

Advanced SWT Widgets^{1,2}



by **Shantha Ramachandran, Christopher Batty, and Weichi Truong**

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

Last revised: February 20, 2004

Overview:

In this document, we describe some of the more advanced SWT widgets.

Tables

To add a table to your shell, use the following code:

```
Table table1 = new Table(shell, SWT.BORDER);
```

The styles that can be used are BORDER, H_SCROLL, V_SCROLL, SINGLE, MULTI, CHECK (this style creates a check box column as the first column), FULL_SELECTION and HIDE_SELECTION.

There are two options you can set for a table. You can decide whether the lines will be visible on the table. You can also decide whether the headers for the columns will be visible. Both of these properties are defaulted to false. The following lines of code will set these properties:

```
table1.setLinesVisible(true);  
table1.setHeaderVisible(true);
```

Now, the table is created, but as a default it only has one column. If you try to add any items to the table, only the first field of the item will be displayed in the table. In order to display all the fields you want, you need to define TableColumns.

TableColumns

To add table columns to your table, use the following code:

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

² © Shantha Ramachandran, Christopher Batty, and David Scuse

```
TableColumn name = new TableColumn(table1, SWT.LEFT);
```

The styles that are allowed are LEFT, RIGHT and CENTER.

You should also set at least the width of the column, and a name if you are going to display the headers:

```
name.setText("Name");  
name.setWidth(50);
```

After creating the columns for your table, you are now ready to add TableItems.

TableItems

To add a table item to your table, use the following code:

```
TableItem item1 = new TableItem(table1, SWT.NONE);
```

There are no styles for a TableItem, so you can use SWT.NONE, which is 0.

To set the text for an item, you can use the setText() method:

```
item1.setText(new String[] {"Sarah", "15", "390 Sussex Ave"});
```

This will create a TableItem with fields in the order that you defined them. If you wish to define one field at a time, you can use the setText(int, string) method which places the text into a specific field in the record.

After creating TableColumns and TableItems, your table will be fully populated. Now we will look at three simple examples of tables and see the different styles that can be used.

The first example has a border, fixed size columns, with different horizontal alignments:

```
Table table1 = new Table(shell, SWT.BORDER);  
table1.setBounds(10, 10, 270, 60);  
table1.setLinesVisible(true);  
  
TableColumn name = new TableColumn(table1, SWT.LEFT);  
name.setText("Name");  
name.setWidth(50);  
TableColumn age = new TableColumn(table1, SWT.RIGHT);  
age.setText("Age");  
age.setWidth(30);  
TableColumn address = new TableColumn(table1, SWT.LEFT);  
address.setText("Address");
```

```
address.setWidth(200);
```

```
TableItem item1 = new TableItem(table1, SWT.NONE);  
item1.setText(new String[] {"Sarah", "15", "390 Sussex Ave"});  
TableItem item2 = new TableItem(table1, SWT.NONE);  
item2.setText(new String[] {"Joseph", "56", "7 Yourstreet St"});
```

The second example has a checkbox column. We can set the checked status of the item through the code. There are no grid lines, but the header is visible. This example also sets the background color of one of the table items:

```
Table table2 = new Table(shell, SWT.CHECK|SWT.HIDE_SELECTION);  
table2.setBounds(10, 80, 270, 80);  
table2.setHeaderVisible(true);
```

```
TableColumn fruit = new TableColumn(table2, SWT.LEFT);  
fruit.setText("Fruit");  
fruit.setWidth(100);  
TableColumn colour = new TableColumn(table2, SWT.LEFT);  
colour.setText("Colour");  
colour.setWidth(170);
```

```
TableItem fruit1 = new TableItem(table2, SWT.NONE);  
fruit1.setText(0, "Apple");  
fruit1.setText(1, "Red");  
fruit1.setChecked(true);  
TableItem fruit2 = new TableItem(table2, SWT.NONE);  
fruit2.setText(new String[] {"Kiwi", "Green"});  
fruit2.setBackground(new Color(display, 255, 0, 0));  
TableItem fruit3 = new TableItem(table2, SWT.NONE);  
fruit3.setText(new String[] {"Banana", "Yellow"});
```

The last example implements the FULL_SELECTION style, which means that when you select a table item, the entire item is highlighted instead of just the first field. The first column is resizable, meaning you can drag it larger or smaller. Note that the three columns each have a different horizontal alignment. This example also shows how you can select an item using code:

```
Table table3 = new Table(shell, SWT.FULL_SELECTION);  
table3.setLinesVisible(true);  
table3.setBounds(10, 180, 270, 80);  
  
TableColumn first = new TableColumn(table3, SWT.LEFT);  
first.setResizable(true);  
first.setText("First");
```

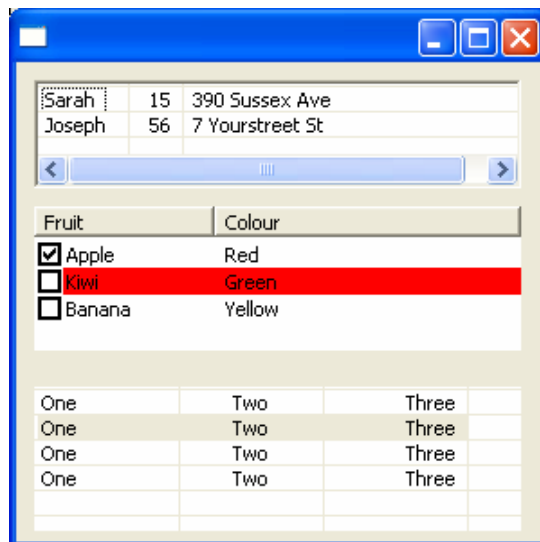
```

first.setWidth(80);
TableColumn second = new TableColumn(table3,SWT.CENTER);
second.setText("Second");
second.setWidth(80);
TableColumn third = new TableColumn(table3,SWT.RIGHT);
third.setText("Third");
third.setWidth(80);

String[] numbers = new String[] {"One","Two","Three"};
TableItem firstItem = new TableItem(table3,SWT.NONE);
firstItem.setText(numbers);
TableItem secondItem = new TableItem(table3,SWT.NONE);
secondItem.setText(numbers);
TableItem thirdItem = new TableItem(table3,SWT.NONE);
thirdItem.setText(numbers);
TableItem fourthItem = new TableItem(table3,SWT.NONE);
fourthItem.setText(numbers);
table3.select(1);

```

The following window incorporates all three of these examples:



TabFolder

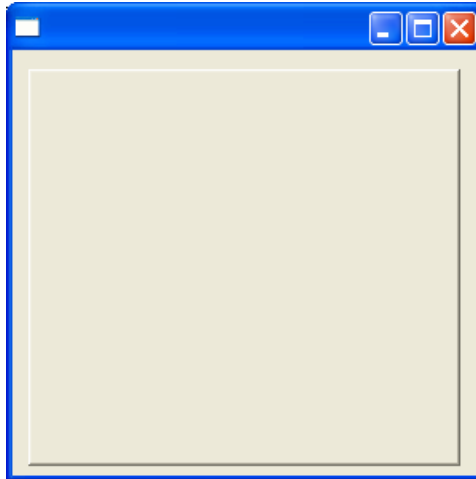
To add a tab folder to your shell, use the following code:

```

TabFolder tabFolder1 = new TabFolder(shell,SWT.NONE);

```

The BORDER style can be used for tab folders. If you create a tab folder and do not add any TabItems to it, it resembles a raised composite:



To create a `TabItem`, use the following code:

```
TabItem item1 = new TabItem(tabFolder1, SWT.NONE);
```

There are no styles available for `TabItems`. There are two important steps for creating tab items. First, if you want a label on the tab, you need to use the `setText()` method. Secondly, in order to populate the tab item, you need to set the control which it contains, using the `setControl()` method:

```
item1.setText("Buttons");  
item1.setControl(buttonComp);
```

Now, in this example, `buttonComp` is a `Composite`, as you will see in the following example. Since you can only set one control into a tab item, a `Composite` is usually desired. In this case, you can add multiple controls to the composite, and they will all appear on the specified tab item. This is all the code that is necessary, as events for tab changing are built into the `TabItem` and `TabFolder` controls:

```
TabFolder tabFolder1 = new TabFolder(shell, SWT.NONE);  
tabFolder1.setBounds(10, 10, 270, 250);  
//Set up the button tab  
Composite buttonComp = new Composite(tabFolder1, SWT.NONE);  
Button button1 = new Button(buttonComp, SWT.PUSH);  
button1.setSize(100, 100);  
button1.setText("Hello");  
button1.setLocation(0, 0);  
Button button2 = new Button(buttonComp, SWT.ARROW);  
button2.setBounds(150, 0, 50, 50);  
  
TabItem item1 = new TabItem(tabFolder1, SWT.NONE);  
item1.setText("Buttons");
```

```

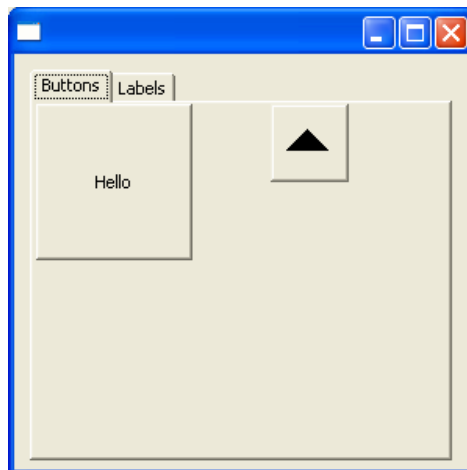
item1.setControl(buttonComp);

//Set up the label tab
Composite labelComp = new Composite(tabFolder1,SWT.NONE);
Label label1 = new Label(labelComp,SWT.NONE);
label1.setText("Here are some labels for your viewing pleasure");
label1.setBounds(0,0,250,20);
Label label2 = new Label(labelComp,SWT.NONE);
label2.setText("A label is a fine fingered fiend");
label2.setBounds(0,40,200,20);

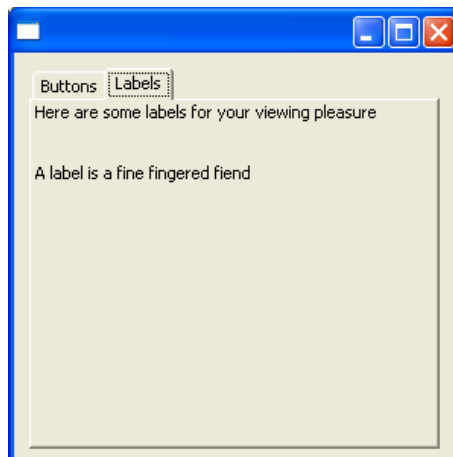
TabItem item2 = new TabItem(tabFolder1,SWT.NONE);
item2.setText("Labels");
item2.setControl(labelComp);

```

This example creates a TabFolder with two TabItems, one for buttons and one for labels. When the window initially appears, it looks like this:



If you click on the **Labels** tab, the tab items switch, and the screen looks like this:



Slider, Scale and Progress Bar

The slider, scale and progress bar are all very similar controls. However, they use a few different methods in order to function properly. We will briefly go over all three, and then show a small example for each control. We will start with the slider and the scale, as these two controls have many of the same functions.

To create a slider, use the following code:

```
slider slider1 = new Slider(shell, SWT.HORIZONTAL);
```

To create a scale, use the following code:

```
scale scale1 = new Scale(shell, SWT.HORIZONTAL);
```

The styles that can be applied to a slider or a scale are BORDER, HORIZONTAL and VERTICAL.

You can set the minimum and maximum value of a slider or a scale using the setMinimum and setMaximum methods.

```
slider1.setMaximum(100);  
slider1.setMinimum(0);
```

You may wish to set the selection of the slider or the scale. This indicates where on the slider you will place the thumb, or where on the scale the marker will be positioned. In this case, the left edge of the thumb will be placed halfway between the two ends of the slider.

```
slider1.setSelection(50);
```

On a slider, you can also set the size of the thumb. This size is relative to the minimum and maximum values that you have set. For example, if you set the minimum to 0 and the maximum to 100, then setting the thumb size to 30 will make the thumb about one third of the length of the slider.

```
slider1.setThumb(30);
```

On a scale, you can set the page increment. This determines how often measuring bars are placed on the scale. For example, if you set the minimum to 0 and the maximum to 500, then setting the page increment to 50 means there will be 10 measuring bars along the scale. It will place a bar every 50 places, relative to the minimum and the maximum.

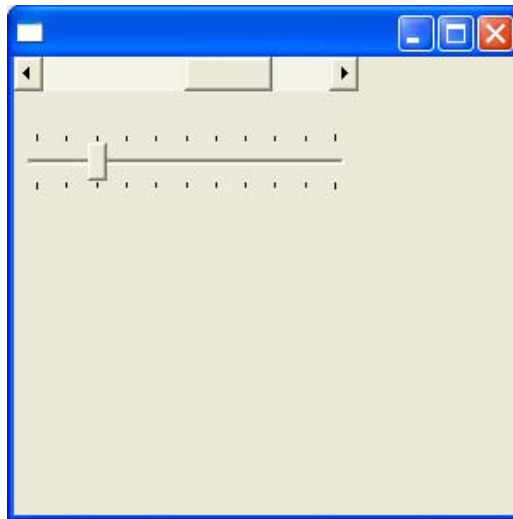
```
scale1.setPageIncrement(50);
```

The following example puts together these two widgets and the methods discussed:

```
Slider slider1 = new Slider(shell, SWT.HORIZONTAL);
slider1.setBounds(0,0,200,20);
slider1.setSelection(50);
slider1.setMaximum(100);
slider1.setMinimum(0);
slider1.setThumb(30);

Scale scale1 = new Scale(shell, SWT.HORIZONTAL);
scale1.setBounds(0,40,200,40);
scale1.setMinimum(0);
scale1.setMaximum(500);
scale1.setSelection(100);
scale1.setPageIncrement(50);
```

This code produces the following window:



Next, we are going to look at the progress bar.

To create a progress bar, use the following code:

```
ProgressBar progressBar1 = new ProgressBar(shell, SWT.HORIZONTAL);
```

The styles that can be used for progress bar are BORDER, VERTICAL and HORIZONTAL, just like the slider and scale. However, you can also use the style SMOOTH, which turns the progress indicator into a smooth bar.

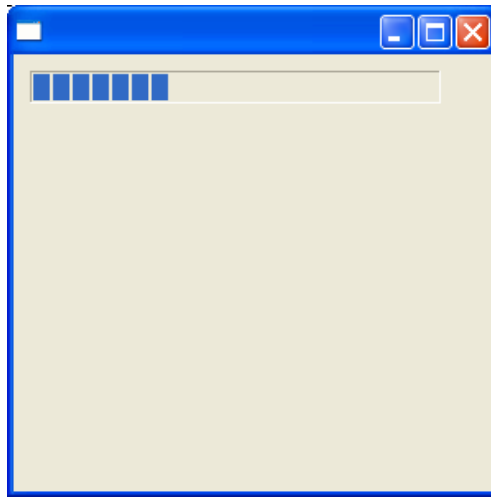
For a progress bar, the only properties which you may want to set are minimum and maximum values, and the selection. The selection indicator is based on the minimum and

the maximum. Unlike the scale or the slider, you cannot set the width of the progress indicator bars, or the size of the progress marker.

The following code illustrates these methods:

```
ProgressBar progressBar1 = new ProgressBar(shell, SWT.HORIZONTAL);  
progressBar1.setMinimum(0);  
progressBar1.setMaximum(100);  
progressBar1.setSelection(30);  
progressBar1.setBounds(10, 10, 250, 20);
```

This produces the following window:



Menus and MenuItem:

We will look at two examples of menus. The first will be the most basic use, to create a simple menu with a selectable option on it, and capture the event generated when it is selected. The second, larger example will demonstrate some of the more complicated aspects of menus, including submenus, accelerators, mnemonics, additional Events that may be captured, and the various different types of menu items available.

To set up menus, you make use of the Menu and MenuItem widgets (see SimpleMenuDemo.java).

To create the menu bar that spans the top of your Shell use the following code:

```
Menu menu = new Menu(shell, SWT.BAR);  
shell.setMenuBar(menu);
```

The style constant for the menu bar must be SWT.BAR.

Next we want to add an item to the menu bar.

```
MenuItem file = new MenuItem(menu, SWT.CASCADE);
file.setText("File");
```

This will create a “File” item on the menu bar. If you run it with just these lines, you will see the menu bar at the top of the shell, with the option “File” displayed. However, if you select one of the options, nothing will happen. Make sure that the style constant is SWT.CASCADE, because otherwise you will be unable to attach a drop-down menu to it.

We will now attach a menu to the File option.

```
Menu filemenu = new Menu(shell, SWT.DROP_DOWN);
file.setMenu(filemenu);
```

This creates the file menu, and attaches it to the correct MenuItem on the Menu bar. Note that the style constant for the menu must be DROP_DOWN.

Finally, we can now add a MenuItem to our file menu, and then associate an action with it.

```
MenuItem actionItem = new MenuItem(filemenu, SWT.PUSH);
actionItem.setText("Action");
```

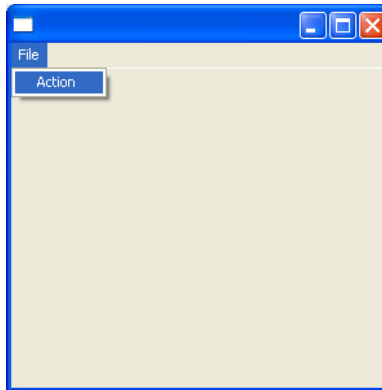
This creates a basic selectable menu item on the File menu.

To be notified when this option is selected, we add a listener, like so:

```
actionItem.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event e) {
        System.out.println("Action performed!");
    }
});
```

This will output the words “Action performed!” to the console, whenever the “Action” option is chosen. Repeating these steps with more Menus and MenuItems is all you need to do to create a useful set of simple menus for your program.

The resulting menu looks like this:



We begin our longer example (see `MenuDemo.java`) with the same initial code as our first example. The first additional feature we will consider is types of `MenuItem`s other than `SWT.PUSH`. Alternate choices for the style constants are `CHECK`, `RADIO`, `SEPARATOR` and `CASCADE`. The first is a checkbox item, while the second is a radio button item. `SEPARATOR` is used to provide a non-selectable dividing horizontal line between other items. The following code demonstrates the `CHECK`, `SEPARATOR` and `RADIO` options. It will create a separator, then a checkbox-style menu item, followed by radio-button style menu item.

```
MenuItem separator = new MenuItem(filemenu, SWT.SEPARATOR);
final MenuItem radioItem = new MenuItem(filemenu, SWT.RADIO);
radioItem.setText("Radio");
final MenuItem checkItem = new MenuItem(filemenu, SWT.CHECK);
checkItem.setText("Check");
```

The next few lines add listeners to the Check and Radio items, which output their current values (true or false) to the console.

```
radioItem.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event e) {
        System.out.println("Radio item toggled to:" +
            radioItem.getSelection());
    }
});
checkItem.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event e) {
        System.out.println("Check item toggled to:" +
            checkItem.getSelection());
    }
});
```

We will now add a sub-menu that branches off of the file menu, which is very similar to how we attached our file menu to the menu bar. We create a new `MenuItem` with style

constant `SWT.CASCADE`, and then create a new menu to attach to it. We then call `setMenu` on the `MenuItem` to which we are attaching the new menu.

```
MenuItem cascadeItem = new MenuItem(filemenu, SWT.CASCADE);
cascadeItem.setText("Cascade");
Menu submenu = new Menu(shell, SWT.DROP_DOWN);
cascadeItem.setMenu(submenu);
```

The next few lines add a menu item to our new submenu. The new item that we add will be called `SubAction`. Notice that in the `setText` method call, there is an ampersand(&) before the letter `S`. This will create a mnemonic for the `SubAction` command, so that when the submenu is displayed, pressing the `S`-key will select it as if you had used the mouse. The call in the following line, to `setAccelerator` associates the `SubAction` menu item with the key combination `Control-S`, so the user can execute the `SubAction` command without bringing up the menu.

```
final MenuItem subactionItem = new MenuItem(submenu, SWT.PUSH);
subactionItem.setText("&SubAction\tCtrl+S");
subactionItem.setAccelerator(SWT.CTRL+'S');
```

We add an additional option to the sub-menu which will be used to demonstrate enabling and disabling of `MenuItems`. This checkbox will enable/disable the `SubAction` we just added.

```
final MenuItem enableItem = new MenuItem(submenu, SWT.CHECK);
enableItem.setText("Enable SubAction");
```

There are a variety of additional listeners that we didn't look at in our simple example. One of these is the ability to add listeners to `Menus` (rather than `MenuItems` as we did before.) Here we add a `MenuListener` to our submenu, which notifies us whenever it is shown or hidden.

```
submenu.addMenuListener(new MenuListener() {
    public void menuShown(MenuEvent e) {
        System.out.println("SubMenu shown");
    }
    public void menuHidden(MenuEvent e) {
        System.out.println("SubMenu hidden");
    }
});
```

Next we will demonstrate the ability to enable and disable menu items, using another checkbox item. The first line disables `SubAction` to begin with, so its text is shown shaded in the menu. When disabled you cannot select the option, and its accelerator also

no longer works. The code after that sets up the listener, which will disable or enable the SubAction MenuItem depending on whether the “Enable SubAction” checkbox is checked.

```
subactionItem.setEnabled(false);
enableItem.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event e) {
        System.out.println("Toggling \"Enable SubAction\" to " +
            enableItem.getSelection());
        subactionItem.setEnabled(enableItem.getSelection());
    }
});
```

Before when we set up a listener for the Action item, we used a generic Listener. Here for demonstration purposes we make use of the more specific SelectionListener. AS before we merely print a message to the console on activation.

```
subactionItem.addSelectionListener(new SelectionListener() {
    public void widgetSelected(SelectionEvent e) {
        System.out.println("SubAction performed!");
    }
    public void widgetDefaultSelected(SelectionEvent e) {
    }
});
```

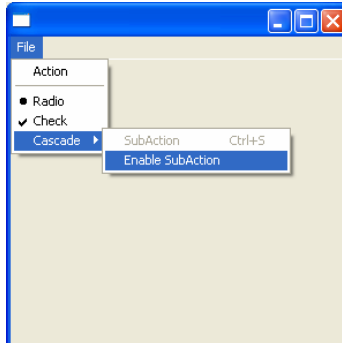
Another interesting listener is the ArmListener, which is fired when a menuitem is highlighted with the mouse or keyboard, but not yet selected. The following code prints a message when the SubAction is armed.

```
subactionItem.addArmListener(new ArmListener() {
    public void widgetArmed(ArmEvent e) {
        System.out.println("SubAction armed!");
    }
});
```

A final listener type is the HelpListener. By pressing F1/Help key, the HelpListener is activated, which maybe useful for providing help on various options.

```
subactionItem.addHelpListener(new HelpListener() {
    public void helpRequested(HelpEvent e) {
        System.out.println("Help requested on SubAction");
    }
});
```

The result is the following menu. Selecting the Enable SubAction option will enable SubAction. Hitting Ctrl+S will execute SubAction as well, but only if Enable SubAction is checked. All the listeners print messages to the console when they are activated.



There are numerous other facilities available for Menus and MenuItems, but we've attempted to cover the majority of the useful ones. Next we will look briefly at a simple pop-up menu.

Popup Menus

Pop-up menus (see `PopupMenuDemo.java`) are useful for context-sensitive menus on different areas of the screen, rather than making the user go to a Menu bar at the top of the screen and hunt for the option they want. By right-clicking on a Composite that has a pop-up menu associated, the menu will appear floating beside your mouse, allowing you to select from a list of options like a regular menu. In the demo below, we will create a button, in a composite, on a shell. Then we will create 2 pop-up menus, and associate one with the shell and composite, the other with the button.

We create the menu, as we did with other menus, except the style constant must be `SWT.POP_UP`. We then add an item to it, and attach a listener to capture selection events. This will be the menu for the shell and composite.

```
Menu popupmenu = new Menu(shell, SWT.POP_UP);
MenuItem actionItem = new MenuItem(popupmenu, SWT.PUSH);
actionItem.setText("Other Action");
actionItem.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event e) {
        System.out.println("Other Action performed!");
    }
});
```

Next we repeat the process, with this menu being associated with the button.

```
Menu popupmenu2 = new Menu(shell, SWT.POP_UP);
```

```
MenuItem buttonItem = new MenuItem(popupmenu2, SWT.PUSH);
buttonItem.setText("Button Action");

buttonItem.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event e) {
        System.out.println("Button Action performed!");
    }
});
```

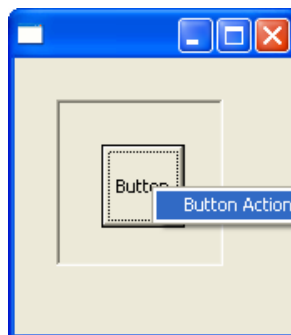
Next we create the composite and button, and place them on the shell.

```
Composite c1 = new Composite (shell, SWT.BORDER);
c1.setSize (100, 100);
c1.setLocation(25,25);
Button b = new Button(c1, SWT.PUSH);
b.setText("Button");
b.setSize(50,50);
b.setLocation(25,25);
```

Finally, we set the pop-up menus for each part. Notice that we reuse the pop-up menu from the Composite on the Shell.

```
b.setMenu(popupmenu2);
c1.setMenu (popupmenu);
shell.setMenu (popupmenu);
```

Now if we run the code, and right-click on the button, it will display one menu, but if we right-click the composite or shell, we get another menu with different options. In the following screen shot we can see the pop-up menu that is displayed when right-clicking on the button.



Although this is a simple example, pop-up menus can be just as complex as the earlier Menu bar example, with submenus, radio-items, listeners, etc.

Trees

Trees are used to display information in a hierarchical format, such as displaying folders and sub-folders in a file manager, or classes and subclasses in a class browser tool.

The following code (TreeDemo.java) creates the tree. The standard style constants for Trees are SWT.MULTI, SWT.SINGLE, and SWT.CHECK. SINGLE allows the selection of only one item from the Tree at a time, while MULTI allows multiple selections. Only one of SINGLE and MULTI may be specified at a time. CHECK adds check boxes beside each item in the tree. As an example, Eclipse uses this CHECK style of tree for choosing which files to import/export from a folder.

The Tree we create here allows multiple selections, and has a border. As usual, the first parameter to the Tree constructor is the Composite on which we want to place the tree, and the second is the SWT style constants.

```
final Tree tree = new Tree(shell, SWT.MULTI | SWT.BORDER);
tree.setSize(150, 150);
tree.setLocation(5,5);
```

We must now attach TreeItems to the tree. Here we create three top (root) level items on the Tree. The first parameter to the constructor is the Tree that we want to attach the items to. The second is the SWT style constant, and the (optional) last parameter is the index at which to place the item in the tree. In this example, the argument 1 specifies that “My Documents” will appear as the middle element in the top-level of the tree, even though it was added after the other two items.

```
TreeItem myComp = new TreeItem(tree, SWT.NONE);
myComp.setText("My Computer");
TreeItem netPlaces = new TreeItem(tree, SWT.NONE);
netPlaces.setText("My Network Places");
TreeItem myDocs = new TreeItem(tree, SWT.NONE, 1);
myDocs.setText("My Documents");
```

To create children TreeItems off of these root-level nodes, we use a similar constructor, except that instead of specifying the Tree to attach them to, we specify the TreeItem that they will appear under. Both of these nodes will appear under the “My Computer” TreeItem. As in the above case, we can use the optional last parameter to specify the index at which the new element should be placed in the sub-tree.

```
TreeItem hardDisk = new TreeItem(myComp, SWT.NONE);
hardDisk.setText("Local Disk (C:)");
TreeItem floppy = new TreeItem(myComp, SWT.NONE, 0);
```

```
floppy.setText("3.5 Floppy (A:)");
```

We can continue the process of creating sub-trees indefinitely, simply by creating more `TreeItems` off of other `TreeItems`. Here we extend the tree to a third level by creating a `TreeItem` off of the “Local Disk” `TreeItem`.

```
TreeItem progFiles = new TreeItem(hardDisk, SWT.NONE);  
progFiles.setText("Program Files");
```

We can also add an image to a `TreeItem`. We load the image as usual (see the section on Images), and then use `TreeItem`'s `setImage` method, as follows:

```
Image icon = new Image(display, "arrow.bmp");  
progFiles.setImage(icon);
```

When the `progFiles` item is visible in the tree, the image will be displayed to the left of the text of the item.

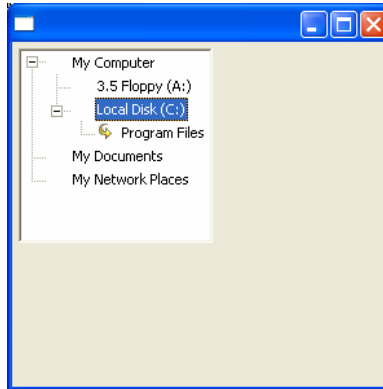
We will now add a listener to the tree. In our examples with menus, the listeners were added to the `MenuItems`, but in the case of Trees, listeners are associated with the tree as a whole. We add a selection listener to the Tree, and can then use the `getSelection()` method of the tree to return a list of the selected `TreeItems`. In the example below, we listen for changes in the selection(s), get an array of `TreeItems` that are selected and print out the text from each item.

```
tree.addSelectionListener(new SelectionAdapter() {  
    public void widgetSelected(SelectionEvent e) {  
        TreeItem[] t = tree.getSelection();  
        System.out.print("Selection: ");  
        for(int i = 0; i < t.length; i++) {  
            System.out.print(t[i].getText() + ", ");  
        }  
        System.out.println();  
    }  
});
```

We can also listen for collapsing and expanding of the tree using a `TreeListener`. We will print “Tree collapsed” and “Tree expanded” when the corresponding event occurs.

```
tree.addTreeListener(new TreeListener() {  
    public void treeCollapsed(TreeEvent e) {  
        System.out.println("Tree collapsed.");  
    }  
    public void treeExpanded(TreeEvent e) {  
        System.out.println("Tree expanded.");  
    }  
});
```

The resulting program should look something like this:



If you use a tree with the SWT.CHECK style, TreeItem's isChecked() method can be used to test whether a particular item in the tree has been checked.

ToolBars

We will now consider tool bars. These are often useful for providing handy visual shortcuts to frequently used tasks, rather than requiring users to dig through complicated menus. The buttons on a toolbar behave in a very similar manner to standard buttons, except that they are grouped on a toolbar, and they can contain images and text at once.

We will create a simple toolbar (ToolBarDemo.java) with several of the standard types of buttons, with and without images and text. First we create and position the toolbar on the shell.

```
final ToolBar bar = new ToolBar(shell, SWT.HORIZONTAL);  
bar.setSize(380,150);  
bar.setLocation(10,10);
```

We want to place an image on some of the buttons, so we must first load it into an Image object.

```
Image icon = new Image(display, "arrow.bmp");
```

Next, we instantiate three ToolItems. Each is of the basic push button style. The first will contain the text "Push", the second will contain the image we just loaded, and the third will hold both the image and the text.

```
ToolItem pushItem1 = new ToolItem(bar, SWT.PUSH);  
pushItem1.setText("Push");  
ToolItem pushItem2 = new ToolItem(bar, SWT.PUSH);  
pushItem2.setImage(icon);
```

```
ToolItem pushItem3 = new ToolItem(bar, SWT.PUSH);
pushItem3.setText("Push");
pushItem3.setImage(icon);
```

As with menus, we have a SEPARATOR style to create a visible separation between items on the toolbar. This is done using a variation of the following line.

```
ToolItem sep = new ToolItem(bar, SWT.SEPARATOR);
```

Again, as with regular buttons and menus, we have CHECK and RADIO style buttons. ToolItem's getSelection() method can be used to test if a particular radio or check button is selected at any given time.

```
ToolItem checkItem = new ToolItem(bar, SWT.CHECK);
checkItem.setText("Check");
ToolItem sep2 = new ToolItem(bar, SWT.SEPARATOR);
```

We now add 2 radio buttons, a separator, and another 2 radio buttons. There are two important things to notice here. The first is that the radio behaviour is done automatically, (as opposed to the case of Menus where this behaviour must be programmed "by hand" by the application programmer.) The second point is that the behaviour only occurs for radio buttons that are adjacent to one another. If another type of ToolItem is between two RADIO ToolItems, the two will not behave as a group, but instead will be independently selectable. The following piece of code demonstrates this by creating two "groups" separated by a SEPARATOR ToolItem.

```
ToolItem radioItem1 = new ToolItem(bar, SWT.RADIO);
radioItem1.setText("Radio 1");
ToolItem radioItem2 = new ToolItem(bar, SWT.RADIO);
radioItem2.setText("Radio 2");

ToolItem sep3 = new ToolItem(bar, SWT.SEPARATOR);

ToolItem radioItem3 = new ToolItem(bar, SWT.RADIO);
radioItem3.setText("Radio 3");
ToolItem radioItem4 = new ToolItem(bar, SWT.RADIO);
radioItem4.setText("Radio 4");
```

The last kind of ToolItem that we can create is a drop-down button. The left half of the button acts as a standard push button, while the right half has a downward arrow indicating a drop-down menu. In order to actually attach a drop-down menu to a drop-down ToolItem, you must capture the selection event on that Item, and use the SelectionEvent's (x,y) coordinates to determine where to place the corresponding pop-up menu on the screen. There is a snippet of code demonstrating this technique on the SWT

Resources page on the Eclipse.org web site. We've taken this code and modified it to fit within this example.

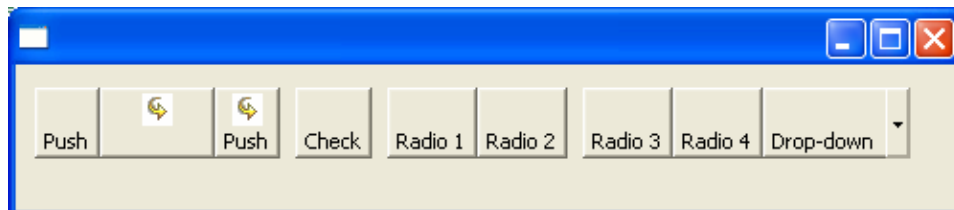
```
final ToolItem dropdown = new ToolItem(bar, SWT.DROP_DOWN);
dropdown.setText("Drop-down");
final Menu menu = new Menu(shell, SWT.POP_UP);
MenuItem choice = new MenuItem(menu, SWT.PUSH);
choice.setText("Choices");

dropdown.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event event) {
        if (event.detail == SWT.ARROW) {
            Rectangle rect = dropdown.getBounds();
            Point pt = new Point(rect.x, rect.y + rect.height);
            pt = bar.toDisplay(pt);
            menu.setLocation(pt.x, pt.y);
            menu.setVisible(true);
        }
    }
});
```

Finally, we demonstrate some basic event handling for ToolBars, by adding a SelectionListener to one of our push buttons. A message is printed to the console when the correct (leftmost) button is pressed.


```
pushItem1.addSelectionListener( new SelectionListener () {
    public void widgetSelected(SelectionEvent e) {
        System.out.println("Push button one selected.");
    }
    public void widgetDefaultSelected(SelectionEvent e) {
    }
});
```

The resulting screen will look like this:



CoolBars

CoolBars are used to create toolbar-like elements that can be dynamically repositioned by the user. For example, the toolbars at the top of the screen in the Eclipse workbench are

CoolBars, since you can click on the vertical bars (that look like this: ) and drag them around to reposition the groups of buttons on the toolbar. Similar CoolBars also appear in Microsoft Word and Internet Explorer.

We will create a simple CoolBar (CoolBarDemo.java), consisting of three CoolItems. Each CoolItem represents a piece of the CoolBar that can be moved around. We can place whatever Controls we like on each CoolItem. In the following example, we will create one empty CoolItem, two CoolItems containing Buttons and a final CoolItem containing a Toolbar.

We create the CoolBar in a standard way, and give it a border. Then four CoolItems are created, specifying "bar" as the CoolBar to which they should be added.

```
final CoolBar bar = new CoolBar(shell, SWT.BORDER);
CoolItem item1 = new CoolItem(bar, SWT.NONE);
CoolItem item2 = new CoolItem(bar, SWT.NONE);
CoolItem item3 = new CoolItem(bar, SWT.NONE);
CoolItem item4 = new CoolItem(bar, SWT.NONE, 2);
```

Next we must go through the process of creating each of the Controls that we will subsequently place on the CoolItems. The first is a flat button with a border, that will be used to demonstrate locking and unlocking of the CoolBar.

```
Button button1 = new Button(bar, SWT.FLAT | SWT.BORDER);
button1.setText("Button");
button1.pack();
```

By packing the button we set it to its correct size. The listener that we next add to the button will call `bar.setLocked(boolean)` to lock and unlock the CoolBar each time it is clicked on. Locking the toolbar removes the ability to click and drag the CoolItems around within the bar.

```
button1.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        bar.setLocked(!bar.getLocked()) ;
    }
});
```

The next button is just a regular button, to be placed in its own CoolItem.

```
Button button2 = new Button(bar, SWT.PUSH);
button2.setText("Another button");
button2.pack();
```

A common use of CoolBars is to create toolbars that can be dynamically rearranged by the user. This is how Eclipse uses them. The next piece of code places a basic 2-button toolbar on its own CoolItem.

```
ToolBar tools = new ToolBar(bar, SWT.NONE);
ToolItem b1 = new ToolItem(tools, SWT.FLAT);
b1.setText("Tool");
ToolItem b2 = new ToolItem(tools, SWT.FLAT);
b2.setText("Bar");
tools.pack();
```

After we have created each of the controls, we can use CoolItem's setControl() method to associate the controls with the right CoolItems. We also use setSize to set the initial size of the each control. Notice how the setMinimumSize method is called on item3. This ensures that when the user rearranges the CoolBar, the CoolItem contain the ToolBar doesn't shrink beneath the size we request, which in this case is the size of the ToolBar itself.

```
Point size = button1.getSize();
item1.setControl(button1);
item1.setSize(item1.computeSize(size.x, size.y));

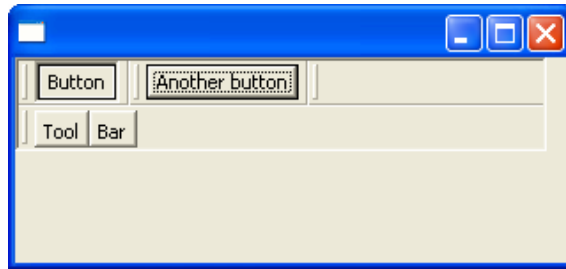
size = button2.getSize();
item2.setControl(button2);
item2.setSize(item2.computeSize(size.x, size.y));

size = tools.getSize();
item3.setControl(tools);
item3.setSize(item3.computeSize(size.x, size.y));
item3.setMinimumSize(size);
```

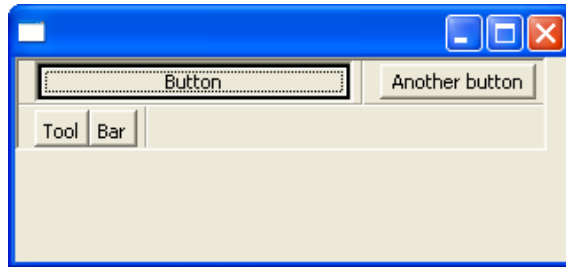
Lastly, setWrapIndices is used to wrap some CoolItems onto the next row at a particular index. setSize is called on bar to specify its size on the shell.

```
bar.setWrapIndices(new int[] {3});
bar.setSize(300, 120);
```

Here is what the CoolBar looks like initially:



After some rearrangement, and locking of the CoolBar, the screen looks like this:



Graphics

SWT provides a wide variety of graphics capabilities. These capabilities are centred largely in the GC class, which we will look at first.

Graphics Contexts

The graphics context, or GC, is the class that you will use to do most of your drawing and displaying of images using SWT. It provides a large number of basic graphics functions, which may be useful if you are doing custom graphics or widgets.

It allows you to:

- draw or fill rectangles, ovals, lines, polygons, or text
- set the background and foreground colours
- set clipping regions
- copy regions to images or other areas of the same widget
- display images
- get and set text/font characteristics.

In this demo (GCDemo.java), we will create a Canvas, and use it to draw on. Note that you can obtain a GC for nearly any widget, and draw on it just as we will with Canvas. After we create the canvas, we will make use of several of the methods of GC to demonstrate the range of graphics functions available in SWT.

To create the canvas, we use the standard constructor, and set the size and location of the widget.

```
Canvas canvas = new Canvas(shell, SWT.BORDER);  
canvas.setSize(150, 150);  
canvas.setLocation(20, 20);
```

Next we open the shell. This is an important step, because if the shell is not open when we do the drawing onto the canvas, the resulting shapes and text will not appear.

```
shell.open();
```

To get a GC for a particular widget, we use the following code:

```
GC gc = new GC(canvas);
```

This graphics context applies just to canvas, and any drawing we do on it will be clipped to the canvas. That is, if we draw outside the boundaries of the canvas, it will not be displayed on neighbouring areas of the shell.

A basic task in graphics is to draw and fill rectangles. We do this with `drawRectangle` and `fillRectangle`. `drawRectangle` uses the current foreground colour (black, so far) to draw the rectangle. The parameters are the (x, y) coordinates of the rectangle, followed by the width, and height of the rectangle. In this example, the upper left corner is at (3, 5) and the lower left is at (3+20, 5+25) = (23, 30).

```
gc.drawRectangle(3, 5, 20, 25);
```

`fillRectangle` is similar to `drawRectangle`, except that the filling is done with the current background colour (currently the default, gray). Since we haven't set the background colour here, the filled rectangle will not be visible on the screen.

```
gc.fillRectangle(30, 5, 20, 25);
```

The easiest way to get colours to work with is to use `Display`'s `getSystemColor` method in conjunction with an SWT colour constant. It is possible to specify custom colours, and this technique will be covered later on.

```
Color blue = display.getSystemColor(SWT.COLOR_BLUE);  
Color red = display.getSystemColor(SWT.COLOR_RED);
```

Now that we have some colours, we can set the foreground colour. The lines after that demonstrate some other basic drawing functions. Note that corresponding fill functions exist for `Oval` and `Polygon`. In the `Oval` case, the parameters specified represent a

rectangle that bounds the oval. For Polygon, the integer array specifies x and y coordinates of points, alternating x and y.

```
gc.setForeground(blue);  
gc.drawLine(80, 20, 100, 80);  
gc.drawOval(40, 40, 10, 10);  
gc.drawPolygon(new int[] {100, 100, 120, 120, 140, 100});
```

We will now demonstrate what filling looks like, by setting the background colour and drawing some additional filled rectangles. Notice that a filled rectangle doesn't cover the exact same space as a drawn rectangle. The fill covers from the top left point to one row and one column less than the corresponding drawn rectangle. In this code, we create a filled rectangle, and another filled rectangle with a drawn rectangle at the same coordinates.

```
gc.setBackground(red);  
gc.fillRect(20, 100, 20, 20);  
gc.fillRect(50, 100, 20, 20);  
gc.drawRect(50, 100, 20, 20);
```

Another useful feature is drawing text. The drawString() method is used for this purpose, specifying the string and the x,y coordinates.

```
gc.drawString("Text", 120, 20);
```

We can also set clipping regions, so that drawing is restricted to a particular clipping window within the widget we are drawing on. The setClipping method works similar to the drawRectangle method, taking (x, y, width, height) integer parameters. In the code below we will clip a filled oval to a particular region. The clipping has no effect on previously drawn shapes.

```
gc.setClipping(40, 60, 40, 40);  
gc.fillOval(30, 50, 30, 25);
```

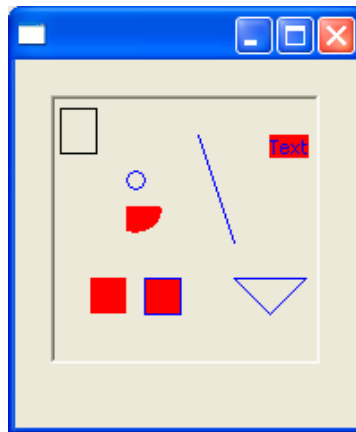
Clipping windows can also be specified using Regions, which are essentially the union of a set of rectangles. This can allow for more complex clipping areas.

Clipping is the last of the major drawing functions covered in this example. There is one more important step however. Numerous graphics classes in the SWT libraries use resources that are allocated at the OS level, and as such they must be explicitly freed up by the programmer. The standard code we have been using did this for display, after the shell was closed. In our example, GC must be disposed with the line:

```
gc.dispose();
```

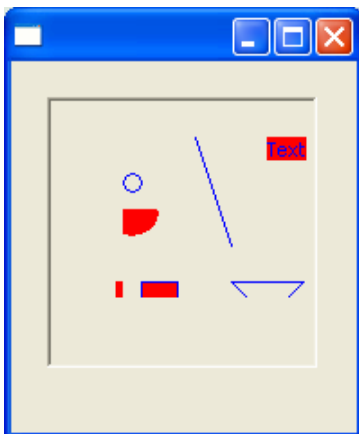
This line can be placed after the last line where drawing is done. It is usually a good policy to dispose of graphics objects like this as soon as you are finished using them. Other classes that usually require explicit disposal include: Colors, Images, and Fonts. You may have noticed that we did not dispose of the Colors blue and red in our example. The reason for this is that they are pre-allocated system colours. A general rule of thumb for disposing of objects is that if you created it, you should dispose of it. Similarly, if you didn't create it, it is not your responsibility to dispose of it. Later on we will demonstrate how to create custom Colors in SWT and these must be explicitly disposed of.

Getting back to our example, the output of the program will look something like the following:



GC contains numerous other functions for graphics, including setting line width and style, getting characteristics of text, and copying areas of the widget. These are detailed in the SWT javadocs.

In the previous example, there was a major flaw in the program. If you minimize the window, or drag it so that part of the canvas is obscured, the drawing that was done is erased. This is because we did the drawing only once when the program started up, and the drawing we did is never repainted a second time.



Eg. After dragging the window off the edge of the screen, part of what we have drawn is erased, and never repainted.

In order to detect when the canvas needs to be repainted, we must add a `PaintListener` to it. This next

demo (GCDemo2.java) retains the rectangle in the top-right, and repaints it whenever it becomes obscured.

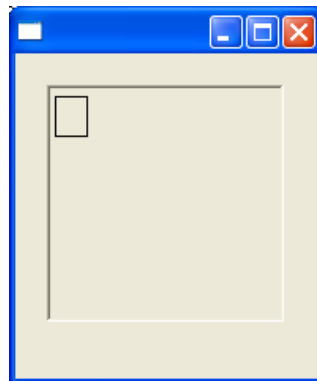
Note: It is rare that it should ever be necessary to do any drawing outside of a `PaintListener`, and it should be avoided most of the time.

```
Canvas canvas = new Canvas(shell, SWT.BORDER);
canvas.setSize(150, 150);
canvas.setLocation(20, 20);
canvas.addPaintListener(new PaintListener() {
    public void paintControl(PaintEvent e) {
        GC gc = e.gc;
        gc.drawRectangle(3, 5, 20, 25);
    }
});
shell.open();
```

The `paintListener` will detect whenever canvas is repainted, and execute the `paintControl` method which draws our rectangle. Note that `shell.open()` occurs after the `PaintListener` is added, because the initial painting is done when the shell is opened. If we only add the `PaintListener` after opening the shell, the rectangle will not be drawn until the next time the canvas must be repainted, ie after the canvas is obscured, or minimized, and then becomes visible again.

Also, notice how we obtained our GC. The `PaintEvent` contains the gc for the object that must be repainted, so we simply access it. There is no need to dispose of this GC because we did not allocate it.

The result of the above code will look like the following screenshot. Minimizing, moving, and resizing and similar operations have no apparent effect on the rectangle we have drawn, because it gets repainted when necessary.



Fonts and Colors

Fonts and Colors are another two important components of graphics in SWT. Both are resources allocated in the OS, so they must be explicitly disposed of, just like GC and Display.

As in the previous example, we will use a `PaintListener` to do our drawing. This time, however, we will draw directly on the shell, just to demonstrate that you don't necessarily have to be using a `Canvas` widget as your drawing surface.

```
shell.addPaintListener(new PaintListener() {
    public void paintControl(PaintEvent e) {
        GC gc = e.gc;
        Display d = e.widget.getDisplay();
```

We use a `Color` constructor to create a new colour, specifying the `Display` and the red, green, and blue values for the colour. Each of these values should be in the range 0 – 255. Since we created the colour, we must dispose of it, so you will see a `dispose` call at the end of the method.

```
        Color color = new Color(d, 120, 42, 105);
        gc.setForeground(color);
```

Similarly, we create a new font using a `Font` constructor, and passing the font name, point size, and font style. The font styles are specified as bitwise ORs of `SWT.BOLD`, `SWT.NORMAL`, and `SWT.ITALIC`. Once the font has been created, we call the `GC` method `setFont`, and pass it our new font. Text that is drawn after that will use our new font. As for `Color`, a `Font` must be disposed of when we are done with it, so we call `font.dispose()`.

```
        Font font = new Font(d, "Arial", 24, SWT.BOLD |
SWT.ITALIC);
        gc.setFont(font);
```

We use `GC`'s `drawString()` method to paint some text to the screen with our new font and color.

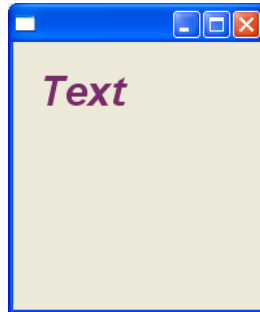
```
        gc.drawString("Text", 20, 20);
```

Again, we dispose of our graphics objects, except `GC` because we didn't create it here.

```
        color.dispose();
        font.dispose();
    }
```

```
});  
shell.open();
```

Notice that we call `shell.open()` after we have added the listener, so that when it is initially drawn, our drawing occurs as well. The result looks like this:



Images

Images are used to allow your program to load, draw on and display images. The types of files we can open include: BMP (Windows Bitmap), ICO (Windows Icon), JPEG, GIF, and PNG. A simple example (`ImageDemo.java`) of using an image is the following, which will load an image from a file, and draw it onto the shell.

First we set up the shell in the standard way:

```
Display display = new Display();  
Shell shell = new Shell(display);  
shell.setSize(140,100);
```

Next we load the image from a file, and open the shell to draw on it. We pass `display` as the `Device` parameter to the `Image` constructor, and a `String` representing the file name and path as the second argument. Notice that we next open the shell, because the shell must be open before we draw to it. (Also, note that we loaded the image before opening the shell. If we swap the ordering of the following 2 lines, we may see a slight delay between when the shell is displayed, and then the image is drawn to it, because loading the image from a file takes some time.)

```
Image img = new Image(display, "EclipseBannerPic.jpg");  
shell.open();
```

Now we acquire a graphics context (or `GC`) for the shell, so that we can draw on it, and then call `drawImage()` to draw the image onto our shell. The first parameter to this version of the `drawImage` function is the image itself, and the remaining 2 parameters are the `x` and `y` coordinates that it is to be drawn to on the shell.

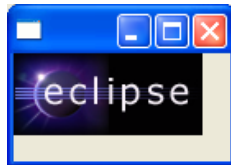
```
GC gc = new GC(shell);
```

```
gc.drawImage(img, 0, 0);
```

After the event loop, there is one more important command.

```
img.dispose();
```

This disposes of the image and frees its resources. This is very crucial, because images are heavyweight objects that require OS resources. Certain operating systems may limit the number of Images available at once, so it is important to release them when you no longer intend to use them. Here is the resulting screen.



Notice that if you minimize or reduce the size of the window, the area of the image obscured by the screen is wiped away, and doesn't come back. This is because we drew the image only once, and didn't provide code to redraw it if the window was altered.

JAR Files

If you intend to package your program and images into a JAR file, there are some modifications you must make to your code to ensure that the images will still load successfully. This is because the images are now inside a JAR file, and the Image constructor can no longer access them directly via the file system. The code you must use to locate and create an image looks like the following:

```
InputStream is =  
getClass().getResourceAsStream("EclipseBannerPic.jpg");  
Image img = new Image(display, is);
```

In the ImageDemo.java code, you can substitute this code for the single line that creates the image, and it should work correctly. If the image is packaged in the JAR file in a location other than at the root level (ie. It's in a subdirectory within the JAR) you will need to specify this path in the getResourceAsStream call as well. Eg. If the image is in an images subdirectory within the JAR, you would need to specify

```
getResourceAsStream("images/EclipseBannerPic.jpg")
```

The first line gets an InputStream from the image file in the JAR. It is just a stream of bytes representing the contents of the file. getClass() returns an object of type Class for the current class. The Class class contains the method getResourceAsStream which is used to load resources, and returns an InputStream.

The `InputStream` returned is then used to construct an `Image` object using an alternate constructor.

This same technique for loading the images can work within Eclipse as well, as long as the image file is located in the same folder as the `.class` files (or some subfolder of that folder). For the demos that follow however, we will just use the basic technique used earlier.

There are a variety of other useful constructors for `Images`.

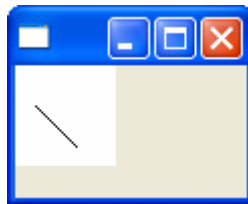
`Image(Device device, int width, int height)` is used to create a blank image with a specified width and height.

`Image(Device device, Rectangle bounds)` has the same purpose, except that the width and height are specified using a `Rectangle` object.

A common use of these blank `Images` is to perform drawing to an `Image` offscreen, and then use `GC.drawImage()` to quickly draw the whole image to the screen. This can prevent flicker since doing drawing directly to the screen can be slow.

In `OffscreenImageDemo.java`, we create a blank `Image`, and a `GC` for it. We then use the `GC` to draw a line onto the image while it is in memory. Then we create a second `GC` for the shell, and use this `GC` to draw the image to the screen.

```
Image img = new Image(display, 50,50);
shell.open();
GC imgGC = new GC(img);
imgGC.drawLine(10, 20, 30, 40);
GC shellGC = new GC(shell);
shellGC.drawImage(img, 0, 0);
```



In this case there likely wouldn't be much difference if we had simply used `shellGC` to draw the line directly to the screen, since drawing a single line would be very quick. However, if we were drawing a complicated figure repeatedly to perform some kind of animation, doing the drawing offscreen could noticeably reduce flicker.

In this example (ImageButtonDemo.java), we demonstrate how to show an image on a button. It is a simple matter of loading/drawing the image, and then call setImage on the button in question, with the image as an argument.

We will also use this example to demonstrate another constructor, used for making duplicates of a given Image.

Image(Device device, Image srcImage, int flag) is used to create a duplicate of srcImage, possibly with some modification specified by flag. flag may be one of SWT.IMAGE_COPY, SWT.IMAGE_GRAY, and SWT.IMAGE_DISABLE.

We load the original image, as usual.

```
Image original = new Image(display, "EclipseBannerPic.jpg");
```

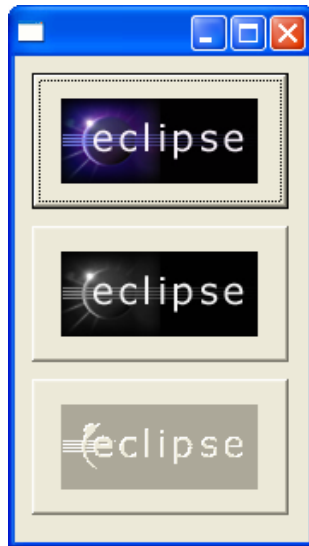
Now we create 3 duplicates of it, one for each possible flag value.

```
Image copy = new Image(display, original, SWT.IMAGE_COPY);  
Image gray = new Image(display, original, SWT.IMAGE_GRAY);  
Image disable = new Image(display, original, SWT.IMAGE_DISABLE);
```

We initialize three buttons(b1, b2, and b3) in the standard way and set their size and locations. (This code is omitted since it is repetitive of something we've already learned.) The next important chunk is actually setting the image on the button. This is done as follows:

```
b1.setImage(copy);  
b2.setImage(gray);  
b3.setImage(disable);
```

The top button (b1) shows an exact copy of the original image. The middle button (b2) displays a gray-scale version of the image. The lowest button (b3) displays a “disabled” version of the image (gray-scale, and somewhat faded.)



Note that if you call `setText` on one of these buttons, the text will replace the Image. The standard `Button` widget cannot display text and an image simultaneously.

Also, notice that it isn't really effective to duplicate an `Image` with `SWT.IMAGE_COPY`, unless you intend to make changes to the resulting image and don't wish to destroy the original `Image` in memory. Otherwise we could simply use the original `Image` in multiple places without penalty. Minimizing the number of images loaded is to our advantage in that it reduces the memory and OS resource requirements of the program. In the example above, the exact duplicate was only created to demonstrate the functionality of the constructor.

We can also add images to items on menus. The code that follows (`ImageMenuDemo.java`) will load "arrow.bmp" and display it beside the text of the action `MenuItem`.

We first create a standard menu with a single choice.

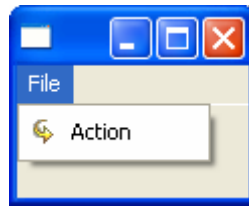
```
Menu bar = new Menu(shell, SWT.BAR);
shell.setMenuBar(bar);
MenuItem file = new MenuItem(bar, SWT.CASCADE);
file.setText("File");
Menu fileMenu = new Menu(shell, SWT.DROP_DOWN);
MenuItem action = new MenuItem(fileMenu, SWT.PUSH);
file.setMenu(fileMenu);
action.setText("Action");
```

Next we load an image, and call `MenuItem`'s `setImage` method.

```
Image icon = new Image(display, "arrow.bmp");
```

```
action.setImage(icon);
```

This is the result.



Setting images (as we've just done for Buttons and MenuItems) can also be performed on a variety of other widgets, such as TreeItems, TableItems, ToolItems and more.

Splash Screens

This example (SplashDemo.java) demonstrates how to produce a simple splash screen effect for your program. The basic technique is to create a Shell with the SWT.NO_TRIM style specified. This produces a shell with no window border around it, which we can place an image in to create the splash screen. We will then just put the running thread to sleep for a few seconds, and then dispose of everything. In a real program you would dispose of the resources and then continue with the real program.

This program differs slightly from the standard ones, so the entire code of the runDemo() method will be listed here.

First we create a display as usual, followed by a shell. The only difference in the shell creation code is that we pass the NO_TRIM style as a parameter to the constructor.

```
Display display = new Display();  
Shell shell = new Shell(display, SWT.NO_TRIM);
```

Next we load the Image that we want to display, and get an ImageData object that contains information about the Image. This is used to size the shell to match the size of the image.

```
Image image = new Image(display, "splash.jpg");  
ImageData imdata = image.getImageData();  
shell.setSize(imdata.width, imdata.height);
```

The next few lines center the new shell on your display. First getBounds() is called, which specifies the size of the screen. We then take the width and height values, and subtract the width and height of our images from each. Then dividing by 2 gives us the correct coordinate-value for the shell to be placed at.

```
Rectangle r = display.getBounds();
int shellX = (r.width - imdata.width) / 2;
int shellY = (r.height - imdata.height) / 2;
shell.setLocation(shellX, shellY);
```

Next we open the shell, and create a graphics context for it. Since we set the shell to be the same size as our image, we can draw the image at coordinates (0, 0) in the shell, and the image will fill the shell completely. As usual drawImage is used to write the image to the shell.

```
shell.open();
GC gc = new GC(shell);
gc.drawImage(image, 0, 0);
```

Once we have displayed the image on the screen, we simply want to pause a moment for the user to look at the splash screen. We create this pause by putting the current thread to sleep for a few seconds. The static method Thread.sleep(int) will do this for us. The parameter is the number of milliseconds that the thread's execution should be suspended for.

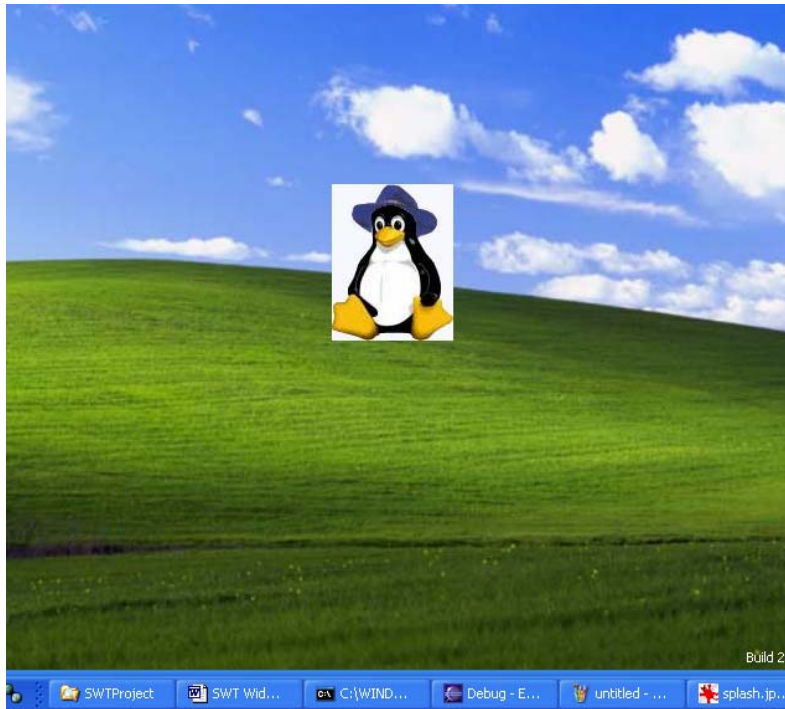
```
final int SLEEP_TIME = 4000;
try {
    Thread.sleep(SLEEP_TIME);
} catch (InterruptedException e) {
}
```

And lastly, we dispose of the resources we used in the standard fashion.

```
image.dispose();
shell.dispose();
display.dispose();
```

When you run the program, the result should look similar to this, with the image “floating” by itself in the center of the screen.

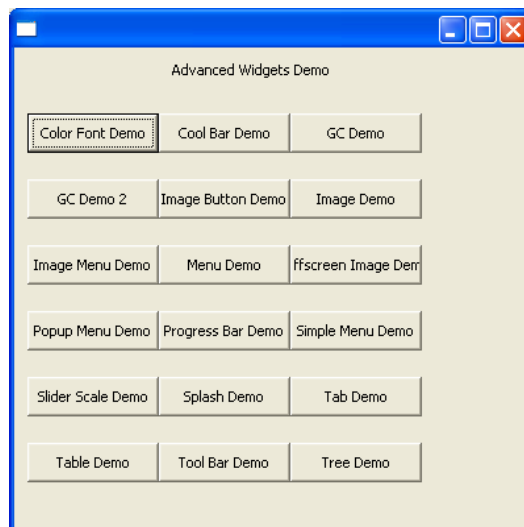
Note that this program simply stops doing anything while the splash screen is displayed. A better use is to display the splash screen while performing any necessary, time-consuming loading tasks. There is a snippet of code available on the Eclipse.org website that demonstrates this technique. It also displays a progress bar that indicates how close the program is to completion of the loading process.



Advanced Widgets Project

A project that contains the examples used in this document is available in a zip file format. As was described at the end of the previous document, 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.

When the project is run, the following shell is displayed. Each example can be run by clicking on the associated button.



MessageBox

A MessageBox is useful to inform or warn the user or to get a basic response from the user. A MessageBox can display a message to the user and provide buttons to allow the user to respond to the system.

To add a message box that asks the user if they like apples and gives them possible replies of "Yes", "No" or "Cancel", use the following code:

```
MessageBox messageBox = new MessageBox(shell, SWT.ICON_QUESTION |  
                                       SWT.YES | SWT.NO | SWT.CANCEL);  
messageBox.setMessage("Do you like apples?");
```

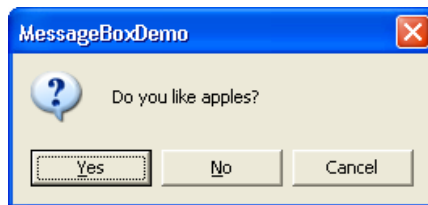
Possible styles for the type of message box are ICON_ERROR, ICON_INFORMATION, ICON_QUESTION, ICON_WARNING and ICON_WORKING. Possible styles for the types of responses are OK, OK | CANCEL, YES | NO, YES | NO | CANCEL, RETRY | CANCEL and ABORT | RETRY | IGNORE. A message box style and a response style can be logically ORed, with the '|' operator, to produce a combination of message boxes. The title of the message box can be set by using the setText() method:

```
messageBox.setText("MessageBoxDemo");
```

To make the dialog visible, use the open() method:

```
response = messageBox.open();
```

The above code will produce the following window:



The open() method of the MessageBox class will return an integer representing the response from the user. The response can be processed as in the following code:

```
switch(response){  
    case SWT.YES:  
        System.out.println("Yes, the user likes apples.");  
        break;  
    case SWT.NO:  
        System.out.println("No, the user does not like apples.");  
        break;  
    case SWT.CANCEL:  
        System.out.println("The user cancelled.");  
        break;  
}
```

ColorDialog

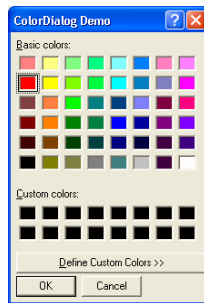
A ColorDialog allows users to select a color from a predefined set of available colors. To add a color dialog to your shell, use the following code:

```
ColorDialog colorDialog1 = new ColorDialog(shell);
colorDialog1.setText("ColorDialog Demo");
colorDialog1.setRGB(new RGB(255,0,0));
```

The above code creates a new instance of the ColorDialog class and adds a title to the dialog by using the setText() method. The setRGB() method can be used to specify the color to be selected when the dialog opens. To make the dialog visible and bring it to the front of the display use the open() method.

```
RGB selectedColor = colorDialog1.open();
```

This will produce the following window:



The open() method will return an object of type RGB representing the color the user selected. It will return null if the dialog was cancelled, if no color was selected or if an error occurred. The color selected can be printed as in the following code:

```
System.out.println("Color Selected:" + selectedColor);
```

DirectoryDialog

A DirectoryDialog allows a user to navigate the file system and select a directory. To add a DirectoryDialog to your shell use the following code:

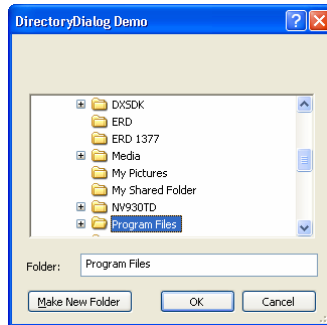
```
DirectoryDialog directoryDialog = new DirectoryDialog(shell);
directoryDialog.setText("DirectoryDialog Demo");
directoryDialog.setFilterPath("C:/Program Files");
```

The above code creates a new instance of the DirectoryDialog class and adds a title to the dialog by using the setText() method. The setFilterPath() method can be used to specify

the directory to be selected when the dialog opens. To make the dialog visible and bring it to the front of the display use the `open()` method.

```
String selectedDirectory = directoryDialog.open();
```

The above code produces the following window:



The `open()` method will return a string describing the absolute path of the selected directory, or null if the dialog was cancelled or an error occurred. The path of the selected directory can be printed as in the following code:

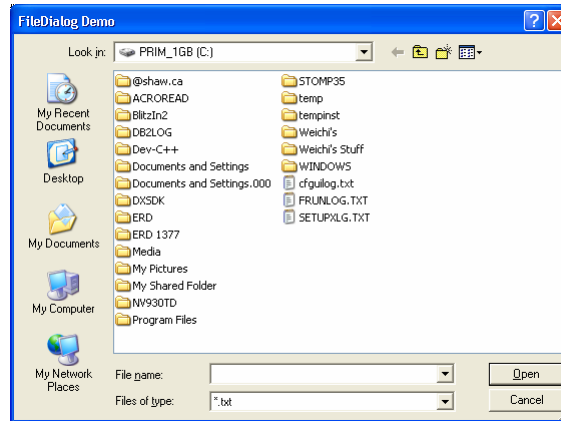
```
System.out.println("Directory Selected:" + selectedDirectory);
```

FileDialog

A `FileDialog` allows a user to navigate the file system and select or enter a file name. To add a `FileDialog` to your shell use the following code:

```
String[] filterExtensions = {"*.txt","*.doc", "*..*"};  
FileDialog fileDialog = new FileDialog(shell, SWT.OPEN);  
fileDialog.setText("FileDialog Demo");  
fileDialog.setFilterPath("C:/");  
fileDialog.setFilterExtensions(filterExtensions);  
String selectedFile = fileDialog.open();
```

This creates a new instance of the `FileDialog` class, as we did in the previous dialog examples, however this time the style of the dialog will be `SWT.OPEN`. Other possible styles are `SWT.SAVE` and `SWT.MULTI`. As in the `DirectoryDialog` example, the title of the dialog can be set with the `setText()` method. The contents of the directory specified in the `setFilterPath()` method will be shown when the dialog is opened. The types of files to be shown can be specified by using the `setFilterExtensions()` method. Calling the `open()` method will produce the following window:



The `open()` method will return a string describing the absolute path of the first selected file, or null if the dialog was cancelled or an error occurred. The absolute path of the selected file can be printed, as in the following code:

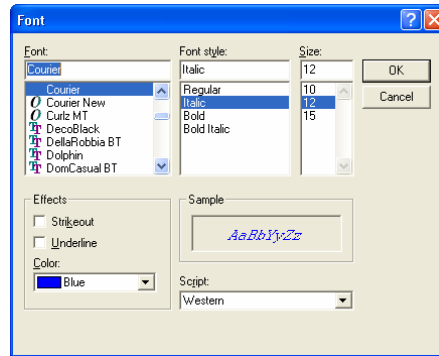
```
System.out.println("File Selected:" + selectedFile);
```

FontDialog

A `FontDialog` allows a user to select a font from all available fonts in the system. To add a `FontDialog` to your shell use the following code:

```
FontData defaultFontData = new FontData("Courier",12,SWT.ITALIC);
FontDialog fontDialog = new FontDialog(shell, SWT.NONE);
fontDialog.setText("FontDialog Demo");
fontDialog.setRGB(new RGB(0,0,255));
fontDialog.setFontData(defaultFontData);
FontData fontData = fontDialog.open();
```

The code creates a new instance of the `FontDialog` class and sets the title of the dialog as in previous dialog examples. The color of the font can be specified by using the `setRGB()` method. The font can also be specified by using the `setFontData()` method, in the above case, the font face, font size and style will be "Courier", "12" and "Italic" respectively. Calling the `open()` method will produce the following window:



The `open()` method will return a `FontData` object describing the font that was selected, or null if the dialog was cancelled or an error occurred. The selected font can be processed as in the following code:

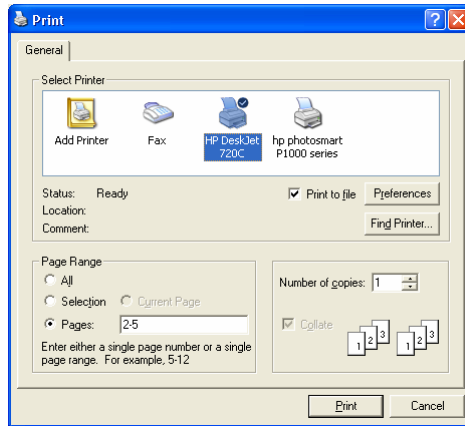
```
if (fontData != null){
    System.out.println("Font Name Selected:"+fontData.getName());
    System.out.println("Font Height Selected:"+fontData.getHeight());
    System.out.println("Font Style Selected:"+fontData.getStyle());
    System.out.println("Font Colour Selected:"+fontDialog.getRGB());
}
```

PrintDialog

A `PrintDialog` allows the user to select a printer and various print-related parameters prior to starting a print job. The add a print dialog to your shell use the following code:

```
PrintDialog printDialog = new PrintDialog(shell, SWT.NONE);
printDialog.setText("PrintDialogDemo");
printDialog.setScope(PrinterData.PAGE_RANGE);
printDialog.setStartPage(2);
printDialog.setEndPage(5);
printDialog.setPrintToFile(true);
PrinterData printerData = printDialog.open();
```

The creates a new instance of the `PrintDialog` class and sets the title as in previous dialog examples. The scope of the print job will initially be set to print a range of pages, starting with second page and ending at the fifth page. The 'print to file' option will also be checked. Calling the `open()` method will produce the following window:



The `open()` method will return a `PrinterData` object containing all the information the user selected from the dialog. If the dialog was cancelled or an error occurred, the `open()` method will return `null`. The print information can be processed as in the following code:

```

if (printerData != null){
    switch(printerData.scope){
        case PrinterData.ALL_PAGES:
            System.out.println("Printing all pages.");
            break;
        case PrinterData.SELECTION:
            System.out.println("Printing selected page.");
            break;
        case PrinterData.PAGE_RANGE:
            System.out.print("Printing page range. ");
            System.out.print("From:"+printerData.startPage);
            System.out.println(" to:"+printerData.endPage);
            break;
    }
    if (printerData.printToFile)
        System.out.println("Printing to file.");
    else
        System.out.println("Not printing to file.");
    if (printerData.collate)
        System.out.println("Collating.");
    else
        System.out.println("Not collating.");
    System.out.println("Number of copies:"+printerData.copyCount);
    System.out.println("Printer Name:"+printerData.name);
}

```

Note that the `PrintDialog` class contains the methods: `getEndPage()`, `getPrintToFile()`, `getScope()` and `getStartPage()` which return their respective fields the user selected before pressing OK or Cancel in the dialog. It is not recommended to use these methods to determine information for a print job as they return values even if the user cancelled the dialog.

Input Dialog

A dialog to accept a text response from the user is not available from SWT, however it is possible to create a class that behaves much like the dialogs that we have seen.

Firstly, we must create a class that will represent the data that is returned from our dialog. We would like to know if the user submitted their response or did they cancel the dialog. If they did not cancel the dialog, we would like to know the user's text response. The following code represents our data class, complete with a constructor, Getter and Setter methods:

```
public class MyInputDialogData {
    String textResponse;
    boolean buttonResponse;
    MyInputDialogData(){
        setTextResponse("");
        setButtonResponse(false);
    }
    public boolean isButtonResponse() {
        return buttonResponse;
    }
    public String getTextResponse() {
        return textResponse;
    }
    public void setButtonResponse(boolean b) {
        buttonResponse = b;
    }
    public void setTextResponse(String string) {
        textResponse = string;
    }
}
```

Next, we must create a class that subclasses the Dialog class. Since this class will be our dialog, we will give this dialog a label to display a message to the user, an area for the user to enter a response, a button for the user to submit the response and another button for the user to cancel the dialog. We will also need a field for the data that will be returned from the dialog.

```
public class MyInputDialog extends Dialog {
    private Label lblQuestion;
    private Text txtResponse;
    private Button btnOkay;
    private Button btnCancel;
    private MyInputDialogData data;
```

We now need a method to display our dialog. This method will be analogous to the open() method of the built in dialogs. In order to have the same behaviour as the built in dialogs we have seen, this method will return an instance of our data class.

```

public MyInputDialogData open () {
    data = new MyInputDialogData();
    final Shell shell = new Shell(getParent(), SWT.DIALOG_TRIM |
SWT.APPLICATION_MODAL);
    shell.setText(getText());
    shell.setSize(300,150);

    lblQuestion = new Label(shell, SWT.NONE);
    lblQuestion.setLocation(25,10);
    lblQuestion.setSize(150,20);
    lblQuestion.setText("What is your name?");

    txtResponse = new Text(shell, SWT.BORDER);
    txtResponse.setLocation(25,30);
    txtResponse.setSize(200,20);

    btnOkay = new Button (shell, SWT.PUSH);
    btnOkay.setText ("Ok");
    btnOkay.setLocation(40,60);
    btnOkay.setSize(55,25);

    btnCancel = new Button (shell, SWT.PUSH);
    btnCancel.setText ("Cancel");
    btnCancel.setLocation(100,60);
    btnCancel.setSize(55,25);

    Listener listener = new Listener () {
        public void handleEvent (Event event) {
            data.setButtonResponse(event.widget == btnOkay);
            data.setTextResponse(txtResponse.getText());
            shell.close ();
        }
    };
    btnOkay.addListener(SWT.Selection, listener);
    btnCancel.addListener(SWT.Selection, listener);

    shell.open();
    Display display = getParent().getDisplay();
    while (!shell.isDisposed()) {
        if (!display.readAndDispatch()) display.sleep();
    }
    return data;
}

```

The preceding code sizes and positions each of the dialog's widgets. A listener is then created to handle when the user clicks on a button. The listener will set the appropriate fields of the MyInputDialogData object. The Dialog will then display itself by calling the open() method of the Shell class.

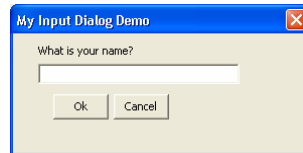
Using our input dialog will be much like using the built in dialogs.

```

MyInputDialog midt = new MyInputDialog(shell);
midt.setText("My Input Dialog Demo");
MyInputDialogData myData = midt.open();

```

The code creates an instance of the MyInputDialog class, sets the text of the window and invokes the open() method. The complete sample code will produce the following window:



The data returned by the open() can be processed as in the following code:

```
if (myData.isButtonResponse())
    System.out.println("The user clicked the 'okay' button.");
else
    System.out.println("The user did not click the 'okay'
button.");
System.out.println("Response: " + myData.getTextResponse());
```

DragAndDrop

In this section, we describe how to implement Drag and Drop and Clipboard to transfer text within an SWT application.

Drag and Drop provides the user with a quick and easy mechanism to transfer data within an application and between applications. Firstly, we must import all the drag and drop classes:

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

To implement Drag and Drop, we must specify which widgets will be our source and destination targets. In the following code, we will specify a label to be our source target. First, create, set and size a label:

```
lblSource = new Label(shell, SWT.BORDER);
lblSource.setSize(200,50);
lblSource.setLocation(50,50);
lblSource.setText("This is the source label");
```

Next, specify the type of transfer we want. Possible types are TextTransfer, RTFTransfer or FileTransfer. We are only going to show how to transfer unformatted text:

```
Transfer[] types = new Transfer[] {TextTransfer.getInstance()};
```

Then we must specify the type of operations we want to enable on the drag source. Possible operations are DROP_MOVE, DROP_COPY and DROP_LINK. We will only concern ourselves with moving and copying text:

```
int operations = DND.DROP_MOVE | DND.DROP_COPY;
```

Create a new instance of the DragSource class, specifying which control will be the source and the operations we want to enable on it and set its transfer type:

```
DragSource source = new DragSource (lblSource, operations);  
source.setTransfer(types);
```

Now we must specify a label to be the target control. First create, set and size a label, then create a new instance of the DragTarget class, indicating which control will be the target, and specifying the operations and transfer types we want enabled.

```
lblTarget = new Label(shell, SWT.BORDER);  
lblTarget.setSize(200,50);  
lblTarget.setLocation(50,150);  
lblTarget.setText("This is the destination label");  
  
DropTarget target = new DropTarget(lblTarget, operations);  
target.setTransfer(types);
```

In order to be notified of when the user starts dragging text we must add a DragSourceAdapter() listener to the source. This is illustrated in the following code:

```
source.addDragListener (new DragSourceAdapter() {  
    public void dragStart(DragSourceEvent event) {  
        if (lblSource.getText().length() == 0) {  
            event.doit = false;  
        }  
    };  
    public void dragSetData (DragSourceEvent event) {  
        if  
(TextTransfer.getInstance().isSupportedType(event.dataType)){  
            event.data = lblSource.getText();  
        }  
    }  
    public void dragFinished(DragSourceEvent event) {  
        if (event.detail == DND.DROP_MOVE)  
            lblSource.setText("");  
    }  
});
```

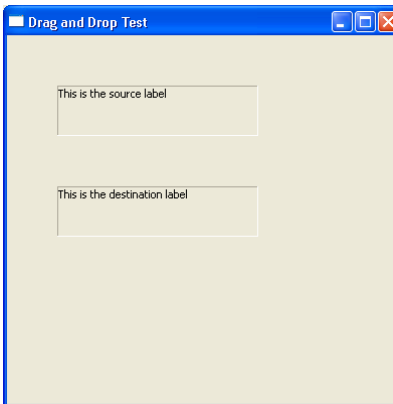
The dragStart() method of the DragSourceAdapter will be executed when the user has begun the actions required to drag the widget. This event gives the application the chance to decide if a drag should be started. The dragSetData() method will be executed when the user drops data on a drop target, and must provide the data to be dropped. This method must also support all the data transfer types that were specified in the setTransfer() method of the drag source. The dragFinished() method will be executed after a drop has been completed successfully or was aborted. This event gives the application the chance to take any appropriate clean up action, for example in a successful move operation the application must remove the data that was transferred.

In order to be notified of when the user drops the text over our target, we must add a `DropTargetAdapter()` to the target. This is illustrated in the following code:

```
target.addDropListener (new DropTargetAdapter() {
    public void drop(DropTargetEvent event) {
        if (event.data == null) {
            event.detail = DND.DROP_NONE;
            return;
        }
        lblTarget.setText ((String) event.data);
    }
});
```

The `drop()` method will be executed when the data is being dropped. The drop target can decide how to handle the dropped data in this method.

The result of the above code will be the following screenshot:



Clicking on the source label and dragging the mouse over the destination label will move the source text to the destination label. To make a copy of the text, hold down the Control key when dropping the text.

Clipboard

The Clipboard provides a mechanism for transferring data from one application to another or within an application. We must first import the drag and drop classes:

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

To get access to the operating system's clipboard, we must create a Clipboard object:

```
cb = new Clipboard(myDisplay);
```

Note: As the clipboard object uses system resources, we must make sure to dispose of the clipboard object when we are done with it. This can be done with the dispose() method at the end of our application.

Next, we need to create two Text widgets to be our source and destination targets. We will also need two Button widgets so we can copy and paste text to and from the clipboard:

```
txtSource = new Text(shell, SWT.BORDER);
txtSource.setSize(200,20);
txtSource.setLocation(50,20);

btnCopy = new Button(shell, SWT.PUSH);
btnCopy.setSize(100,20);
btnCopy.setLocation(275,20);
btnCopy.setText("Copy");

txtDestination = new Text(shell, SWT.BORDER);
txtDestination.setSize(200,20);
txtDestination.setLocation(50,80);

btnPaste = new Button(shell, SWT.PUSH);
btnPaste.setSize(100,20);
btnPaste.setLocation(275,80);
btnPaste.setText("Paste");
```

Now we need to add functionality to our buttons so that they can capture events. We need to add selection listeners to our copy and paste buttons. For the copy button, when it is pressed, we want to transfer the data that is in the source Text to the clipboard. However, if there is no data in the source Text, we will do nothing. This is demonstrated in the following code:

```
btnCopy.addSelectionListener(new SelectionAdapter(){
    public void widgetSelected(SelectionEvent e) {
        String textData = txtSource.getText();
        if (textData == null) return;
        TextTransfer textTransfer = TextTransfer.getInstance();
        Transfer[] types = new Transfer[] {textTransfer};
        cb.setContents(new Object[] {textData}, types);
    }
});
```

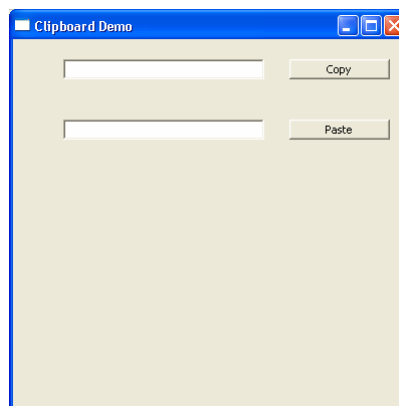
To transfer data to the clipboard, we must first set up the type of transfers we will want to handle. The possible transfer types are TextTransfer and RTFTransfer, however we will only deal with transferring text. To do this, we need to create an array of Transfer containing an instance of the TextTransfer class. Finally, to transfer our data to the clipboard, we use the setContent() method of the Clipboard class.

Next, when the paste button is pressed, we want to transfer the data from the clipboard to the destination Text. If there is no data on the clipboard, we will do nothing. This is demonstrated in the following code:

```
btnPaste.addSelectionListener(new SelectionAdapter(){
    public void widgetSelected(SelectionEvent e){
        TextTransfer textTransfer = TextTransfer.getInstance();
        String data = (String)cb.getContents(textTransfer);
        if (data == null) return;
        txtDestination.insert(data);
    }
});
```

We want to concern ourselves only with the TextTransfer type, so we only need to get an instance of the TextTransfer class. We ask for the contents of the clipboard of type TextTransfer by using the getContents() method. Then, if there is data of type TextTransfer, we insert it into the destination Text.

The sample code will produce the following window:



Data can be copied from the source Text and pasted in the destination Text or in another application, like Notepad. Data can also be copied from another application and pasted into the destination Text of the Demo.