidgetButton[Null, "OK", 1, Tool⌣Tip → "Press to finish"], ¶0}} //
  SuperGUIRun

- GUIObject -
```

Only the top array is editable, the bottom one displays the inverse of the matrix and is updated on the fly. The following options may be used:

`Table⌣Headings` – Either `Null`, or a list of heading strings of the correct length for the number of columns of the matrix.

`ChangeFunction` – Function to be called when the data is changed by the user. The updated control variable will be passed, and as usual, it may be useful to declare the function to have attribute HoldFirst.

`Columns⌣Editable` – Either a boolean array with an entry for each column to indicate if it is editable, or `All` or `None`.

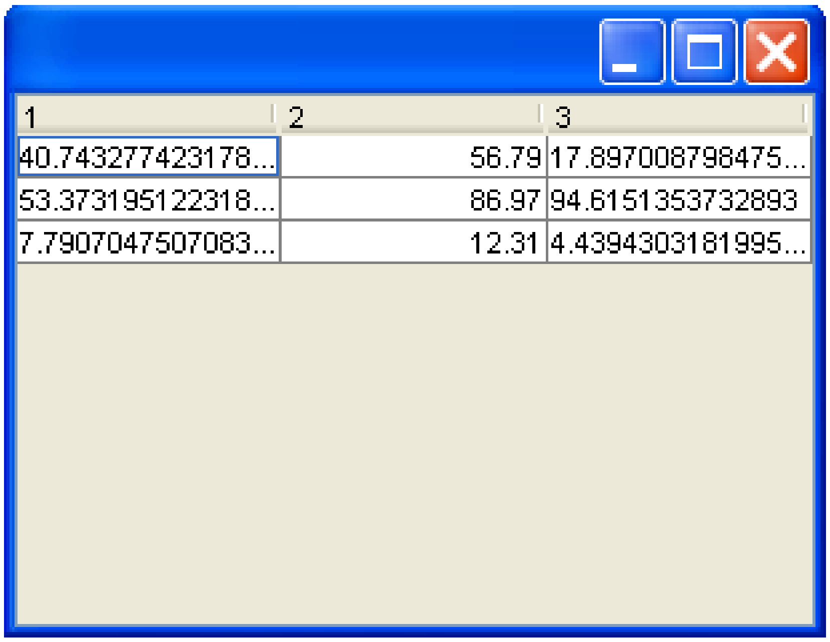
`Tool⌣Tip` – String to be used as a tooltip.

`Pixel⌣Width`, `Pixel⌣Height` – Specify the size of the control. If the array is too large for the specified size, scroll bars will be used.

Note carefully that data within each column should be either all integer or all Real – not mixed, fractional, or complex. Typically you might want to use `//N` or the data as it is being set up.

Individual columns can be set up to display Real numbers in fixed point format (very useful for currency values) by using the `Digits⌣After⌣Point` option. For example:

```
tt = Table[Random[] * 100, {k, 1, 3}, {j, 1, 3}];

SuperWidgetTable[tt, Digits⌣After⌣Point → {Automatic, 2, Automatic}] // SuperGUIRunModal
```



1	2	3
40.743277423178...	56.79	17.897008798475...
53.373195122318...	86.97	94.6151353732893
7.7907047507083...	12.31	4.4394303181995...

```
- Graphics -
```

Here the centre column has been formatted in fixed format, the other two columns have been left with the Automatic setting. Note that fixed format numbers must be suitable in size for display without an exponent. The fixed point display uses right alignment.

SuperWidgetTree

`SuperWidgetTree[v,opts]`	Displays a tree stored in the associated variable v.

Tree super widget

`Background`	Background colour – default RGBColor[1,1,1]
`SelectionFunction`	Function that will be called when a leaf node is selected. The argument will be the associated variable.
`Double⌣Click⌣Function`	Function that will be called a leaf node is double-clicked. The argument will be the associated variable. The first click of the double click will have already selected the node.
`Tool⌣Tip`	Specifies a tooltip string.
`Pixel⌣Width`	Width in pixels, default 300. (scrolling will be used if the table size is not set large enough).
`Pixel⌣Height`	Height in pixels, default 200 (scrolling will be used if the table size is not set large enough).

Options for Tree super widget

`Tree⌣Node[name,spare,sel]`	Represents a leaf (terminal) node with name and selection indicator (0 or 1). The spare argument can be used to hold user data
`Tree⌣Node[name, spare,sel,sub-node-list]`	Represents a non-terminal node.

Representation of tree structure

This creates a tree representation of a data structure, which should be setup as in this example:

```
Needs["SuperWidgetPackage`"]

x = Tree⌣Node["Language", 0, 0, {Tree⌣Node["French", 1,
      0, {Tree⌣Node["Country", 0, 0], Tree⌣Node["Dictionary", 0, 0]}],
     Tree⌣Node["English", 0, 0, {Tree⌣Node["Country", 0, 0], Tree⌣Node["England", 0, 0],
       Tree⌣Node["USA", 0, 0], Tree⌣Node["Australia", 0, 0]}],
     Tree⌣Node["Dictionary", 0, 0]}];
```

The first argument of each `Tree⌣Node` object is the name of that node, the second argument is spare, and could be used to hold additional data. The third argument will be 1 if that node is selected, and the fourth argument is a list of sub-nodes of the tree. It is omitted for a terminal node. The `SelectionFunction` option can be used to supply a function to be called each time a selection is made. Here is a simple example which displays all the permutations of five objects as a tree:

```
tree⌣fn[x_] := Print[Cases[x, Tree⌣Node[_, _, 1], ∞]];
build⌣tree[prefix_String, items_List] := Module[{s},
    If[Length[items] == 0,
     Tree⌣Node[prefix, 0, 0],
     Tree⌣Node[prefix <> "...", 0, 0,
      Map[build⌣tree[prefix <> ToString[#], DeleteCases[items, #]] &, items]
     ]
    ]
   ];
x = build⌣tree["", {1, 2, 3, 4, 5}];
SuperWidgetTree[x, SelectionFunction → tree⌣fn,
   Pixel⌣Width → 300, Pixel⌣Height → 300] // SuperGUIRunModal

    {}

    {Tree⌣Node[32415, 0, 1]}
```

A more substantial example of the use of this widget is included in the larger examples.

Tip ! Use the single-click function to perform reversible operations, and the double-click function (if any) to perform less easily reversed operations – the double click is a more deliberate act. Some people have difficulty performing a double click - so it is helpful to provide a button that does the same operation.

SuperWidgetPanel

`SuperWidgetPanel[v,widgetlist1, opts]`	Creates a panel which contains other widgets. The panel is not itself visible, but is useful as a way of grouping widgets for more elaborate layouts.

Panel super widget

Border⌣Color	Border colour – default RGBColor[0,0,0], or no border if Border⌣Size is not set. The UK spelling – Border⌣Colour will also work.
Border⌣Size	Border size – defaults to 0 unless Border⌣Color has been set, when it defaults to 1.

Options for Panel super widget

A common use for a panel is to make a vertical display of buttons to be placed alongside another, larger control. For example:

```
Needs["SuperWidgetPackage`"]
```

```
txt = "To be or not to be\n(To be continued.....)";
{{{SuperWidgetPanel[Null,
{SuperWidgetButton[Null, "Test1", fff],
SuperWidgetButton[Null, "Test2", fff],
SuperWidgetButton[Null, "Test3", fff],
SuperWidgetButton[Null, "Test4", fff]
}
], ¶0}, ¶10, SuperWidgetTextEditor[txt, Pixel⌣Width → 300, Pixel⌣Height → 200]}} //
  SuperGUIRunModal
```

The border options can be used to achieve special effects, and can also sometimes be useful to pick out the location of panels while trying to achieve particular layout designs.

SuperWidgetTabPanel

SuperWidgetTabPanel[v,{{<name1>,widgetlist1},...}]	Creates a tab panel where each 'pane' has the specified name and contents (more super widgets). This is useful for condensing a large number of options into a small panel.

Tab panel super widget

Minimum⌣Width	Forces a minimum width in pixels for the structure – to ensure the tabs are displayed in one line.

Options for SuperWidgetTabPanel

This creates a tab panel, in which different super widgets are displayed on different panes which the user can select by clicking on their names. The first argument is the controlling variable, the second is a list of pairs. Each pair represents one sub-panel of this widget, and consists of a name and a list of super widgets to appear on that pane. All this is best illustrated by a simple example:

```
Needs["SuperWidgetPackage`"]

p1 = 1; p2 = 2; p3 = 3; p4 = 4; p5 = 5; p6 = 6;
SuperGUIRunModal[SuperWidgetTabPanel[tfr, {
    {"Parameters 1, 2, 3", {
      ¶10, {"Parameter 1", SuperWidgetIntegerBox[p1]},
      ¶10, {"Parameter 2", SuperWidgetIntegerBox[p2]},
      ¶10, {"Parameter 3", SuperWidgetIntegerBox[p3]},
      ¶10
     }} , {"Parameters 4, 5, 6", {
      ¶10, {"Parameter 4", SuperWidgetIntegerBox[p4]},
      ¶10, {"Parameter 5", SuperWidgetIntegerBox[p5]},
      ¶10, {"Parameter 6", SuperWidgetIntegerBox[p6]},
      ¶10
     }
    }
   }, Minimum⌣Width → 250
  ]
 ];
```

The option `Minimum⌄Width` can be used to force the tabs to be laid out horizontally. If you remove this option from the above example, the result is rather ugly.

SuperWidgetLabelledBox – grouping things in a pleasing way

`SuperWidgetLabelledBox[ v,label,contents,opts]`	Creates a box with a label to group other widgets.The' contents' should be a list of super widgets to include in the box.The box edges appear as if scored into the surface of the window, and the label is spliced in on the top edge.

Labelled box super widget

This widget is designed to make a scored rectangular box to group a set of controls within it. The arguments are control variable, name of box, and a list of the controls to placed within it. For example:

```
Needs["SuperWidgetPackage`"]

{¶10, {¶10, "Solution procedure", ¶10}, ¶10, SuperWidgetLabelledBox[
   Null, "Integration method", {¶20, SuperWidgetRadioButtonGroup[k,
     {"Analytic", "Series expansion", "Numerical integration"}], ¶20}],
  ¶10, {¶0, SuperWidgetButton[Null, "OK", 1], ¶0}, ¶10} // SuperGUIRunModal
```

Wizards

A wizard (at least in the jargon of GUI interfaces!) is a window of fixed size which steps the user through a sequence of operations. The left hand panel typically contains a list of all the steps with the current one highlighted. The right panel contains the super widgets required to obtain the data. The user is free to use the buttons at the bottom to navigate through the steps or to abort if he wishes. Wizards are top-level objects – in other words, they should not be embedded inside frames or other widgets – they are analogous to frames or dialogs. In the SWP, wizards are created using `SuperWidgetWizard`, and the individual pages of the wizard are represented by `Wizard⌄Page` objects.

`SuperWidgetWizard[v,contents, opts]`	Creates wizard with the given contents, which should be a list of . Wizard⌄Page objects. The control variable, v, is updated with number of the current page which is on display (remember that this can move in either direction).

Wizard super widget

CloseFunction	Function that is called when the wizard closes.
Side‿Bar‿Title	Title for the left had panel.
Title	Overall title.
Wizard‿Steps	A list of names for the various steps.
Page‿Turn‿Function	Function to call (with the control variable as argument) each time a new page is to be displayed.

Options for SuperWidgetWizard

`Wizard‿Page` objects are not super widgets as such, and have no control variable, they are only used inside `SuperWidgetWizard`.

Wizard‿Page[contents,opts]	Represents one page of a wizard with the given contents (a list of. super widgets).

Wizard page object

Title	Title for this page.

Options for Wizard‿Page

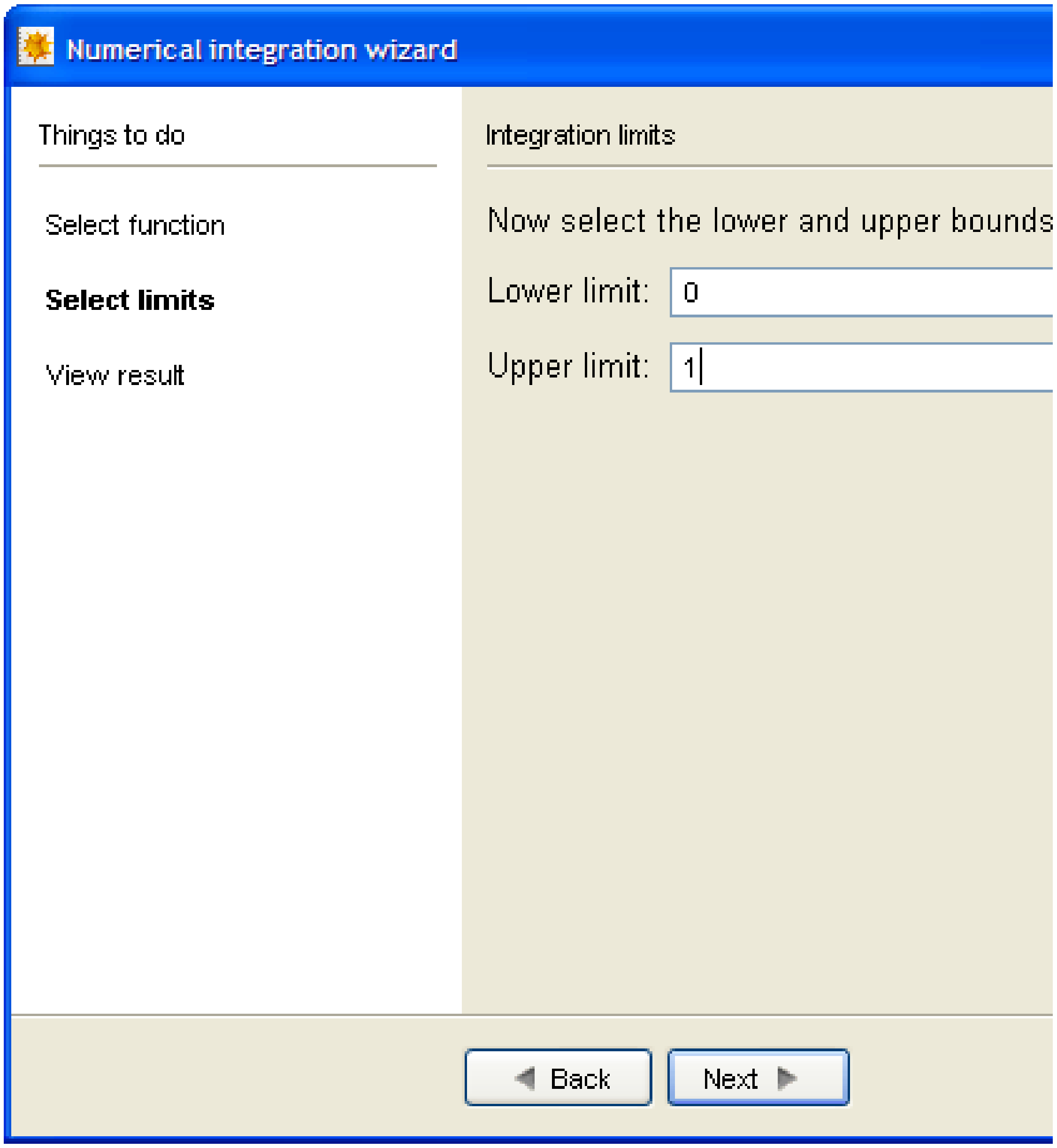
A wizard is normally created using `SuperGUIRunModal`, which will return 1 for a successful completion of the wizard, and 0 if the wizard is canceled or closed. It is important to test this value to avoid proceeding with a computation that the user intended to cancel.

Each page of the wizard contains three navigation buttons, "Back", "Cancel", and "Next" or "Finish" as appropriate. Usually some of these buttons need to be greyed out until suitable data has been supplied by the user. In the following numerical integration example, the `check‿ok` function prevents the user progressing to the third page of the wizard until he has entered both limits, and it also tests that the result will not be complex in the case that the `Sqrt` function is selected.

Tip ! Use Although it is tempting to avoid the extra complexity involved in controlling the button states, your user will not thank you for the result! Part of the unwritten 'contract' of using a wizard is that the program checks the data and keeps the user safe.

```
gui⌣integral[] :=
 Module[{fn, limit⌣1, limit⌣2, explanation⌣1, explanation⌣2, ptf, check⌣ok},
  fn = "Sin";

  check⌣ok[_] := Module[{},
    Print[limit⌣1, "   ", limit⌣2, "   ", NumericQ[limit⌣1]];
    If[NumericQ[limit⌣1] &&
      NumericQ[limit⌣2] && (fn ≠ "Sqrt" || (limit⌣1 ≥ 0 && limit⌣2 ≥ 0)),
     Set⌣Wizard⌣Button⌣State[xxx, "Next", 2, True],
     Set⌣Wizard⌣Button⌣State[xxx, "Next", 2, False]
    ]
   ];

limit⌣1 =.;
limit⌣2 =.;
explanation⌣1 =
   "Numerical integration operates on a \nfunction between numerical limits.
     First you must choose the function:\n";
explanation⌣2 = "Now select the lower and upper bounds of integration\n";

  ptf[page⌣no_] := If[page⌣no == 3, Set⌣Label⌣Contents[result⌣var,
     ToString[NIntegrate[ToExpression[fn][x], {x, limit⌣1, limit⌣2}]]]];

check⌣ok[0];
  SuperGUIRunModal[SuperWidgetWizard[xxx, {Wizard⌣Page[
       {
        explanation⌣1,
        ¶10,
        {"Requred function:",
          SuperWidgetComboBox[fn, {"Sin", "Cos", "Sqrt"}, ChangeFunction → check⌣ok]}
       }, Title -> "Select a function to integrate"], Wizard⌣Page[{explanation⌣2,
        ¶10,
        {"Lower limit: ", TAB, SuperWidgetRealBox[limit⌣1, ChangeFunction → check⌣ok]},
        ¶10,
        {"Upper limit: ", TAB, SuperWidgetRealBox[limit⌣2, ChangeFunction → check⌣ok]}},
       Title -> "Integration limits"],
      Wizard⌣Page[{{"The result of the integral is",
          SuperWidgetLabel[result⌣var, ""]}
       }, Title -> "Result"]},
    Side⌣Bar⌣Title -> "Things to do",
    Title -> "Numerical integration wizard",
    Wizard⌣Steps → {"Select function", "Select limits", "View result"},
    Page⌣Turn⌣Function → ptf]
  ]
 ]
```

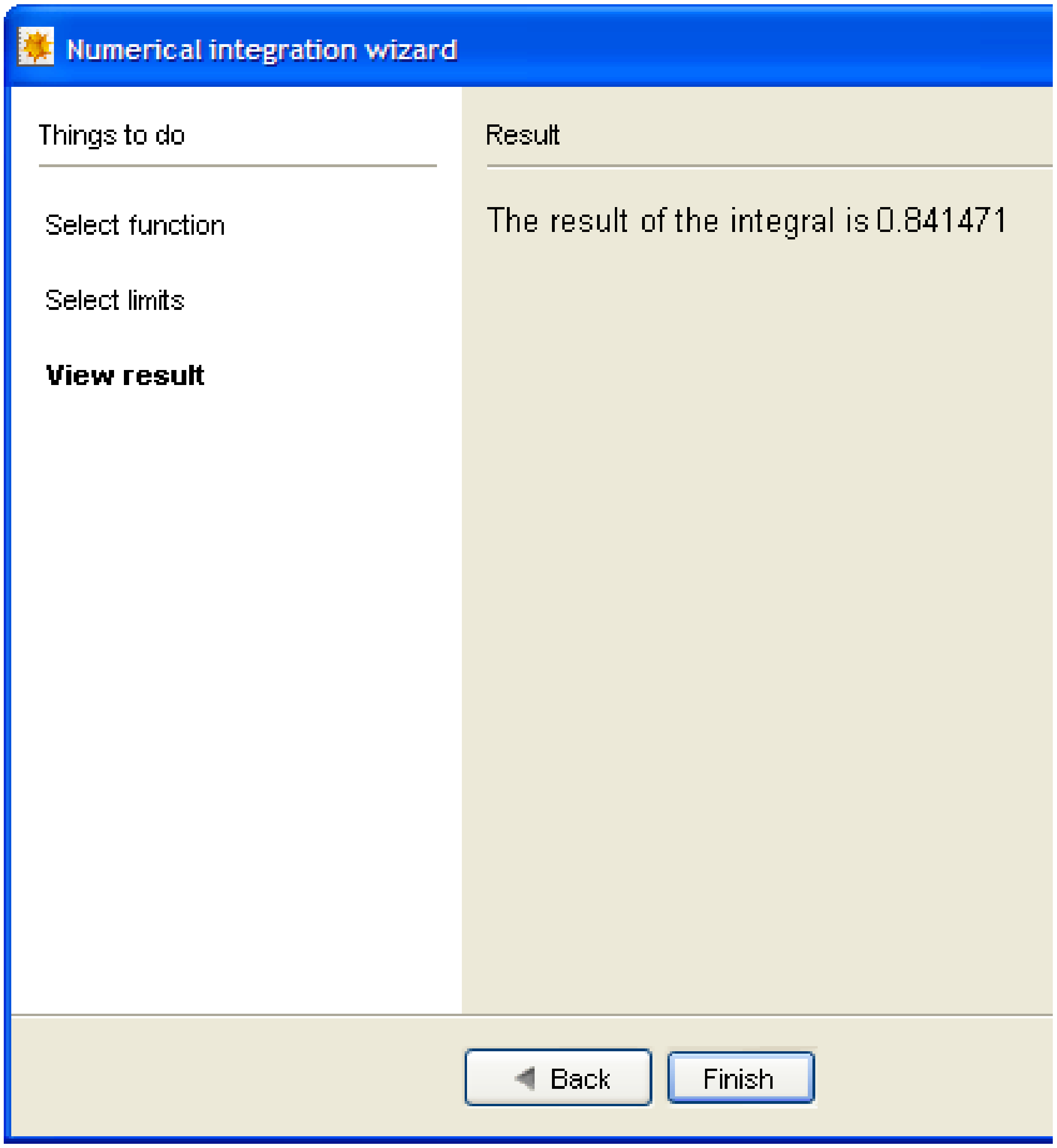
```
gui˘integral[]
    limit˘1$24954   limit˘2$24954   False
    0   limit˘2$24954   False
    0   1   False
0
```



Now select the lower and upper bounds

Lower limit: 0

Upper limit: 1



The result of the integral is 0.841471

Observe that a typical wizard contains a lot of super widgets (some for each page) and so may take a little longer to be displayed. Note also that every page of the wizard is created at once – even though all but the first are initially covered up. Thus, for example, if page 5 displays a graph created out of data collected on the previous four pages, you should supply a 'temporary graphic' to prevent problems.

Using UNICODE characters in input boxes

Java uses UNICODE for all its character manipulations – just like *Mathematica*. This means, that, in principle, UNICODE characters can be used in input boxes (In other contexts, it is probably easier to use an image). There are several details to consider:

• The normal Java font does not show all UNICODE characters, and some (unfortunately including \ [Breve]) are displayed as modifiers of a previous character.

• Although it is possible to paste UNICODE characters from *Mathematica* into Java input boxes, special provision, such as the use of accelerators, must be supplied to input such characters otherwise.

Here is a simple example:

```
alpha[] := (
   zz = zz <> "α";
   UpdateWidgetValue[zz]
  );
beta[] := (
   zz = zz <> "β";
   UpdateWidgetValue[zz]
  );
gamma[] := (
   zz = zz <> "Γ";
   UpdateWidgetValue[zz]
  );
zz = "α+β+Γ";
SuperWidgetFrame[Null, {{¶10, "Enter desired expression", ¶10},
   ¶10,
   SuperWidgetStringBox[zz]
  }, Menu → {{"Char", {{"Alpha", alpha, "Alt-A"},
      {"Beta", beta, "Alt-B"}, {"Gamma", gamma, "Alt-G"}}}}] // SuperGUIRunModal
```

Making arrays of widgets (new at Version 4.70)

Prior to version 4.70, it was quite hard to create an array of widgets because each widget required its own associated variable. Now it is much easier to achieve this using an array expression. However a little care is still needed because the array expression cannot contain variables because it is not evaluated immediately (super Widgets all have the HoldFirst attribute). For example:

```
data = {10, 20, 30};
Table[With[{k = k}, SuperWidgetIntegerBox[data[[k]]]], {k, 1, 3}] // SuperGUIRunModal;

data

{10, 42, 30}
```

Note that the older way to achieve this remains valid, but is superceded by this new mechanism.

Variable scoping

Experienced users of *Mathematica* will have noticed that many of the SWP examples use globally scoped variables. This is more or less inevitable because variables provide the 'glue' between the various widgets. For example, the variable that is passed to SuperWidgetIntegerBox is also the variable you would use to grey out the box. For this reason, it is recommended that you either use very long, distinctive names for global variables, or place SWP code inside a package. This can obviously be done after the code has been developed, and therefore none of the SWP examples have been complicated in this way.

Special restrictions applying to modeless windows

Normally it is suggested that you create windows using `SuperGUIRunModal`. This function will create a modal window which takes control while it is visible (except if it creates additional windows). Modal windows have no special limitations.

Modeless windows – created using `SuperGUIRun` – are analogous to palettes in that they can be accessed by the user as required – alongside other windows. A modeless window can also stay open after the *Mathematica* command that created it has completed and the FrontEnd is waiting for the next command. Modeless windows are subject to the following restriction:

A top-level modeless window cannot open additional windows – i.e. a button, menu, etc. that attempts to create an additional window will cause a fault.

Note that a modal top-level window can create both modal and modeless windows and these are not subject to this restriction because they are not top-level.

The modeless windows that are created as part of a multiple document interface (MDI) are also not subject to this restriction.

SuperWidgetDesktop, and the multiple document interface.

Start⌣Point	{x,y} location of first child window.

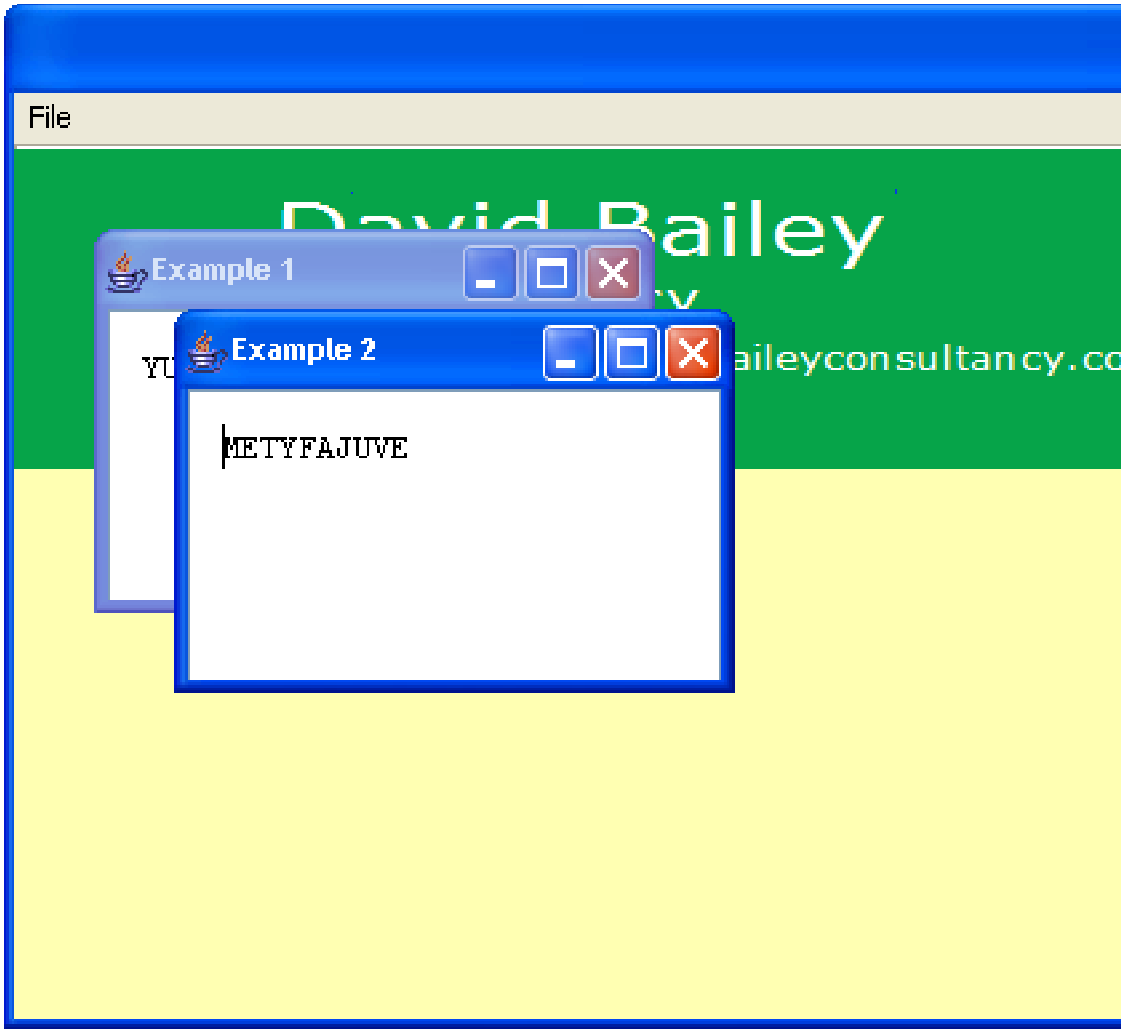
Options for SuperWidgetDesktop

Some GUI applications – such as word processors and image editors – use what is known (at least in Windows parlance) as a multiple document interface (MDI). The application consists of one large window containing several movable smaller windows within it. The user can work in any of these windows just by clicking between them (they are modeless among themselves) and can also access controls from the back window (typically just a menu and toolbars) without obscuring the various documents.

If you are designing an application in which you are thinking of using modeless windows, you should certainly consider if MDI would be suitable.

Creating an MDI effect is extremely simple. You create the back window as a modal window (that will normally remain visible for the entire duration of your program) in the normal way, but you include a SuperWidgetDesktop to represent the bulk of the window that you wish to use to display documents. It is hard to illustrate this effect usefully in a trivial example, but here is a program in which each time the File/New menu item is activated, a new text editor window is opened with a random text string. Notice that each sub-window is given a title – otherwise the effect can be quite confusing. Note also that the sub-windows must be opened by `SuperGUIRun` not `SuperGUIRunModal`.

```
doc⌣no = 0;
New⌣Document[] := Module[{fr, str},
    doc⌣no++;
    str = FromCharacterCode[Table[Random[Integer, {65, 90}], {10}]];
    SuperWidgetFrame[fr, {
        SuperWidgetTextEditor[str]
      }, Desktop → ddd, Title -> "Example " <> ToString[doc⌣no]] // SuperGUIRun
  ];


SuperGUIRunModal[SuperWidgetFrame[Null, {
    SuperWidgetDesktop[ddd, {
      Image⌣File[SuperWidgetPackagePath[] <> "\\ExampleFiles\\Consultancy.gif"]
     }, Background → RGBColor[1.0, 1.0, 0.7], Pixel⌣Width → 300, Pixel⌣Height → 200]
  }, Menu → {{"File", {{"New", New⌣Document}}}}]]
```



METYFAJUVE

In the above example, I ran the above code, stretched the top window a little, and activated the File/New menu item twice. Try the above example, and experiment with maximising the main window and/or one of the sub-windows.

Get‿Topmost‿MDI‿Frame[var]	If var is the control variable of a SuperWidgetDesktop, returns the control variable name (as a string) of the topmost child window. Otherwise returns an empty string

Functions to control the behaviour SuperWidgetDesktop

Dynamic manipulation of basic data input widget properties

The three super widgets – `SuperWidgetIntegerBox`, `SuperWidgetRealBox`, and `SuperWidgetStringBox` have a number of properties that are handled automatically or set at startup. In certain cases it may be useful to adjust these settings dynamically using the following functions:

Grab‿Focus[v]	Shifts the input focus to the widget associated with variable v.
Set‿Editable[v,val]	Sets the editability of the widget associated with variable v to the value val (True or False).
Last‿Focus‿Time[v]	Returns a representation in miliseconds of the time when the widget associated with variable v last acquired focus.
Select‿All[v]	Selects all the text in the widget associated with variable v – which it makes it easy for the user to type over the box.
Set‿Text‿Colour[v,colour]	Sets the text colour of the widget associated with variable v. Any *Mathematica* colour representation can be used.
Set‿Text‿Color[v,colour]	Sets the text colour of the widget associated with variable v. Any *Mathematica* colour representation can be used.
Set‿Background‿Colour[v,colour]	Sets the background colour of the widget associated with variable v. Any *Mathematica* colour representation can be used.
Set‿Background‿Color[v,colour]	Sets the background colour of the widget associated with variable v. Any *Mathematica* colour representation can be used.

Functions to control the behaviour of basic input widgets

Most of these functions are self explanatory, however the purpose of the function Last‿Focus‿Time may be less obvious. Suppose you had a form with a number of input fields and you wished to know which one had focus at a given moment – say from within a menu function. The problem is that the widget in question would momentarily lose focus when the menu was activated, so merely seeking a widget that has focus is not particularly useful. However, by comparing the last focus times of the various fields in question, it is possible to determine the box that has *effective* focus at a given instance.

Remembering what the user did last

Many GUI applications – such as *Mathematica* itself – remember various user-preferences both within a session and from one session to another. For example, the first time that a user tries to open a file with your program, he or she will probably have to navigate through the filestore to find the appropriate directory. This can be very tedious, and it is handy if he can start the next file open dialog at the place where the previous file was located. To remember this information, a persistence file name must be supplied:

```
Set⌣Application⌣Dump⌣File[file]
```

This should be called at the start of a program. It will load any information from the file if it exists, and remember the name of the file for subsequent operations. The string argument can be a complete path, or a simple file name, which will be located in `$UserBaseDirectory`.

One of the simplest uses of this feature is to add the option Use⌣Last⌣Directory->True to the `Open⌣File⌣Dialog` or `Save⌣File⌣Dialog` routines. In combination with the previous call, to set up the file to contain the information, this will make the application remember where the user saves his files, and will be a huge time saver.

Future versions of the SWP may define further properties that can be preserved across sessions, but you can also define properties of your own that operate in this way:

```
Add⌣Application⌣Variable[var]
```

This will associate the given variable with the persistence file already defined. To ensure that the latest values of all such variables are saved away before an application exits, you should call:

```
Save⌣Application⌣Dump⌣File[]
```

Be aware, however, that data may be saved at other times.

Accessing the Java layer

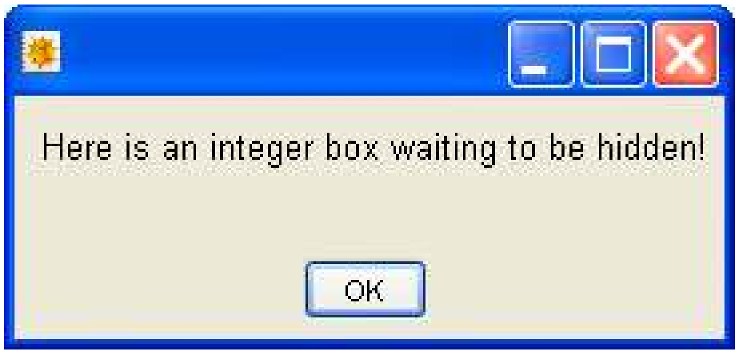
Although the SWP has been designed to hide J/Link, and Java layers that underpin it, it is sometimes useful – or simply interesting – to access the Java objects that implement the GUI. This can be achieved by passing any control variable corresponding to an active widget to the Java⌣Widget function.

To understand how to use the Java object, consult the documentation for the J/Link package (which is automatically loaded with the SWP).

For example, in the following code the Java object for the integer box is obtained and the setVisible method is invoked to hide the widget!

```
Needs["SuperWidgetPackage`"]


v = 42;
{¶10,
{¶10, "Here is an integer box waiting to be hidden!", ¶10},
{¶10, SuperWidgetIntegerBox[v], ¶10},
¶10,
{¶0, SuperWidgetButton[Null, "OK", 1], ¶0},
¶10
} // SuperGUIRun


- GUIObject -
```

```
Java⌣Widget[v]@setVisible[False]
```



Here is an integer box waiting to be hidden!

The integer box is only hidden, and can be made visible again by calling the setVisible method again with the argument True. The entier set of available methods can be obtained by using the J/Link function `Methods`:

```
Methods[Java⌣Widget[v]]
```

```
boolean action(java.awt.Event, Object)
void addActionListener(java.awt.event.ActionListener)
void addAncestorListener(javax.swing.event.AncestorListener)
void addCaretListener(javax.swing.event.CaretListener)
void addComponentListener(java.awt.event.ComponentListener)
void addContainerListener(java.awt.event.ContainerListener)
void addFocusListener(java.awt.event.FocusListener)
void addHierarchyBoundsListener(java.awt.event.HierarchyBoundsListener)
void addHierarchyListener(java.awt.event.HierarchyListener)
void addInputMethodListener(java.awt.event.InputMethodListener)
java.awt.Component add(java.awt.Component)
java.awt.Component add(java.awt.Component, int)
void add(java.awt.Component, Object)
void add(java.awt.Component, Object, int)
void add(java.awt.PopupMenu)
void addKeyListener(java.awt.event.KeyListener)
static javax.swing.text.Keymap addKeymap(String, javax.swing.text.Keymap)
void addMouseListener(java.awt.event.MouseListener)
void addMouseMotionListener(java.awt.event.MouseMotionListener)
void addMouseWheelListener(java.awt.event.MouseWheelListener)
void addNotify()
void addPropertyChangeListener(java.beans.PropertyChangeListener)
void addPropertyChangeListener(String, java.beans.PropertyChangeListener)
java.awt.Component add(String, java.awt.Component)
void addVetoableChangeListener(java.beans.VetoableChangeListener)
void applyComponentOrientation(java.awt.ComponentOrientation)
boolean areFocusTraversalKeysSet(int)
java.awt.Rectangle bounds()
int checkImage(java.awt.Image, int, int, java.awt.image.ImageObserver)
int checkImage(java.awt.Image, java.awt.image.ImageObserver)
void computeVisibleRect(java.awt.Rectangle)
boolean contains(int, int)
boolean contains(java.awt.Point)
void copy()
int countComponents()
java.awt.Image createImage(int, int)
java.awt.Image createImage(java.awt.image.ImageProducer)
```

```
javax.swing.JToolTip createToolTip()
java.awt.image.VolatileImage createVolatileImage(int, int)
java.awt.image.VolatileImage createVolatileImage(int, int, java.awt.ImageCapabilities)
void cut()
void deliverEvent(java.awt.Event)
void disable()
void dispatchEvent(java.awt.AWTEvent)
void doLayout()
void enable()
void enable(boolean)
void enableInputMethods(boolean)
boolean equals(Object)
java.awt.Component findComponentAt(int, int)
java.awt.Component findComponentAt(java.awt.Point)
void firePropertyChange(String, boolean, boolean)
void firePropertyChange(String, byte, byte)
void firePropertyChange(String, char, char)
void firePropertyChange(String, double, double)
void firePropertyChange(String, float, float)
void firePropertyChange(String, int, int)
void firePropertyChange(String, long, long)
void firePropertyChange(String, short, short)
javax.accessibility.AccessibleContext getAccessibleContext()
javax.swing.Action getAction()
java.awt.event.ActionListener getActionForKeyStroke(javax.swing.KeyStroke)
java.awt.event.ActionListener[] getActionListeners()
javax.swing.ActionMap getActionMap()
javax.swing.Action[] getActions()
float getAlignmentX()
float getAlignmentY()
javax.swing.event.AncestorListener[] getAncestorListeners()
boolean getAutoscrolls()
java.awt.Color getBackground()
javax.swing.border.Border getBorder()
java.awt.Rectangle getBounds()
java.awt.Rectangle getBounds(java.awt.Rectangle)
javax.swing.text.Caret getCaret()
java.awt.Color getCaretColor()
javax.swing.event.CaretListener[] getCaretListeners()
int getCaretPosition()
Class getClass()
Object getClientProperty(Object)
java.awt.image.ColorModel getColorModel()
int getColumns()
java.awt.Component getComponentAt(int, int)
java.awt.Component getComponentAt(java.awt.Point)
int getComponentCount()
java.awt.Component getComponent(int)
java.awt.event.ComponentListener[] getComponentListeners()
java.awt.ComponentOrientation getComponentOrientation()
java.awt.Component[] getComponents()
int getConditionForKeyStroke(javax.swing.KeyStroke)
java.awt.event.ContainerListener[] getContainerListeners()
java.awt.Cursor getCursor()
int getDebugGraphicsOptions()
```

```
static java.util.Locale getDefaultLocale()
java.awt.Color getDisabledTextColor()
javax.swing.text.Document getDocument()
boolean getDragEnabled()
java.awt.dnd.DropTarget getDropTarget()
char getFocusAccelerator()
java.awt.Container getFocusCycleRootAncestor()
java.awt.event.FocusListener[] getFocusListeners()
boolean getFocusTraversalKeysEnabled()
java.util.Set getFocusTraversalKeys(int)
java.awt.FocusTraversalPolicy getFocusTraversalPolicy()
java.awt.Font getFont()
java.awt.FontMetrics getFontMetrics(java.awt.Font)
java.awt.Color getForeground()
java.awt.Graphics getGraphics()
java.awt.GraphicsConfiguration getGraphicsConfiguration()
int getHeight()
java.awt.event.HierarchyBoundsListener[] getHierarchyBoundsListeners()
java.awt.event.HierarchyListener[] getHierarchyListeners()
javax.swing.text.Highlighter getHighlighter()
int getHorizontalAlignment()
javax.swing.BoundedRangeModel getHorizontalVisibility()
boolean getIgnoreRepaint()
java.awt.im.InputContext getInputContext()
javax.swing.InputMap getInputMap()
javax.swing.InputMap getInputMap(int)
java.awt.event.InputMethodListener[] getInputMethodListeners()
java.awt.im.InputMethodRequests getInputMethodRequests()
javax.swing.InputVerifier getInputVerifier()
java.awt.Insets getInsets()
java.awt.Insets getInsets(java.awt.Insets)
java.awt.event.KeyListener[] getKeyListeners()
javax.swing.text.Keymap getKeymap()
static javax.swing.text.Keymap getKeymap(String)
java.awt.LayoutManager getLayout()
java.util.EventListener[] getListeners(Class)
java.util.Locale getLocale()
java.awt.Point getLocation()
java.awt.Point getLocation(java.awt.Point)
java.awt.Point getLocationOnScreen()
java.awt.Insets getMargin()
java.awt.Dimension getMaximumSize()
java.awt.Dimension getMinimumSize()
java.awt.event.MouseListener[] getMouseListeners()
java.awt.event.MouseMotionListener[] getMouseMotionListeners()
java.awt.event.MouseWheelListener[] getMouseWheelListeners()
String getName()
javax.swing.text.NavigationFilter getNavigationFilter()
java.awt.Component getNextFocusableComponent()
java.awt.Container getParent()
java.awt.peer.ComponentPeer getPeer()
java.awt.Dimension getPreferredScrollableViewportSize()
java.awt.Dimension getPreferredSize()
java.beans.PropertyChangeListener[] getPropertyChangeListeners()
java.beans.PropertyChangeListener[] getPropertyChangeListeners(String)
```

```
javax.swing.KeyStroke[] getRegisteredKeyStrokes()
javax.swing.JRootPane getRootPane()
int getScrollableBlockIncrement(java.awt.Rectangle, int, int)
boolean getScrollableTracksViewportHeight()
boolean getScrollableTracksViewportWidth()
int getScrollableUnitIncrement(java.awt.Rectangle, int, int)
int getScrollOffset()
String getSelectedText()
java.awt.Color getSelectedTextColor()
java.awt.Color getSelectionColor()
int getSelectionEnd()
int getSelectionStart()
java.awt.Dimension getSize()
java.awt.Dimension getSize(java.awt.Dimension)
String getText()
String getText(int, int) throws javax.swing.text.BadLocationException
java.awt.Toolkit getToolkit()
java.awt.Point getToolTipLocation(java.awt.event.MouseEvent)
String getToolTipText()
String getToolTipText(java.awt.event.MouseEvent)
java.awt.Container getTopLevelAncestor()
javax.swing.TransferHandler getTransferHandler()
Object getTreeLock()
javax.swing.plaf.TextUI getUI()
String getUIClassID()
boolean getVerifyInputWhenFocusTarget()
java.beans.VetoableChangeListener[] getVetoableChangeListeners()
java.awt.Rectangle getVisibleRect()
int getWidth()
int getX()
int getY()
boolean gotFocus(java.awt.Event, Object)
void grabFocus()
boolean handleEvent(java.awt.Event)
boolean hasFocus()
int hashCode()
void hide()
boolean imageUpdate(java.awt.Image, int, int, int, int, int)
java.awt.Insets insets()
boolean inside(int, int)
void invalidate()
boolean isAncestorOf(java.awt.Component)
boolean isBackgroundSet()
boolean isCursorSet()
boolean isDisplayable()
boolean isDoubleBuffered()
boolean isEditable()
boolean isEnabled()
boolean isFocusable()
boolean isFocusCycleRoot()
boolean isFocusCycleRoot(java.awt.Container)
boolean isFocusOwner()
boolean isFocusTraversable()
boolean isFocusTraversalPolicySet()
boolean isFontSet()
boolean isForegroundSet()
boolean isLightweight()
static boolean isLightweightComponent(java.awt.Component)
boolean isManagingFocus()
boolean isMaximumSizeSet()
boolean isMinimumSizeSet()
boolean isOpaque()
boolean isOptimizedDrawingEnabled()
boolean isPaintingTile()
boolean isPreferredSizeSet()
boolean isRequestFocusEnabled()
boolean isShowing()
boolean isValid()
boolean isValidateRoot()
boolean isVisible()
boolean keyDown(java.awt.Event, int)
boolean keyUp(java.awt.Event, int)
void layout()
void list()
void list(java.io.PrintStream)
void list(java.io.PrintStream, int)
void list(java.io.PrintWriter)
void list(java.io.PrintWriter, int)
static void loadKeymap(javax.swing.text.Keymap, javax.swing.text.JTextComponent$KeyBin
java.awt.Component locate(int, int)
java.awt.Point location()
boolean lostFocus(java.awt.Event, Object)
java.awt.Dimension minimumSize()
java.awt.Rectangle modelToView(int) throws javax.swing.text.BadLocationException
boolean mouseDown(java.awt.Event, int, int)
boolean mouseDrag(java.awt.Event, int, int)
boolean mouseEnter(java.awt.Event, int, int)
boolean mouseExit(java.awt.Event, int, int)
boolean mouseMove(java.awt.Event, int, int)
boolean mouseUp(java.awt.Event, int, int)
void moveCaretPosition(int)
void move(int, int)
void nextFocus()
void notify()
void notifyAll()
void paintAll(java.awt.Graphics)
void paintComponents(java.awt.Graphics)
void paintImmediately(int, int, int, int)
void paintImmediately(java.awt.Rectangle)
void paint(java.awt.Graphics)
void paste()
void postActionEvent()
boolean postEvent(java.awt.Event)
java.awt.Dimension preferredSize()
boolean prepareImage(java.awt.Image, int, int, java.awt.image.ImageObserver)
boolean prepareImage(java.awt.Image, java.awt.image.ImageObserver)
void printAll(java.awt.Graphics)
void printComponents(java.awt.Graphics)
void print(java.awt.Graphics)
void putClientProperty(Object, Object)
void read(java.io.Reader, Object) throws java.io.IOException
void registerKeyboardAction(java.awt.event.ActionListener, javax.swing.KeyStroke, int)
void registerKeyboardAction(java.awt.event.ActionListener, String, javax.swing.KeyStro
void removeActionListener(java.awt.event.ActionListener)
void removeAll()
void removeAncestorListener(javax.swing.event.AncestorListener)
void removeCaretListener(javax.swing.event.CaretListener)
void removeComponentListener(java.awt.event.ComponentListener)
void removeContainerListener(java.awt.event.ContainerListener)
void removeFocusListener(java.awt.event.FocusListener)
void removeHierarchyBoundsListener(java.awt.event.HierarchyBoundsListener)
void removeHierarchyListener(java.awt.event.HierarchyListener)
void removeInputMethodListener(java.awt.event.InputMethodListener)
void remove(int)
void remove(java.awt.Component)
void remove(java.awt.MenuComponent)
void removeKeyListener(java.awt.event.KeyListener)
static javax.swing.text.Keymap removeKeymap(String)
void removeMouseListener(java.awt.event.MouseListener)
void removeMouseMotionListener(java.awt.event.MouseMotionListener)
void removeMouseWheelListener(java.awt.event.MouseWheelListener)
void removeNotify()
void removePropertyChangeListener(java.beans.PropertyChangeListener)
void removePropertyChangeListener(String, java.beans.PropertyChangeListener)
void removeVetoableChangeListener(java.beans.VetoableChangeListener)
void repaint()
void repaint(int, int, int, int)
void repaint(java.awt.Rectangle)
void repaint(long)
void repaint(long, int, int, int, int)
void replaceSelection(String)
boolean requestDefaultFocus()
void requestFocus()
boolean requestFocus(boolean)
boolean requestFocusInWindow()
void resetKeyboardActions()
void reshape(int, int, int, int)
void resize(int, int)
void resize(java.awt.Dimension)
void revalidate()
void scrollRectToVisible(java.awt.Rectangle)
void selectAll()
void select(int, int)
void setActionCommand(String)
void setAction(javax.swing.Action)
void setActionMap(javax.swing.ActionMap)
void setAlignmentX(float)
void setAlignmentY(float)
void setAutoscrolls(boolean)
void setBackground(java.awt.Color)
void setBorder(javax.swing.border.Border)
void setBounds(int, int, int, int)
void setBounds(java.awt.Rectangle)
void setCaretColor(java.awt.Color)
void setCaret(javax.swing.text.Caret)
void setCaretPosition(int)
void setColumns(int)
void setComponentOrientation(java.awt.ComponentOrientation)
void setCursor(java.awt.Cursor)
void setDebugGraphicsOptions(int)
static void setDefaultLocale(java.util.Locale)
void setDisabledTextColor(java.awt.Color)
void setDocument(javax.swing.text.Document)
void setDoubleBuffered(boolean)
void setDragEnabled(boolean)
void setDropTarget(java.awt.dnd.DropTarget)
void setEditable(boolean)
void setEnabled(boolean)
void setFocusable(boolean)
void setFocusAccelerator(char)
void setFocusCycleRoot(boolean)
void setFocusTraversalKeysEnabled(boolean)
void setFocusTraversalKeys(int, java.util.Set)
void setFocusTraversalPolicy(java.awt.FocusTraversalPolicy)
void setFont(java.awt.Font)
void setForeground(java.awt.Color)
void setHighlighter(javax.swing.text.Highlighter)
void setHorizontalAlignment(int)
void setIgnoreRepaint(boolean)
void setInputMap(int, javax.swing.InputMap)
void setInputVerifier(javax.swing.InputVerifier)
void setKeymap(javax.swing.text.Keymap)
void setLayout(java.awt.LayoutManager)
void setLocale(java.util.Locale)
void setLocation(int, int)
void setLocation(java.awt.Point)
void setMargin(java.awt.Insets)
void setMaximumSize(java.awt.Dimension)
void setMinimumSize(java.awt.Dimension)
void setName(String)
void setNavigationFilter(javax.swing.text.NavigationFilter)
void setNextFocusableComponent(java.awt.Component)
void setOpaque(boolean)
void setPreferredSize(java.awt.Dimension)
void setRequestFocusEnabled(boolean)
void setScrollOffset(int)
void setSelectedTextColor(java.awt.Color)
void setSelectionColor(java.awt.Color)
void setSelectionEnd(int)
void setSelectionStart(int)
void setSize(int, int)
void setSize(java.awt.Dimension)
void setText(String)
void setToolTipText(String)
void setTransferHandler(javax.swing.TransferHandler)
void setUI(javax.swing.plaf.TextUI)
void setVerifyInputWhenFocusTarget(boolean)
void setVisible(boolean)
void show()
void show(boolean)
```

```
java.awt.Dimension size()
String toString()
void transferFocus()
void transferFocusBackward()
void transferFocusDownCycle()
void transferFocusUpCycle()
void unregisterKeyboardAction(javax.swing.KeyStroke)
void update(java.awt.Graphics)
void updateUI()
void validate()
int viewToModel(java.awt.Point)
void wait(long, int) throws InterruptedException
void wait(long) throws InterruptedException
void wait() throws InterruptedException
void write(java.io.Writer) throws java.io.IOException
```

Needless to say, only a small proportion of these methods can be usefully called in this context!

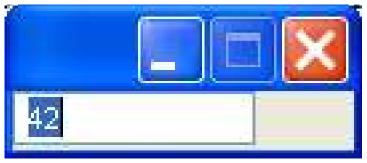
Snapshot mode

Most of the examples in this guide are illustrated with pictures of the resultant windows. The pictures were created by using snapshot mode. Calling `SetSnapshotMode[]` will display a small window containing a camera and a count of stored images (initially 0). Each time you click on the camera, an image of every open SWP window (except the camera!) is recorded. These images can be obtained as Graphics objects using `GetSnapshots[]` and displayed in the usual way using `Show`. Simply close the camera window when you are finished with it.

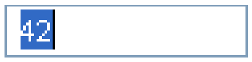
An older version of this mechanism which recorded windows as they were closed was discontinued at version 2.81 in favour of this new, more flexible scheme. For partial compatibility, a call to `SetSnapshotMode[True]` will execute in the same way as `SetSnapshotMode[]`.

As of version 4.16, there is also a function that will take a picture of any control or whole window specified by control variable:

```
x = 42;
SuperWidgetFrame[fff, {SuperWidgetIntegerBox[x]}] // SuperGUIRun

Snap⌣Component[fff] // Show
```


42

```
Snap⌣Component[x] // Show
```


42

Note that for this routine to work correctly, the window in question must be on screen, although it may be obscured by other windows. It will be brought to the front as part of this operation.

Additional functions

The following functions help in constructing your GUI interface.

`Set⌄Enabled⌄Status[var,True/False]` – This will enable/disable a control. You can also enable/disable menu items by passing the name of the corresponding function. By default, everything is enabled. This function can be called before the window is displayed, to set things up initially, or on the fly to change a setting. For example, it may make sense to start with a 'Save' option greyed out (disabled) until the user has done something that might need saving. It is always worth disabling features that are unusable in particular contexts – because it makes your GUI interface easier to use, and you don't have to worry about what might happen if your code is called when it does not make sense.

`Message⌄Beep[]` – This creates a beep sound via Java. Beeps are useful to remind the user that he has made an error, and are also often useful while debugging a GUI application. Later version of this function may take an argument to determine the type of noise produced.

`UpdateWidgetValue[var]` – The value of a widget – e.g. the number displayed in a `SuperWidgetReal` box – can be changed programmatically using this function. You alter the value of a super widget's associated variable as desired, and then call this function to make the change visible. By design, no `ChangeFunction` calls are made in response to this, because this could easily result in an infinite loop. Call any functions directly if necessary. This function has not (yet) been implemented for all types of Super Widgets, and obviously does not even make sense in all cases. It will fault where not available.

`GetURL[string]` – This is a tidied up version of the function defined in the J/Link help files. You give it a URL string and it copies the data from the internet to a temporary file and returns the name of the temporary file as its result (which you can then open with `OpenRead`). If this process fails for any reason, `$Failed` is returned. Because the internet is never 100% reliable, you should always test for the `$Failed` return value.

`HTTP⌄Post[address,{{name1,value1}…}]` – Performs an HTTP POST operation (equivalent to an HTML form) to the given URL. The second argument should be a list of 2-element sub-lists of the form {name,value}. For example, consider the following HTML form:

```
<html><head><title>My Form</title></head><body>
<form action='http://something.com/process.php' method='post'>
<input type='text' name='mydata' size='60'/>
<input type='submit'/>
</form>
</body>
</html>
```

This would display a text input box whose contents could be submitted to a website by pressing the 'submit' button. Say the text was 'Hello', the same operation could be achieved using the following call:

```
Needs["SuperWidgetPackage`"]

HTTP⌄Post["http://something.com/process.php", {{"mydata", "Hello"}}]
```

In a more complex HTML form with several sections, each section would have a different name – so in the corresponding call to `HTTP⌄Post`, the list of name/value pairs would contain several terms.

If this function succeeds, it returns whatever string of data is sent by the website. If it fails (which is always possible with operations involving the internet) it returns `$Failed` – so it is important to test for this value.

The following at example illustrates the use of this function to communicate with the SWP site.

`Color⌄Chooser⌄Dialog[]` – Displays a dialog to permit the user to select a colour. The value is returned as an `RGBColor` value.

`File⌣Open⌣Dialog[]` – Displays a dialog to permit the user to select an existing file. If the user selects a file, it is returned as a string (ready to be opened by OpedRead), `Null` is returned if no file is selected. This function can also take a string argument to label the dialog box.

`File⌣Save⌣Dialog[]` – Displays a dialog to permit the user to select a file to be written. If the user selects a file, it is returned as a string (ready to be opened by `OpenWrite`), `Null` is returned if no file is selected.

`Close⌣Frame[var,opts]` – Closes the window whose SuperWidgetFrame has the given associated variable, or the window that contains a widget controlled by variable var. Typically called in an 'Exit' menu. The option `Return⌣Value` can be used to specify an integer value to be returned by `SuperGUIRunModal`.

`Java⌣Console⌣Print[args]` – Prints its arguments on the Java console, creating the console if necessary. This can be useful to debug GUI applications, particularly those that use concealed notebooks or are stand-alone. Also, once the console has been created, any messages that are generated by Java code (e.g. calls to System.out.println) will also be displayed.

`Get⌣Screen⌣Size[]` – Returns the size in pixels of the screen as a 2-element list.

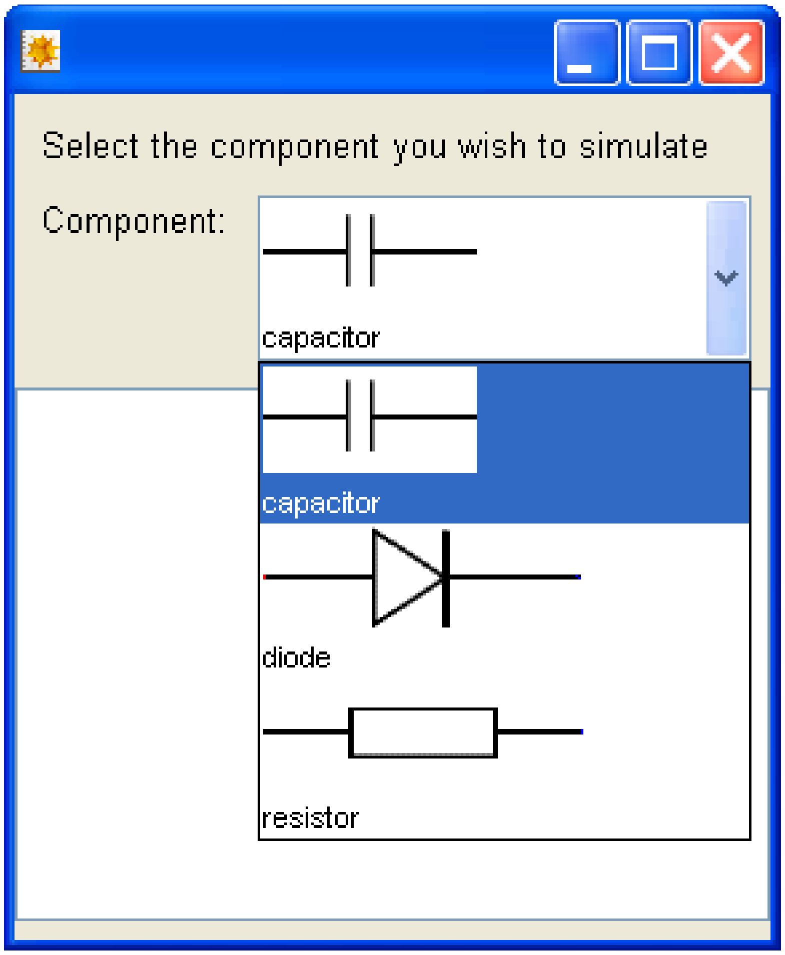
Using HTML inside super widgets

As you know, the SWP is based on Java to create the actual GUI. This means that you can exploit a very neat feature of the Java/Swing classes. If you put an HTML string into controls such as SuperWidgetButton, SuperWidgetComboBox, tool-tip options, etc. this will be displayed as HTML – not as a boring text string. This can be used to generate some amazing effects, including the use of images (although, it would seem animated GIF's sometimes only display their first frame), text in several fonts and/or special layout, mixed text and images, etc. Bear in mind that the images you use can, if you wish, have been created on the fly by your *Mathematica* code!

Here we put images and text in a combo box:

```
Needs["SuperWidgetPackage`"]

myFileName[z_, description_] := "<html><img src=\"file:" <> SuperWidgetPackagePath[] <>
   "/ExampleFiles/" <> z <> "\"><p>" <> description <> "</html>";
```

```
x = 1;
txt = "";
{
  ¶10,
  {¶10, "Select the component you wish to simulate", ¶10},
  ¶10,
  {¶10, "Component:   ", SuperWidgetComboBox[x,
    Map[myFileName[# <> ".gif", #] &, {"capacitor", "diode", "resistor"}]], ¶10},
  ¶10,
  SuperWidgetTextEditor[txt, Pixel⌄Height → 200],
  ¶10
 } // SuperGUIRunModal
```

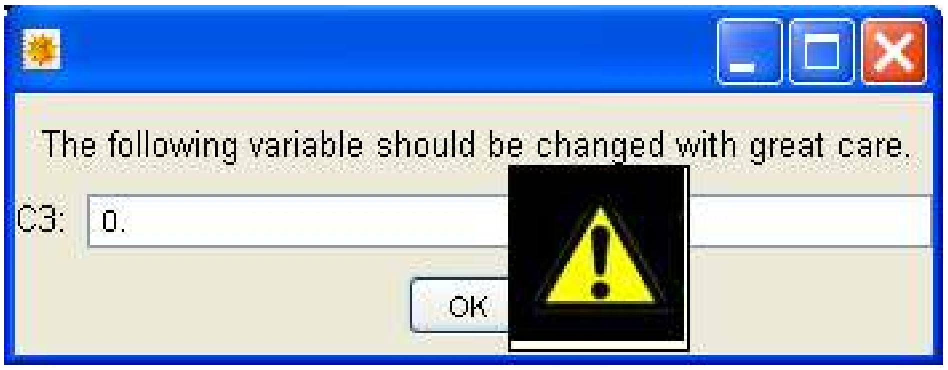


Note that it would be better to select images of the same size for this job!

Tip! In a practical application with a ChangeFunction it might be convenient to call `ComboBox⌄Index[x]` to extract the position of the selected item (1,2,etc.) rather than work with the HTML string.

Here we are using an image as a tool tip:

```
c3 = 1.0;
SuperWidgetFrame[Null, {¶10,
   {¶10, "The following variable should be changed with great care.", ¶10},
   ¶10,
   {"C3: ", SuperWidgetRealBox[c3, Tool⌣Tip -> "<html><img src=\"file:" <>
       SuperWidgetPackagePath[] <> "/ExampleFiles/warning.gif" <> "\"></html>"]},
   ¶10,
   {¶0, SuperWidgetButton[Null, "OK", 1], ¶0},
   ¶10
  }] // SuperGUIRunModal
```



HTML can also be used to create an implicit SuperWidgetLabel, as the label of a button, as the labels of a tree, and in many other places. However, not everywhere can process HTML – for example, the title of a frame or of a SuperWidgetLabelledBox will not accept HTML. You should test the you can use HTML in a particular way before relying on it. In some cases – such as tree widgets – Java does not seem to allow enough space to display the HTML. Also, since this is a property of the underlying Java implementation, there may be some variability between different platforms.

Tip! As you can see, HTML strings make some aspects of the SWP a little redundant. I must admit, I only discovered this feature of the Java swing classes fairly late in the development of the SWP. However, I suspect there may still be value in using features such as `Image⌣File` when they are sufficient rather than creating an HTML string. For simple tasks the old notation is much easier to use, and I also suspect there is additional overhead involved in loading and executing the general-purpose HTML layout code.

User-defined super widgets

Even using the SWP, definitions of large and complex windows can easily become messy and repetitious. One way to reduce the clutter is to define your own super widgets.

Define⌣Super⌣Widget[widget:>widgets]	Define a new super widget in terms of one or more built–in super widgets.

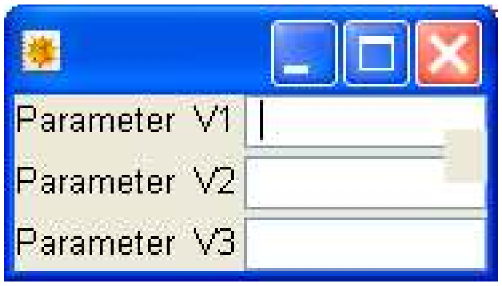
For example, consider the following definition:

```
Needs["SuperWidgetPackage`"]
```

```
SetAttributes[realPrompt, HoldFirst];
Define⌣Super⌣Widget[
 realPrompt[x_, p_String] :> {{"Parameter ", p, SuperWidgetRealBox[x]}}]
```

```
ReplaceAll::reps :
 {835., 320.5, 311.5} is neither a list of replacement rules nor a valid
   dispatch table, and so cannot be used for replacing. More…
```

```
{realPrompt[v1, "V1"], realPrompt[v2, "V2"], realPrompt[v3, "V3"]} // SuperGUIRunModal

GetSnapshots[] // Show;
```



The system will fault if you try to define a super widget which does not have attributes `HoldFirst` or `HoldAll`, or is otherwise malformed. Notice in particular that the definition uses :> (`RuleDelayed`), so that the right hand side of a definition can contain arbitrary chunks of *Mathematica* code to determine what finally gets displayed. The only requirement is that the end result is a widget, or list of widgets and layout.

As you know, a simple list of super widgets is displayed vertically, nested lists, horizontally, etc. Within a user-defined super widget, the top level list is displayed vertically, etc., regardless of the nesting of the user defined super widget within the window as a whole. Thus a user defined super widget can be thought of as a self-contained widget in its own right.

The first argument to a user-defined super widget must be an associated variable (or `Null`), just as with the built-in super widgets, however a user-defined widget may have attribute `HoldAll` – which enables it to pass on additional arguments as control variables to the various super widgets into which it resolves.

Concealing notebooks

If you have being trying the examples as you read this user guide, you may already have encountered the problem that Java windows can easily be obscured by a *Mathematica* notebook. The end user (who may be fairly naive) may start a GUI application and then click on the notebook. If the application opens with a modal window, this may give the impression that *Mathematica* has hung, because the only thing that will accept input is out of sight!

This is a problem which is inherited from J/Link. It is caused by the fact that the Java windows are run as a separate process (connected by *MathLink*) – so the operating system treats the *Mathematica* FrontEnd (which is displaying the notebooks) and the Java windows as two completely independent applications that can obscure each other on the screen.

The problem is only really relevant to modal windows, since modeless windows are meant to be susceptible to being overlaid by other things (c.f. palettes).

One partial solution to this problem is to use the option `ConcealNotebooks->True` on `SuperGUIRunModal`. This option will hide all open notebooks while the modal window is displayed, and make them visible when the window finally closes. It should be used on the main window of an application. If this option is used, it is suggested that it be used on complete and tested applications. This is because if the application aborts in some way, you may be left with your notebook hidden – possibly with unsaved changes.

Another solution to the problem of notebooks obscuring Java windows is to set up a stand-alone program.

Stand alone programs

A stand-alone program executes using the MathKernel program – without displaying the frontend or any notebooks. The idea is that the entire user interaction is via Java windows. In this way, it is possible to create applications that have all the 'look and feel' of conventional applications that do not use *Mathematica*.

Although certain simple stand-alone programs were possible prior to version 4.00, various problems with GUIKit prevented the serious exploitation of this feature. One appeal of a stand alone program is that the end user need not be distracted by the presence of *Mathematica* (and there is no problem of a notebook obscuring the Java window). Indeed, the end user might not even be aware that *Mathematica* was being used in the calculation at all.

It is possible to start a kernel-only *Mathematica* session that reads and executes a .m file (not a notebook) and never displays a window of any kind. Thus, if the .m file contains code to display a super widget this will operate in a totally uncluttered fashion. Furthermore, the program can be developed and debugged from within a *Mathematica* notebook, and written to the .m file using the `AutoGeneratedPackage` facility.

"c:\program files\wolfram research\mathematica\5.1\Mathkernel" -mathlink -initfile test1.m

The -mathlink option suppresses even the kernel window from showing, so you may wish to omit this option while testing.

If you anticipate running your program in this way, it is important not to use any features that require the FrontEnd.

See the next section for a more convenient way to start applications using the kernel.

Creating programs that start when you double-click their data files

Although it is still possible to start the MathKernel directly to run such programs, as of version 6.24, there is a much more elegant option. Think for a moment of a typical Windows application, such as Word for Windows (or Libre Office, if you prefer). Such applications can be started by double-clicking on one of their data files (on the desktop, in Explorer, or in certain other contexts). This feature is actually very useful, and using the SuperWidgetPackage, this is possible for *Mathematica* applications.

To use this feature, you first need to select an unused file suffix to refer to your kind of data. Take great care to avoid commony used suffices, particularly those in use on your machine. It may be worth GOOGLEing your suffix to make sure it is not already in use. There is no restriction to 3-character suffixes, and longer ones are more likely to be unique. As an example, suppose your program dealt with astronomical data, you might decide to let your data files end with the suffix .STAR.

Clearly, when you start your *Mathematica* application from an icon, neither *Mathematica*, nor your application will normally be active, so, just as with other Windows applications, this information has to be enetered ahead of time, and is then stored in the Windows registry for future use. To perform this task, execute the following function call:

```
Install⌣MathLauncher⌣Application["c:\\prog\\StarCalculator.m",
  ".star", "StellarDataFile",
  "Stellar data file",
  "c:\\\Data\\stellar_data.ico",
  "SplashScreen" -> "c:\\Data\\SplashStarCalculator.jpg",
  "UseFrontEnd" → False];
```

• This function only needs to be executed once to make the necessary file associations, but it can be executed more than once, without causing any problems, so you may want to simply execute it each time the application is run.

• An application can be associated with more than one kind of data – simply call this function once for each file suffix you need to register.

• The data within the file can be text or binary, as required.

• Setting "UseFrontEnd" → True, will, of course, open the application in the front end. This is not normally useful for finished applications, but it may be useful for development, or for other applications.

• The splash screen is vital because your user needs this for visual feedback that his data is being opened (otherwise, he may click again). It normally appears very quickly, before the actual loading of *Mathematica,* which may impose a delay before your program can start working. If you set this argument to Null, the SWP will use an amusing image instead!

• Argument 4 is used as a tooltip if the user hovers over the icon for a .STAR data file.

• When your application is started from a data file, the full pathname of that file will be placed in the Global variable, MathLauncherSuppliedOpen. If ValueQ[MathLauncherSuppliedOpen] returns False, the application was started conventionaly. The path will include the file suffix, so this can be tested if the application can accept several types of data:

```
If[ToUpperCase[FileExtension[MathLauncherSuppliedOpen]] == ".STAR", doSomething[]];
```

• Note that you can also create an association to an extra data type – say .STARAPP – which can be used to create an icon on the desktop that just starts your application with no actual data (the data file can contain anything). In this case, your application will examine the data file name, and determine that the file extension is .STARAPP, and not open it. You should use an icon for such a file type that suggests your application as a whole.

• Some windows applications, such as word processors, allow more than one data file to be open at once. If your application is complex enough for this to be useful to you, and you are using the SWP for your GUI, you can proceed as follows:

Supply a definition for the function SuperWidgetPackage`MathLauncher⌣Open[file_] , and supply a file name to the following function:

```
Set⌣Command⌣File[commandFile]
```

The file will not normally exist, but should be writable – it is required internally by the SWP. Using this feature, subsequent files that are opened while your application is active, will result in a call to your supplied open function <u>at a time when your GUI is otherwise idle</u>.

Utility functions

`Close⌣Frame[v]`	Closes the window whose frame (or dialog window) is associated with v or which contains a super widget associated with v.

`Color⌣Chooser⌣Dialog[]`	Displays a dialog to select a colour. Returns an RGBColor value, or Null if nothing was selected.
`Colour⌣Chooser⌣Dialog[]`	Synonym for Color⌣Chooser⌣Dialog.
`Define⌣Super⌣Widget[ widget:→widgets]`	Define a new super widget (in terms of one or more built−in super widgets or *GUIKit* widgets).
`File⌣Open⌣Dialog[]`	Displays a file−open dialog, and returns the file selected (as a string) or Null.
`File⌣Open⌣Dialog[str]`	Displays a file−open dialog labelled by the specified string, and returns the file selected (as a string) or Null.
`File⌣Save⌣Dialog[]`	Displays a file−save dialog, and returns the file selected (as a string) or Null.
`File⌣Save⌣Dialog[str]`	Displays a file−save dialog labelled by the specified string, and returns the file selected (as a string) or Null.
Get⌣Screen⌣Size`[]`	Returns the width and height of the screen in pixels as a 2−element list.
`GetSnapshots[]`	Returns a list of snapshots recorded since SetSnapshotMode[True] was called.
`GetURL[url-name]`	Copies the data at the given URL into a temporary file, which is returned as the result. If the operation fails for any reason, $Failed is returned.
`HTTP⌣Post[address, {{name1,value1}…}]`	Performs an HTTP POST operation (equivalent to an HTML form) to the given URL. The second argument should be a list of 2−element sub− lists of the form {name,value}. If the operation fails $Failed is returned, otherwise the response by the website is returned as a string.
`Image⌣Boxes[boxes]`	This does not evaluate directly, but is used to wrap boxes (as made by ToBoxes) being sent to super widgets that can take images.
`Image⌣Expression[expr,form]`	This does not evaluate directly, but is used to wrap an expression (such as Sqrt[X]) to be converted to an image in the specified form (default StandardForm) and sent to super widgets that can take images.
`Image⌣File[v,opts]`	This does not evaluate directly, but is used to wrap the path−name of a file containing an image to be sent to super widgets that can take images.
`Image⌣String[v,opts]`	This does not evaluate directly, but is used to wrap strings (as made by ExportString) being sent to super widgets that can take images.
`Interval⌣Timer[secs,func]`	Interval⌣Timer[secs,func] –Initiates a one−shot timer that calls func[] after the given number of seconds (which need not be integer). Note that the function can call Interval⌣Timer again to create a (safe) repeating timer.

`Java⌣Console⌣Print[args]`	Prints arguments on the Java console – useful to debug GUI applications that use concealed notebooks or are stand−alone.
`Java⌣Widget[v]`	Returns the Java object (ready for use with J/Link) corresponding to the widget with control variable, v.
`LiveGraphics3DApplet[v]`	If v is the associated variable for a LiveGraphics3D super widget, this will return the raw applet so that J/Link calls can be made to it.
`Message⌣Beep[]`	Generates a beep via Java.
`Mouse⌣Button⌣Info[v]`	Returns additional information about a mouse action performed in a SuperWidgetGraphicsPanel. The variable `v` should be the control variable for the whole SuperWidgetGraphicsPanel for mouse moves (which are handled on a per− panel basis) or the control variable of the relevant Graphics⌣Region. This function is not relevant in the case of drag operations. The information is returned as a list.
`Mouse⌣Position[v]`	Returns the position of the mouse in the Graphics⌣Region with control variable v. The coordinates are in the coordinate system of the whole graphics area, but if the mouse is not within the given graphics area, this function returns {Intermediate,Intermediate}.
`Open⌣SuperWidget⌣Log[]`	Opens an extra notebook to contain assorted logging information associated with the SuperWidgetPackage – mainly intended for internal debugging of the SWP.
`SetSnapshotMode[mode]`	Argument mode can be True or False. Sets a mode to tell the system to record a snapshot of a window whenever it is closed. These snapshots (Graphics objects) are returned as a list by calling GetSnapshots[]. This function can be used before or during the time that a window is displayed.
`Set⌣Wizard⌣Button⌣State[v, button-name,page-no,True/False]`	Enables/disables the given button on a particular page of a wizard that has been created with control variable v. The button names are strings and are case−insensitive.
`Set⌣Enabled⌣Status[ v,True/False]`	Enables/disables the super widget with associated variable v. This can be called before the relevant super widget has been created, in which case it sets its initial state, or it may be called while the super widget is live, to change its state.
`Set⌣Label⌣Contents[v,value]`	Changes the value of the label with control variable v to the given value, which may be text or image. However, this function cannot convert a text label into an image label or vice−versa.
`Set⌣Mouse⌣Mode[v,gv,mode]`	Changes the Mouse⌣Mode setting for the Graphics⌣Region with control variable gv associated with the SuperWidgetGraphicsPanel with control variable v.
`Set⌣Variable⌣Options[v,opts]`	This can be used to associate options with the variable v. These options are used when a super widget is subsequently created with v as its associated variable. This feature can help to avoid clutter in super widget declarations.

`ShowMessageBox[message,title,button-list]`	This is a convenience function to display a text message in a modal dialog box with a title and a number of buttons supplied as a list of strings. The box closes when a button is pressed, and it returns the number of the button pressed.
`Using⌄PlayerPro[]`	Returns True if the program is running under *Mathematica* PlayerPro. This function can be useful to avoid executing code that will not work under PlayerPro (such as using Get on unencrypted files).
`Kernel⌄Only⌄Mode[]`	Returns True if the program was started from MathKernel.exe rather than from the frontend. By starting a program from MathKernel, it is possible to create a GUI application in which only Java windows are visible. This is ideal for applications that will be used by people uninterested in *Mathematica* as such.
`SuperWidgetPackagePath[]`	Returns the full path name of the directory containing the SuperWidgetPackage. This is mainly useful to calculate the path of the various files contained in the ExampleFiles directory.
`UpdateWidgetValue[v,opts]`	If v is associated with a super widget that displays its value, the widget will be updated to reflect any change in the value of v performed by the program. The ChangeFunction is not called in this situation to avoid a possible infinite loop. Use the option Use ReCalibrate−> True to force the re−calibration of a SuperWidgetGraphicsPanel.
`Update⌄Graphics[v,graphics,opts]`	This function is now obsolete

Larger examples

The examples in this guide have been mostly very small. A collection of rather larger examples is available at http://www.d-baileyconsultancy.co.uk/swp_examples/swp_examples.html

If you feel you have constructed an interesting GUI application using the Super Widget Package, and would like to contribute it to this collection, please contact me.