Basic SWT Widgets [1, 2]

by Shantha Ramachandran

Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada

Last revised: June 4, 2003

Overview:

In this document, we describe the basic SWT (Standard Widget Toolkit) widgets. Our focus is on developing Java applications that use the SWT, not on developing plugins to extend the Eclipse workbench. If you have not already run an SWT-based application using the Eclipse workbench, you should review the information on installing Eclipse and setting up the SWT libraries in the previous document, *Installing Eclipse*.

Widget Structure:

To create an application that uses SWT widgets, you need to start by importing the following packages:

```
org.eclipse.swt.*;
org.eclipse.swt.widgets.*;
```

It is necessary to create a Display and a Shell to hold the SWT widgets:

```
Display display = new Display();
Shell shell = new Shell(display);
```

A display is an object that contains all GUI components. A display is not actually visible but the components added to a display are visible. Typically, only one display is created for an application. A shell is a window within the application. There may be any number of shells created for an application, with shells being at the top level (attached to a display) or shells being attached to other shells.

[1] This work was funded by an IBM Eclipse Innovation Grant.

[2] © Shantha Ramachandran and David Scuse

The properties of the SWT widgets are initialized and/or modified in the same manner as Swing widgets. You can set the size of the shell:

```
shell.setSize(100,100);
```

Finally, you need to open the shell and run an event loop. This event loop is required for the shell to visible on the screen. You can place it after any code you have initializing widgets on the shell. When the shell is closed, the display must be disposed.

```
shell.open();
while(!shell.isDisposed()){
      if(!display.readAndDispatch())
            display.sleep();
      }
display.dispose();
```

Now, you are ready to add any widgets to your shell. We will not use a layout manager for now, as creating widgets with a layout manager is a little more complicated. The following examples all deal with placing widgets on a shell with no layout.

Label

A label is a collection of characters that can not be modified by the user (although the characters can be modified by the programmer).

To place a label on your shell, use the following code:

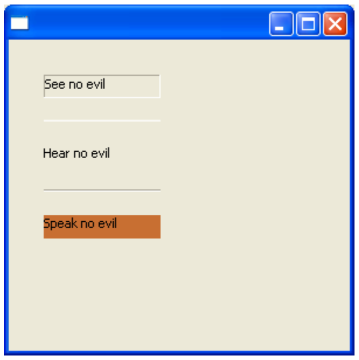
```
Label label1 = new Label(shell, SWT.BORDER);
```

The different styles of labels are BORDER, CENTER, LEFT, RIGHT, WRAP and SEPARATOR. The separator style is a special kind of label that draws a line separating your other widgets. The styles that can be added for a separator are HORIZONTAL, VERTICAL, SHADOW_IN, SHADOW_OUT and SHADOW_NONE.

The following code creates five labels, two of which are separators. The first text label has a border, the second is a normal label, and the third has a colored background that was set using the setBackground() method. The two separators in the example are horizontal, but the first has a shadow style applied to it.

```
Label label1 = new Label(shell, SWT.BORDER);
label1.setText("See no evil");
label1.setSize(100,20);
label1.setLocation(30,30);

Label sep1 = new
      Label(shell, SWT.SEPARATOR| SWT.HORIZONTAL |
                                       SWT.SHADOW_IN);
sep1.setBounds(30,60,100,20);

Label label2 = new Label(shell, SWT.NONE);
label2.setText("Hear no evil");
label2.setSize(100,20);
label2.setLocation(30,90);

Label sep2 = new Label(shell, SWT.SEPARATOR | SWT.HORIZONTAL);
sep2.setBounds(30,120,100,20);

Label label3 = new Label(shell, SWT.NONE);
label3.setSize(100,20);
label3.setLocation(30,150);
label3.setBackground(new Color(display,200,111,50));
label3.setText("Speak no evil");
```

This results in the following window:



Text

A text widget contains text that can normally be modified by the user.

To place a text widget on your screen, use the following code:

```
Text text1 = new Text(shell, SWT.BORDER);
```

The styles that are supported by Text are BORDER, H_SCROLL, V_SCROLL, MULTI, SINGLE, READ_ONLY and WRAP.

The text widget has some interesting properties that can be set. If your text is editable, you can set the maximum number of characters that can be typed:

```
text1.setTextLimit(30);
```

You can also create password text boxes by using the setEchoChar() method. The argument for this method is the character that is to be displayed in the text. The correct string will still be in the text, but will only be viewed as a sequence of a specified character:

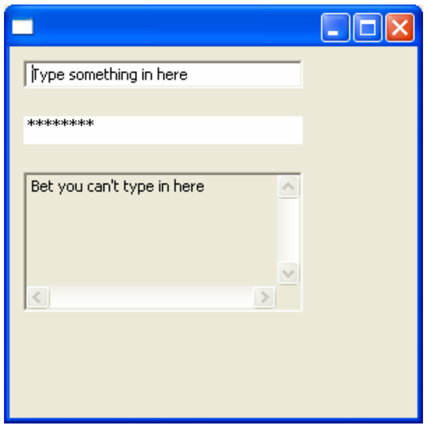
```
text2.setEchoChar('*');
```

The following code creates three text widgets. The first one has a 30 character text limit. The second one displays all text as a series of asterisks, much like a password field. The final text has used the setEditable() method to make the control read only. Another way to accomplish this would be to give it the style SWT.READ_ONLY.

Try running the following code and typing in the text widgets:

```
Text text1 = new Text(shell, SWT.BORDER);
text1.setText("Type something in here");
text1.setBounds(10,10,200,20);
text1.setTextLimit(30);

Text text2 = new Text(shell, SWT.NONE);
text2.setEchoChar('*');
text2.setBounds(10,50,200,20);
text2.setText("Password");

Text text3 = new Text(shell, SWT.BORDER | SWT.H_SCROLL |
                                                    SWT.V_SCROLL);
text3.setBounds(10,90,200,100);
text3.setEditable(false);
text3.setText("Bet you can't type in here");
```

This produces the following window:



Button

A button is a widget that is clicked by the user in order to initiate some processing.

To place a button on the shell, use the following code:

```
Button button1 = new Button(shell,SWT.PUSH);
```

You can replace SWT.PUSH with any of the button types that SWT supports. This includes PUSH, CHECK, RADIO, TOGGLE and ARROW. You can also use the styles FLAT, BORDER, LEFT, RIGHT and CENTER with one of these types.

To set the text of the button, use the setText() method:

```
button1.setText("Hello");
```

In order for the button to be visible on the screen, you need to set the size and the location of the button. Alternatively, you can set the bounds of the button and both the size and the location will be set at once.

```
button1.setLocation(0,0);
button1.setSize(100,20);
```

You can also set the style, background, image and more by using set methods. To add an event handler to your button, you first need to import the events package:

```
import org.eclipse.swt.events.*;
```

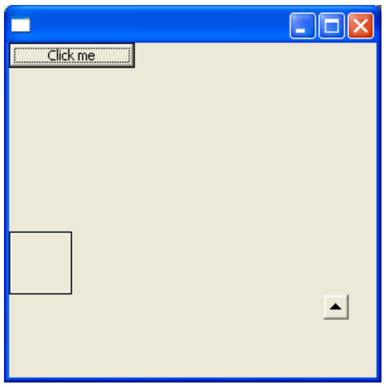
A basic event handler for the button is a selection event handler. This is called when the button is selected by clicking on it. To create the event handler, we add a listener to the button using the following code:

```
button1.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent e) {
            System.out.println("Button1 was clicked");
      }
});
```

Here is a small sample that incorporates all the code seen above. It creates a button which, when clicked, will print “Button was clicked” to the console, an arrow button, and a button with more than one style - toggle and flat.

```
Button button1 = new Button(shell,SWT.PUSH);
button1.setText("Click me");
button1.setLocation(0,0);
button1.setSize(100,20);
button1.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent e) {
            System.out.println("Button1 was clicked");
      }
});
Button button2 = new Button(shell, SWT.ARROW);
button2.setSize(20,20);
button2.setLocation(250,200);

Button button3 = new Button(shell, SWT.FLAT|SWT.TOGGLE);
button3.setSize(50,50);
button3.setLocation(0,150);
```

The following window is displayed:



List

A list is a widget that contains a collection of items; the user may select an item from the list.

To place a list on your screen, use the following code:

```
List list1 = new List(shell, SWT.MULTI);
```

The different styles that a list supports are BORDER, H_SCROLL, V_SCROLL, SINGLE and MULTI.

To add items to a list, you can use the setItems() method, or the add() method.

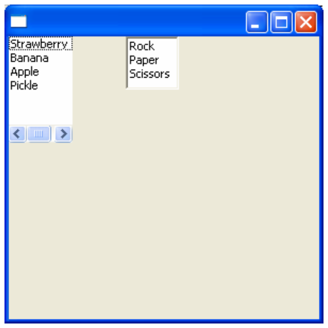
In the following example, two lists are created. The first list uses the H_SCROLL style to create a horizontal scrollbar within the list. The second one uses a MouseListener to respond to MouseDown and MouseUp events. The MouseListener works in a similar fashion to the SelectionListener. A MouseAdapter is created with one of or both of mouseUp() and mouseDown() methods. This is the code that is necessary:

```
list2.addMouseListener(new MouseAdapter() {
      public void mouseDown(MouseEvent e) {
            System.out.println(list2.getSelection()[0] +" wins");
      }
      public void mouseUp(MouseEvent e) {
            System.out.println("Try again!");
      }
});
```

Note that list2 must be declared as final in order to use it within the adapter:

```
List list1 = new List(shell, SWT.MULTI | SWT.H_SCROLL);
list1.setItems(new String[] {"Strawberry","Banana","Apple"});
list1.add("Pickle");
list1.setBounds(0,0,60,100);

final List list2 = new List(shell, SWT.SINGLE | SWT.BORDER);
list2.setItems(new String[] {"Rock","Paper","Scissors"});
list2.setBounds(110,0,50,50);
list2.addMouseListener(new MouseAdapter() {
      public void mouseDown(MouseEvent e) {
            System.out.println(list2.getSelection()[0] +" wins");
      }
      public void mouseUp(MouseEvent e) {
            System.out.println("Try again!");
      }
});
```

The following window is displayed:



Combo

A combo widget allows the user to select an item from a collection of items. The user may also type a value into the combo widget.

To place a combo box on the shell, use the following code:

```
Combo combo1 = new Combo(shell,SWT.DROP_DOWN | SWT.READ_ONLY);
```

The styles that are supported by Combo are BORDER, DROP_DOWN, READ_ONLY and SIMPLE. More than one style can be used at a time, as in the above code.

To add items to a Combo, use the setItems() method. This takes a string array as an argument.

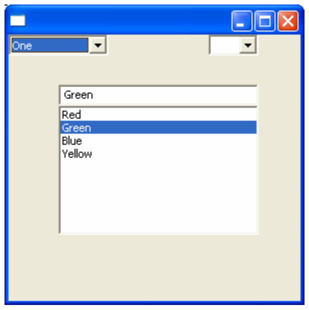
```
combo1.setItems(new String[] {"One","Two","Three"});
```

Use the select() method to select one of the list items. This method takes an integer argument - the index of the item to be selected.

The following example creates three Combos. The first is a drop down list whose location and size are set. The second is a simple Combo whose location and size are set at one time using the setBounds() method. The third is a Combo with no items.

```
Combo combo1 = new Combo(shell, SWT.DROP_DOWN|SWT.READ_ONLY);
combo1.setItems(new String[] {"One","Two","Three"});
combo1.select(0);
combo1.setLocation(0,0);
combo1.setSize(100,20);
Combo combo2 = new Combo(shell, SWT.SIMPLE);
combo2.setItems(new String[] {"Red","Green","Blue","Yellow"});
combo2.setBounds(50,50,200,150);
combo2.select(1);
Combo combo3 = new Combo(shell, SWT.DROP_DOWN);
combo3.setLocation(200,0);
combo3.setSize(50,50);
```

The following is window is displayed:



Composite

A composite is a widget that can contain other widgets. Widgets are placed inside a composite in the same manner that widgets are placed on a shell. The position of each widget inside a composite is relative to the composite, so if the composite is moved on the shell, the widgets inside the composite retain their relative positions.

To place a composite on your shell, use the following code:

```
Composite composite1 = new Composite(shell,SWT.BORDER);
```

The styles supported by composites are BORDER, H_SCROLL and V_SCROLL.

You can set properties of the composite, like background colour, but you cannot set a text value.

```
Label label = new Label(composite1,SWT.NONE);
```

The following code is an example of placing a composite within a composite. The first composite has a border and a coloured background, so it is easily spotted on the screen. In order to use the setBackground() method for any control, you need to import the graphics library and create a color:

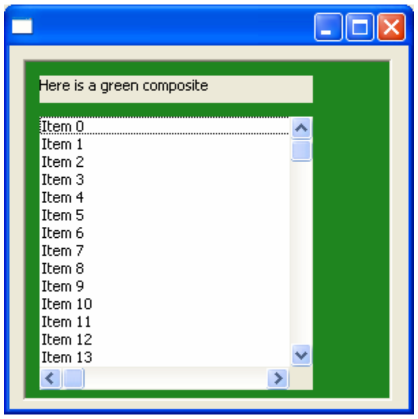
```
import org.eclipse.swt.graphics.*;
Color myColor = new Color(display, 0, 0, 0);
```

Note that if you do not give a composite a border or a background, you will not be able to distinguish it from the rest of the shell. The second composite exists within the first. It has horizontal and vertical scroll bars, and a list within the composite:

```
Composite composite1 = new Composite(shell,SWT.BORDER);
composite1.setBounds(10,10,270,250);
composite1.setBackground(new Color(display,31,133,31));
Label label = new Label(composite1,SWT.NONE);
```

```
label.setText("Here is a green composite");
label.setBounds(10,10,200,20);
Composite composite2 = new
      Composite(composite1,SWT.H_SCROLL|SWT.V_SCROLL);
composite2.setBounds(10,40,200,200);

List list = new List(composite2,SWT.MULTI);
for (int i=0; i<50; i++) {
      list.add("Item " + i);
}
list.setSize(300,300);
```

This produces the following screen:


Here is a green composite

Item 0
Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10
Item 11
Item 12
Item 13

Group

A group is also a widget that can contain other widgets. A group is surrounded by a border and may, optionally, contain a title. As with composites, the position of each widget inside a group is relative to the group, so if the group is moved on the shell, the widgets inside the group retain their relative positions.

To add a group to your screen, use the following code:

```
Group group1 = new Group(shell, SWT.BORDER);
```

The styles that are supported by group are BORDER, SHADOW_ETCHED_IN, SHADOW_ETCHED_OUT, SHADOW_IN, SHADOW_OUT and SHADOW_NONE.

You can set all the standard properties for a group, such as size, location, background, and more. However, the principal use of a group is to separate other controls into sections. A group is a subclass of composite, and works in very much the same way. It has additional functionality though, such as setting the text:

```
group1.setText("Group 1");
```

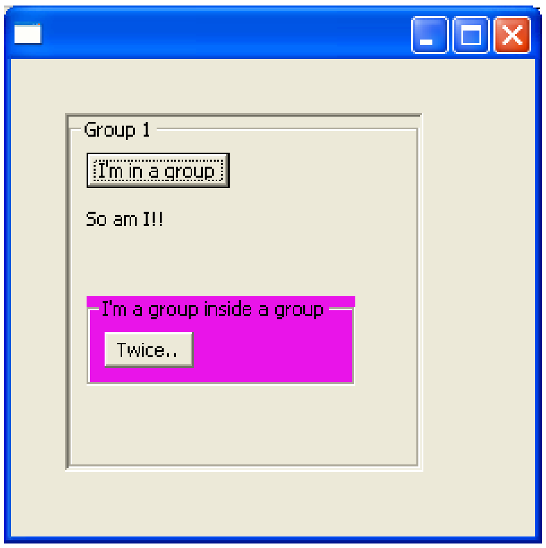
You can place a control onto a group in the same way you would add a control to a shell or a composite:

```
Button button = new Button(group1, SWT.PUSH);
```

The following example only uses one group on the main shell, but has a second group embedded within the first group. Since a group is a control as well as a composite, this is allowable. You can have many nested groups to arrange numerous controls on a screen:

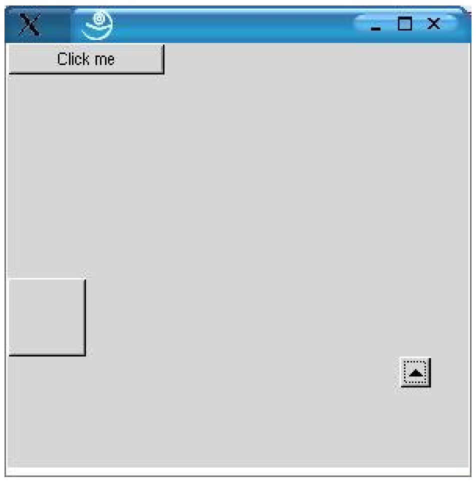
```
Group group1 = new Group(shell, SWT.BORDER);
group1.setBounds(30,30,200,200);
group1.setText("Group 1");

Button button = new Button(group1, SWT.PUSH);
button.setBounds(10,20,80,20);
button.setText("I'm in a group");

Label label = new Label(group1, SWT.NONE);
label.setBounds(10,50,80,20);
label.setText("So am I!!");

Group group2 = new Group (group1, SWT.NONE);
group2.setBounds(10,100,150,50);
group2.setBackground(new Color(display,233,20,233));
group2.setText("I'm a group inside a group");

Button button2 = new Button(group2, SWT.PUSH);
button2.setBounds(10,20,50,20);
button2.setText("Twice..");
```

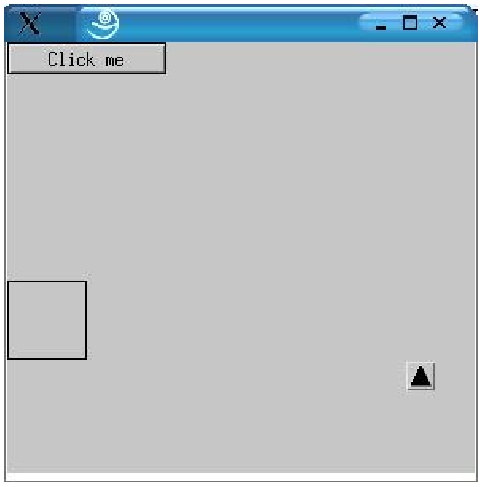
This results in the following window:



Running under Linux:

The widgets shown in this document were also run under Linux (GTK and Motif). Other than some slight differences in appearance, we have not found any differences in behaviour.





GTK **Motif**

Events:

In the earlier sections of this document, we introduced the Eclipse event structure by defining listeners for buttons and a few other simple events. We now describe the Eclipse event structure in more detail and create listeners for a variety of common events.

To capture events in SWT, you need to add listeners to your controls. Then when the event corresponding to the listener occurs, the listener code will be executed.

In order to capture events in SWT, you need to do two things. First, you create a listener specific to the event that you wish to capture. There is an interface for each listener that you can use (ie. SelectionListener). There is also a class that will provide you with the event information (ie. SelectionEvent). Within the listener that you create, you must implement the methods defined in the interface.

```
SelectionListener listener = new SelectionListener() {
      public void widgetSelected(SelectionEvent arg0) {
            System.out.println("Button Selected");
      }
      public void widgetDefaultSelected(SelectionEvent arg0) {
      }
};
```

If there is more than one method, you can create an adapter instead (ie. SelectionAdapter). An adapter implements the interface with empty methods so that you do not have to define code for methods that you do not need in your listener. Note that in our SelectionListener, we did not add any code for the widgetDefaultSelected() method. In this case, we can create an adapter instead, and only implement the widgetSelected() method.

```
SelectionAdapter adapter = new SelectionAdapter() {
      public void widgetSelected(SelectionEvent arg0) {
            System.out.println("Button Selected");
      }
};
```

The second thing you need to do is add the listener to your widget. Each control has methods to add listeners (ie. addSelectionListener()). You pass your listener to this method, and you are all set to capture events.

```
Button button = new Button(shell, SWT.PUSH);
button.addSelectionListener(listener);
```

Or, in our case, we could have also used:

```
button.addSelectionListener(adapter);
```

Now that we know how to add a listener to a widget, we will go over the more common types of listeners. The most common type of listener is the selection listener, which we have just seen. Some other important ones include key listeners and mouse listeners.

Key Listener

The KeyListener has two methods, keyPressed and keyReleased, whose functions are obvious. The event object is KeyEvent. Because there are two methods, there is also a KeyAdapter you can use to implement only one of the methods.

The KeyEvent is the more interesting feature of capturing key events. In most cases, if you are using a key listener, you wish to know which key was pressed. The KeyEvent has a property called character, which stores which character was just pressed. You can access this character value using **e.character** (where **e** is the KeyEvent parameter). You can also access the key code and state mask of the key that was pressed. The key codes for special keys are constants in the SWT library, like SWT.CTRL for the control key and SWT.CR for the carriage return key. The state mask indicates the status of the keyboard modifier keys. The example at the end of this section demonstrates how you can use the state mask.

The following example demonstrates using the key pressed and key released methods in the key listener. We attach the key listener to a basic text widget and a key event is generated every time you type in the text. In the key pressed method, we check to see if the key pressed was a special key (such as tab, carriage return, etc.), and print out the readable character for that key. Otherwise, we print out the character as is. In the key released method, we check to see if the control key was pressed before another character was pressed, such as Control-C. This is a method that could be used to capture ctrl-key sequences in a widget such as a text:

```
Text text = new Text(shell, SWT.MULTI|SWT.BORDER);
text.setBounds(10,10,100,100);
text.addKeyListener(new KeyListener() {
      public void keyPressed(KeyEvent e) {
            String string = "";
            switch (e.character) {
                  case 0:           string += " '\\0'"; break;
                  case SWT.BS:      string += " '\\b'"; break;
                  case SWT.CR:      string += " '\\r'"; break;
                  case SWT.DEL:     string += " DEL"; break;
                  case SWT.ESC:     string += " ESC"; break;
                  case SWT.LF:      string += " '\\n'"; break;
                  default:    string += " '" + e.character +"'";
                                    break;
            }
            System.out.println (string);
      }
      public void keyReleased(KeyEvent e) {
            if (e.stateMask == SWT.CTRL && e.keyCode != SWT.CTRL)
                  System.out.println("Command can execute here");
      }
});
```

Try this code out and see what kind of results you get from pressing various keys. Note that if we used a KeyAdapter instead of a KeyListener, we would have only had to implement one of keyPressed and keyReleased.

Mouse Listeners

There are three types of mouse listeners; MouseListener, which tracks pressing the mouse, MouseMoveListener, which tracks when the mouse is being moved, and MouseTrackListener, which tracks where the mouse is in relation to the control. We will go over all three of these listeners. Since we are dealing with events, it is not possible to show the results here, so make sure you try out the code, to see what kind of results you get.

MouseListener

The MouseListener has three methods, mouseDown, mouseUp and mouseDoubleClick. The functions of these methods are stated in the names. Because there are three methods in this listener, a MouseAdapter is available. The class that contains information on the event is MouseEvent.

The fields in MouseEvent which are unique to the MouseListener are x, y, stateMask and button. The button property returns the number of the button that was pressed or released. So the first button on the shell would return a value of 1, the second would return 2, and so on. The x and y properties return the location coordinates of the mouse at the time of the event. The stateMask property returns the state of the keyboard modifier keys at the time of the event.

The following example has two MouseAdapters. One of the adapters implements the mouseUp and mouseDown methods. This adapter is attached to two different buttons. We can do this by first creating the adapter, then adding it to whichever widgets need to use it. We find the number of the button that has been pressed, and also the coordinates of the mouse when it is both pressed and released. The other adapter implements the mouseDoubleClick method. It opens a new window when the double click event it created on the Label:

```
Button button = new Button(shell, SWT.PUSH);
button.setText("Push Me");
button.setBounds(10,10,60,20);
Button button2 = new Button(shell, SWT.PUSH);
button.setText("Push Me Too");
button.setBounds(100,10,60,20);

MouseAdapter mouseAdapter = new MouseAdapter() {
      public void mouseDown(MouseEvent e) {
            System.out.println("Button "+e.button+" pressed");
            System.out.println("Mouse pressed at ("
                  + e.x + ",” + e.y + ")");
            }
      public void mouseUp(MouseEvent e) {
            System.out.println("Mouse released at ("
                  + e.x + "," + e.y + ")");
      }
};
button.addMouseListener(mouseAdapter);
button2.addMouseListener(mouseAdapter);

Label label = new Label(shell, SWT.NONE);
label.setText("Double Click Me");
label.setBounds(10,40,100,20);
label.addMouseListener(new MouseAdapter() {
```

```
            public void mouseDoubleClick(MouseEvent e) {
                  Shell shell2 = new Shell(display);
                  shell2.setSize(100,100);
                  Label label2 = new Label(shell2,SWT.NONE);
                  label2.setText("Hello New Window!");
                  label2.setBounds(0,50,100,20);
                  shell2.open();
            }
      });
```

Try this code out. Note that if you press on the first button, it tells you that Button 1 has been pressed, and pressing on the second button tells you that Button 2 has been pressed. If you press your mouse down, move it around, and then release it, you will see the different coordinates that your mouse was pressed and released at.

Mouse Move Listener

The MouseMoveListener has one method, mouseMove. This method is invoked when the mouse is moved over the control onto which it is attached. The event generated is a MouseEvent again.

See the example at the end of this section.

Mouse Track Listener

The MouseTrackListener has three methods, mouseEnter, mouseExit and mouseHover. The mouseEnter method is called when the mouse enters into the area covered by the control to which it is attached. Similarly, the mouseExit method is called when the mouse exits said region. The mouseHover method is executed when the mouse hovers over the control to which it is attached.

The following example illustrates how to use both the mouse move listener and the mouse track listener. The first canvas has a mouse move listener attached to it. When you move the mouse over top of the canvas, the colour of the canvas changes. The second canvas has a mouse track listener attached to it. When your mouse enters over top of the canvas, a number is displayed in the canvas. Each time you hover your mouse over the canvas, the number is increased. When the mouse leaves the area of the canvas, the number disappears and starts at zero again when the mouse re-enters the area.

```
      int count1=0, count2=0;
      final Canvas canvas = new Canvas(shell, SWT.BORDER);
      canvas.setBounds(10,10,100,100);
      Canvas canvas2 = new Canvas(shell, SWT.BORDER);
      canvas2.setBounds(150,150,100,100);
      final Text text = new Text(canvas2, SWT.READ_ONLY);
      text.setBounds(40,30,20,20);
```

```
MouseMoveListener mouseMove = new MouseMoveListener() {
      public void mouseMove(MouseEvent e) {
            Color color = canvas.getBackground();
            color.dispose();
            canvas.setBackground(new
                  Color(display,count1,0,255));
            count1++;
            count1 = count1%255;
      }
};
canvas.addMouseMoveListener(mouseMove);

MouseTrackListener mouseTrack = new MouseTrackListener() {
      public void mouseEnter(MouseEvent arg0) {
            text.setText(Integer.toString(count2));
      }

      public void mouseExit(MouseEvent arg0) {
            text.setText("");
            count2 = 0;
      }

      public void mouseHover(MouseEvent arg0) {
            count2++;
            text.setText(Integer.toString(count2));
      }
};
canvas2.addMouseTrackListener(mouseTrack);
```

Note that in order to change a control within a listener, it must be declared as a final variable. Run this code and see what kind of results you get by generating mouse events on the controls.

Text Listeners

There are a couple of listeners that are useful when using Text widgets. These are the ModifyListener and the VerifyListener. The ModifyListener is called when text is modified, and the VerifyListener is called before text is modified. This one can be used to verify that the new values being entered are valid. We will briefly go over both of these listeners.

ModifyListener

The ModifyListener has one method, modifyText. The event object is a ModifyEvent. This event is similar to a TypedEvent, and has fields such as time and widget.

The ModifyListener can be used when we need to be notified if text has been changed. The modifyText method is called every time a character is modified. See the example at the end of this section.

VerifyListener

The VerifyListener has one method, verifyText. The event object is a VerifyEvent. This event has all the properties of a TypedEvent and a KeyedEvent, as well as the fields start, end, doit and text. The start and end properties indicate the range of text which is about to be modified. The text variable indicates the new text to which your text is being changed. The doit flag indicates whether or not we should complete the action that triggered the event.

We can use the VerifyListener to check and make sure certain characters can be entered in specific fields. In the following example, we check to see if the user is entering an asterisk in the text. If so, the action is cancelled. Notice that no matter how many times the '*' key is pressed, no asterisks will appear in the text.

```
Text text = new Text(shell, SWT.MULTI|SWT.WRAP);
text.setBounds(10,10,200,100);
text.setText("Here is some sample text");
text.addModifyListener(new ModifyListener() {
      public void modifyText(ModifyEvent e) {
            System.out.println("Modified at " + e.time);
      }
});
text.addVerifyListener(new VerifyListener() {
      public void verifyText(VerifyEvent e) {
            if (e.text.equals("*")) {
                  System.out.println("Cannot type *");
                  e.doit = false;
            }
      }
});
```

Focus Listeners

This section deals with FocusListener and TraverseListener. The FocusEvent is generated when a widget receives focus, by any means. A TraverseEvent is generated when a widget gains focus by means of traversal. We will cover both kinds of event handlers.

FocusListener

The FocusListener has two methods, focusGained and focusLost. They are called at obvious times. The event generated is a FocusEvent. This event has all the properties of a TypedEvent. See the example at the end of this section for more details.

TraverseListener

The TraverseListener has one method, keyTraversed. This is called when the widget the handler is attached to is traversed by a traversal method, such as tabs or up and down arrows.

The event generated is a TraverseEvent. Some useful properties of this event object are detail and doit. The doit property is the same as for the VerifyListener; it sets a flag indicating whether the event should continue. The detail property indicates what kind of traversal generated the event. These traversals can all be referred to by using SWT.TRAVERSE_XXXX.

In the following example, you can see we compare the detail in the TraverseEvent with SWT.TRAVERSE_TAB_PREVIOUS. If the traversal was a previous tab, we cancel the action.

```
Button b1 = new Button(shell, SWT.PUSH);
Button b2 = new Button(shell, SWT.PUSH);
Button b3 = new Button(shell, SWT.PUSH);
Button b4 = new Button(shell, SWT.PUSH);
Button b5 = new Button(shell, SWT.PUSH);
Button b6 = new Button(shell, SWT.PUSH);
b1.setBounds(10,10,50,50);
b2.setBounds(100,10,50,50);
b3.setBounds(200,10,50,50);
b4.setBounds(10,100,50,50);
b5.setBounds(100,100,50,50);
b6.setBounds(200,100,50,50);
b1.setText("1");
b2.setText("2");
b3.setText("3");
b4.setText("4");
b5.setText("5");
b6.setText("6");

FocusListener focusListener = new FocusListener() {
      public void focusGained(FocusEvent e) {
            System.out.println(e.widget + " has focus");
      }
      public void focusLost(FocusEvent e) {
            System.out.println("And now has lost it.");
      }
};
```

```
	TraverseListener traverseListener = new TraverseListener() {
		public void keyTraversed(TraverseEvent e) {
			if (e.detail == SWT.TRAVERSE_TAB_PREVIOUS) {
				System.out.println("You can't go back!");
				e.doit = false;
			}
			System.out.println(e.widget + " was traversed");
		}
	};

	b1.addFocusListener(focusListener);
	b4.addTraverseListener(traverseListener);
	b4.addFocusListener(focusListener);
	b6.addTraverseListener(traverseListener);
```

Basic Widgets Project

A project that contains the examples used in this document is available in a zip file format. In order to make it possible to run each demo from a main shell, the instructions used to create each demo had to be modified because Eclipse does not permit an application to create more than one display.

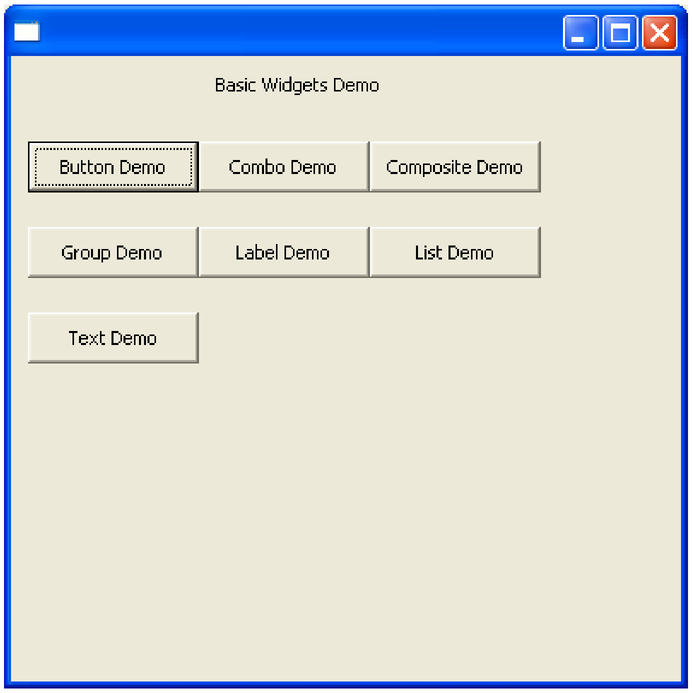
To run a demo program by itself, a display is created at the beginning of the run method.

```
public class ButtonDemo {
	public static void main(String[] args) {
		ButtonDemo bd = new ButtonDemo();
		bd.runDemo();
	}
	public void runDemo() {
		Display display = new Display();
		Shell shell = new Shell(display);
```

If the method may be called either by the demo program itself or by an external supervisory routine, the following changes must be made to the program structure.

```
public class ButtonDemo {

	public static Display myDisplay;
	public static boolean internalCall = false;

	public static void main(String[] args) {
		internalCall = true;
		myDisplay = new Display();
		ButtonDemo bd = new ButtonDemo();
		bd.runDemo(myDisplay);
	}

	public void runDemo(Display display) {
		myDisplay = display;
		Shell shell = new Shell(display);
```

Now, the method may be called by an external supervisory routine, as shown below.



Summary:

We have examined only the basic widgets, some of their common properties, and the basic event structure in this document. For more information on each widget, review the Eclipse API documentation at:

http://download.eclipse.org/downloads/documentation/2.0/html/plugins/org.eclipse.platform.doc.isv/reference/api/

We will examine some of the more sophisticated widgets in subsequent documents.