DOCUMENTATION BY DEVELOPER.COM (http://www.developer.com)

SWT Programming with Eclipse

By Koray Guclu

"The best way to predict the future is to invent it."—Alan Kay

1. Why SWT?

SWT is a cross platform GUI developed by IBM. Why has IBM created another GUI? Why have not they used existing Java GUI frameworks? To answer those questions, we need to go back to the early days of Java.

Sun has created a cross platform GUI framework AWT (Abstract Windowing Toolkit). The AWT framework uses native widgets but it was suffering from a LCD problem. The LCD problem causes loss of major platform features. In other words, if platform A has widgets 1–40 and platform B has widgets 20–25, the cross-platform AWT framework only offers the intersection of these two sets.

To solve this problem, Sun has created a new framework that uses emulated widgets instead of native widgets. This approach solved the LCD problem and provided a rich set of widgets but it has created other problems. For instance, Swing applications no longer look like native applications. Although they're the latest improvements in the JVM, Swing applications suffer performance problems that do not exist in their native counterparts. Moreover, Swing applications consume too much memory, which is not suitable for small devices such as PDAs and Mobile Phones.

IBM has decided that neither of the approaches fulfill their requirements. Consequently, IBM has created a new GUI library, called SWT,which solves the problems seen with the AWT and the Swing frameworks. The SWT framework accesses native widgets through JNI. If a widget is not available on the host platform, SWT emulates the unavailable widget.

2. Building Blocks of an SWT Application

Display, Shell, and Widgets are basic building blocks of an SWT application. Displays are responsible from managing event loops and controlling communication between the UI thread and other threads. Shell is the window in an application managed by the OS window manager. Every SWT application requires at least one Display and one or more Shell instances.

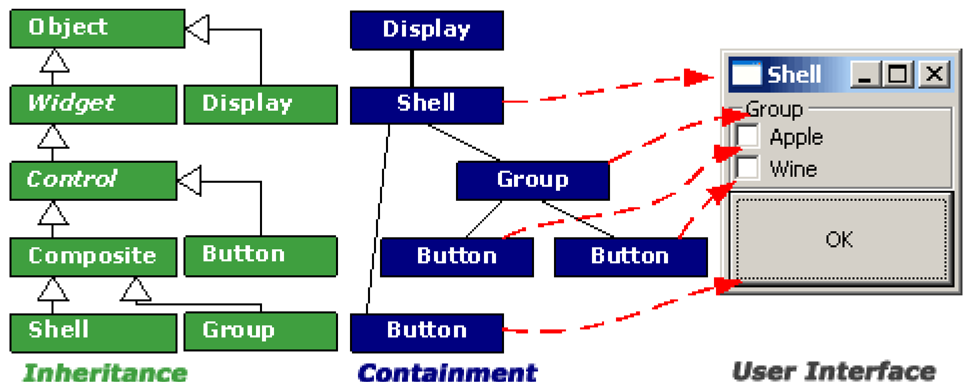


Figure 1. *An SWT application from different perspectives.*

Figure 1 shows an SWT application from different perspectives. The first diagram is the simplified inheritance diagram of the UI objects. The second diagram is the containment structure of the UI objects. The third diagram is the created UI.

If an application uses multiple threads, each thread uses its own instance of a Display object. You can get the current active instance of a Display object by using the static Display.getCurent() method.

A Shell represents a window in a particular operating system. A shell can be maximized, normal, or minimized. There are two types of shells. The first one is the top-level shell that is created as a child, main window of the Display. The second one is a dialog shell that depends on the other shells.

The type of a Shell depends on style bits passed to the Shell's constructor. The default value of a Shell is DialogShell. That is to say, if nothing is given to the parameter, it is by default a DialogShell. If a Display object is given to the parameter, it is a top-level shell.

Some widget properties must be set at creation time. Those widget properties are called *style bits*. Style bits are defined as constants in SWT class, for example, Button button = new Button(shell, <styleBits>). It is possible to use more then one style bit by using the OR operation |. For instance, to use a bordered push button, you need to use SWT.PUSH | SWT.BORDER as style bit parameters.

3. Environment Set-Up

Developing an SWT application is different from developing a Swing application. To begin with an SWT application development, you need add SWT libraries to your classpath and set necessary environment variables accordingly.

The first library that you need is the swt.jar file that is under the ECLIPSE_HOME\eclipse\plugins\org.eclipse.swt.win32_2.1.0\ws\win32 directory. Depending on the version of the Eclipse that you are using, you might need to use a different directory. The swt.jar file must be added to your classpath to make this go to **Project->Properies->JavaBuildPath->Libraries->Add Variable -> Eclipse Home ->Extend** and select the swt.jar library under the director abovey and then click on **OK**.

Afterwards, you will be able to compile an SWT application but, because of a runtime exception shown below, it won't be able to run because the swt.jar library uses native libraries. You need to set the java.library.path environment variable to use native libraries in Java.

Console output

```
java.lang.UnsatisfiedLinkError: no swt-win32-2133 in java.library.path
at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1403)
at java.lang.Runtime.loadLibrary0(Runtime.java:788)
at java.lang.System.loadLibrary(System.java:832)
...
at org.eclipse.swt.widgets.Display.<init>(Display.java:287)
at Main.main(Main.java:25)
Exception in thread "main"
```

To set the java.library.path variable, go to **Run-> Run...-> Java Applicaton-> New ->Arguments -> VM Arguments**. Thereafter, if neccesary, modify the path below and paste it to the **VM Arguments** field.
-Djava.library.path=c:\eclipse\plugins\org.eclipse.swt.win32_2.1.0\os\win32\x86

Loading native libraries

If you need to load any native library that your application uses, you can use the Runtime.getPlatform.loadLibrary("libraryname") method.

Finishing these steps will enable you to run an SWT application within your eclipse environment.

4. Your First SWT Application

Creating a typical SWT application requires the following steps:

- Create a Display
- Create one or more Shells
- Set a Layout manager of the Shell
- Create widgets inside the Shells
- Open the Shell window
- Write an event dispatching loop
- Dispose display

You can use the following code template to quickly run the code snippets in this article. You can copy and paste the code to the area, as shown in *Source 1*.

Source 1. SWT application template

```
import
org.eclipse.swt.layout.RowLayout;
import
org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class SliderExample
{
  public static void main(String
args[])
  {
1  Display display = new Display();
2  Shell shell = new Shell(display);
3  shell.setLayout( new
RowLayout());
    // ------------------------
    // Your code comes to here.
    // ------------------------
    shell.pack();
4  shell.open();
5  while( !shell.isDisposed())
    {
      if(!display.readAndDispatch())
      display.sleep();
    }
6  display.dispose();
  }
}
```

This example displays an empty window. You can add your widgets to the template above. Every SWT application requires a Display and one or more Shells. The Shell is a composite object; it can contain other composite objects. If the layout of the shell is not set, added widgets to the Shell won't be visible. The Shell window must be opened to be displayed. The event handling loop reads and dispatches GUI events. If there is no event handling loop, the application window cannot be shown, even if the Shell window is opened by the open() method. Afterwards, you should dispose of the Display, when the Shell is discarded.

tip *Importing required libraries*

You can use the **Source->Organize Imports** menu or **Ctrl+Shift+O** to import the required libraries automatically.

5. Running SWT Applications Outside of Eclipse

To run the application without using Eclipse, the swt.jar library must be in your classpath, and the java.library.path enviroment variable must be set properly. Depending on the host platform, the appropriate native library file must be available. For the Windows platform, you can do the following to make the native library configuration for your application:

1. Put swt.dll in the same directory as the program.
2. Put swt.dll in the JAVA_HOME\bin\ directory.
3. Put swt.dll in the c:\windows\system32 directory.

javac classpath c:\swt\swt.jar HelloWorld.java
Java classpath c:\swt\swt.jar;. Djava.library.path=c:\swt HelloWorld
The java.library.path is the required environment variable for the JNI. If you don't set this environment, your DLL class is not accessible. In that case, the application cannot run properly and throws an exception.

SWT libraries

Swt libraries are available under the Eclipse plug-in directory. If you want to get the SWT libraries without downloading the whole Eclipse package, it is possible to download a single SWT library under the http://www.eclipse.org/downloads directory.

6. SWT Packages

The SWT consists of the following main packages. Definitions of the packages are taken from the Eclipse API documentation. You can find the API documentation on the Eclipse Web site.
org.eclipse.swt: contains classes that define constants and exceptions that are used by SWT classes. An SWT package consists of three classes: SWT, SWTException and, SWTError. The SWT will probably be your favorite class because it contains constants for SWT libraries such as keyboard, error, color, layout, text style, button etc. constants.
org.eclipse.swt.widgets: Contains most core SWT widget components, including the support interface and classes.
org.eclipse.swt.events: Defines typed events, listeners and events, that SWT components use. This package contains three different groups of classes: Listener interfaces, Adapter classes, and Event class.
org.eclipse.swt.dnd: Contains classes for drag-and-drop support of the SWT widgets.
org.eclipse.swt.layout: Contains classes providing automated positioning and sizing of the SWT widgets.
org.eclipse.swt.print: Contains classes providing print support for the SWT widgets.
org.eclipse.swt.graphics: Package provides the classes which implement points, rectangles,

regions, colors, cursors, fonts, graphics contexts (that is, GCs) where most of the primitive drawing operations are implemented, and images including both the code for displaying them and the public API for loading/saving them.

7. Dialogs

Dialog implementations are native. That is to say, dialogs, like widgets, are platform components. Dialogs in SWT are derived from the Dialog abstract class. A dialog is not a widget but it can contain widgets.

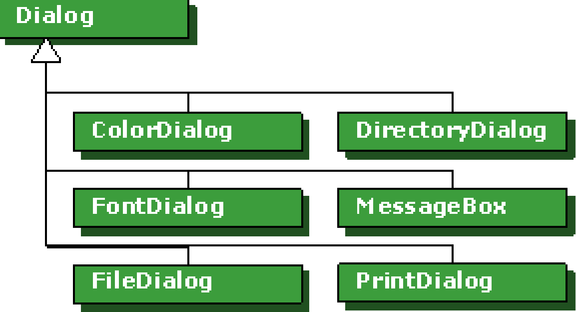


Figure 2. *Dialog class hierharcy.*

There are different types of dialogs available. Some dialogs can have specific properties. A a dialog can be used as shown in *Source 2*.

Source 2. `MessageBox` example

```
MessageBox messageBox =
  new MessageBox(shell, SWT.OK|SWT.CANCEL);
if (messageBox.open() == SWT.OK)
{
  System.out.println("Ok is pressed.");
}
```

Each dialog's open() method returns different types. For instance, the MessageBox dialog returns int from the open() metod. Therefore, one must write different conditions to handle the return value for each dialog.

ColorDialog shows a color selection pallet. It returns an RGB object from return method.

DirectoryDialog enables you to select a directory. It returns A String from THE open() method. The returning value is the selected directory. It is also possible to set additional filters to filter the directories.

The Font dialog enables a user to select a font from all available fonts in the system. It returns a FontData object from the open() method.

FileDialog enables a user to select a file. Additionally, you can set an extension filter, path filter, and filename filters. This dialog has the styles shown in *Table 1*:

Table 1. SWT Dialog style bit constants

SWT.OPEN	Shows Open button in the dialog
SWT.SAVE	Shows Save button in the dialog

PrintDialog enables a user to select a printer before starting a print job. It returns a Printer Data object from the open() method.

The MessageBox dialog is used to give feedback to the user. You can combine different styles by the | operation as shown in *Source 3*.

Source 3. `MessageBox` example

```
MessageBox messageBox =
  new MessageBox(shell,
   SWT.OK|
   SWT.CANCEL|
   SWT.ICON_WARNING);

messageBox.setMessage("www.korayguclu.de");
 messageBox.open();
```

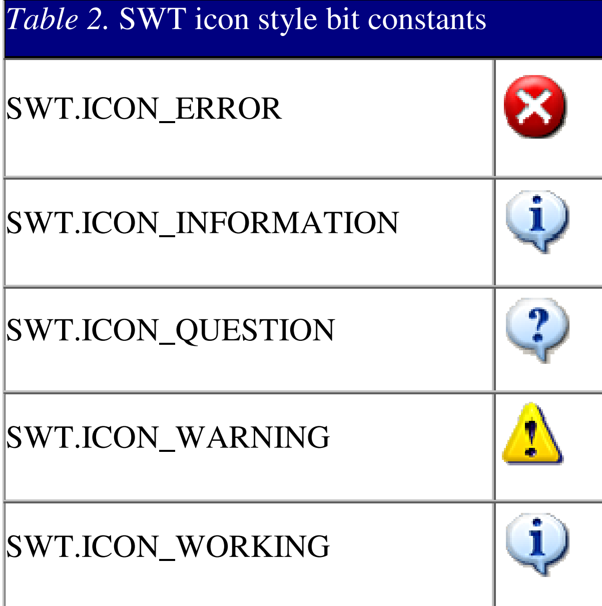
The resulting message box of the above example is shown in *Figure 3*.

Figure 3. *MessageBox dialog.*

Available button constants are listed below. A combination of different buttons can be made by using the | operation. The SWT framework builds the dialog depending on the style bits. The button constants are: SWT.ABORT, SWT.OK, SWT.CANCEL, SWT.RETRY, SWT.IGNORE, SWT.YES, and SWT.NO.

Table 2 shows a list of available icons to be used by dialogs.

Table 2. SWT icon style bit constants



Constant	Icon
SWT.ICON_ERROR	
SWT.ICON_INFORMATION	
SWT.ICON_QUESTION	
SWT.ICON_WARNING	
SWT.ICON_WORKING	

8. Widgets

The SWT GUI objects derived from Widget and Control classes. The Widget object is the base class and defines methods common to all GUI classes. The Control class is the base class of all the windowed GUI classes, which means that the components derived from Control require a window or dialog to be displayed.

Menu objects also require a window to be displayed, but this requirement is indirectly satisfied. A Menu object requires a Control object.

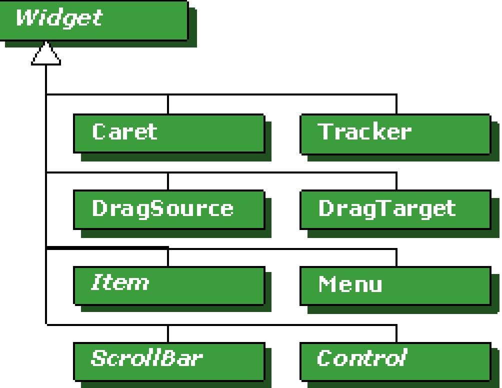


Figure 4. *Widget class hierarchy.*

Figure 4 shows the widget class hierarchy. The Widget, Item, ScrollBar, and Control classes are abstract classes.

8.1. Widget Events

Widget events are summarized in *Table 3*. For the sake of simplicity, the table contains only the event names. It is easy to figure out the name of an event class by using this **<EventName>Event**. Likewise, the name of the associated listener can be figured out by using **<Listener Name>Listener**. Each event does not have a matching adaptor class. For that reason, events having adaptors are marked in bold. The name of an adaptor can be figured out by using **<EventName>Adaptor**.
Examples:
Event Name is a Control, event class is a ControlEvent, listener class is a ControlListener, adaptor class is a ControlAdaptor.

Table 3. SWT Events

Event Name	*Widgets*	*Generated When*
Arm	MenuItem	a menu item is highlighted
Control	Control, TableColumn, Tracker	a control is resized or moved
Dispose	Widget	a widget is disposed
Focus	Control	a control gains or loses focus
Help	Control, Menu, MenuItem	the user requests help (e.g. by pressing the `F1` key)
Key	Control	a key is pressed or released when the control has keyboard focus

Menu	Menu	a menu is hidden or shown
Modify	Combo, Text	a widget's text is modified
Mouse	Control	the mouse is pressed, released, or double-clicked over the control
MouseMove	Control	the mouse moves over the control
MouseTrack	Control	the mouse enters, leaves, or hovers over the control
Paint	Control	a control needs to be repainted
Selection	Button, Combo, CoolItem, List, MenuItem, Sash, Scale, ScrollBar, Slider, StyledText, TabFolder, Table, TableColumn, TableTree, Text, ToolItem, Tree	an item is selected in the control
Shell	Shell	the shell is minimized, maximized, activated, deactivated, or closed
Traverse	Control	the control is traversed (tabbed)
Tree	Tree, TableTree	a tree item is collapsed or expanded
Verify	Text, StyledText	a widget's text is about to be modified

8.2. Useful widgets

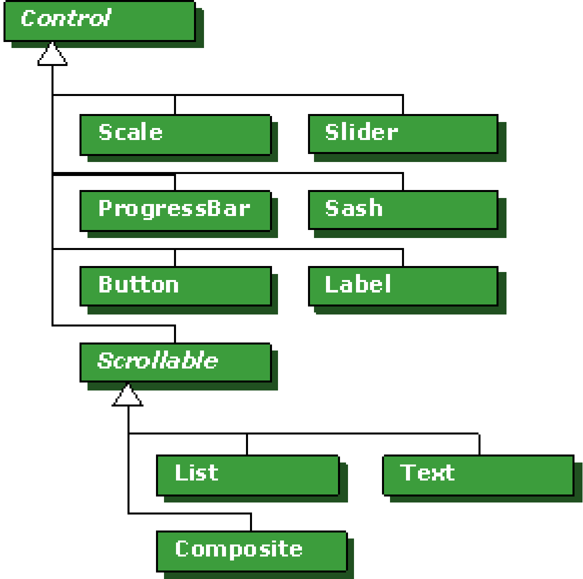


Figure 5. *Control class hierarchy.*

All Control classes can be bordered. You can add a border to a control class by using the SWT.BORDER constant.

tip *SWT style constant*

It is required to specify a style constant (style bit). If you do not know which constant to use or if you do not want to specify it, you can use SWT.NULL.

8.2.1. Buttons

A button can have different styles. The style of a button depends on its defined style bit. A list of buttons and their style constants is shown in *Table 4*.

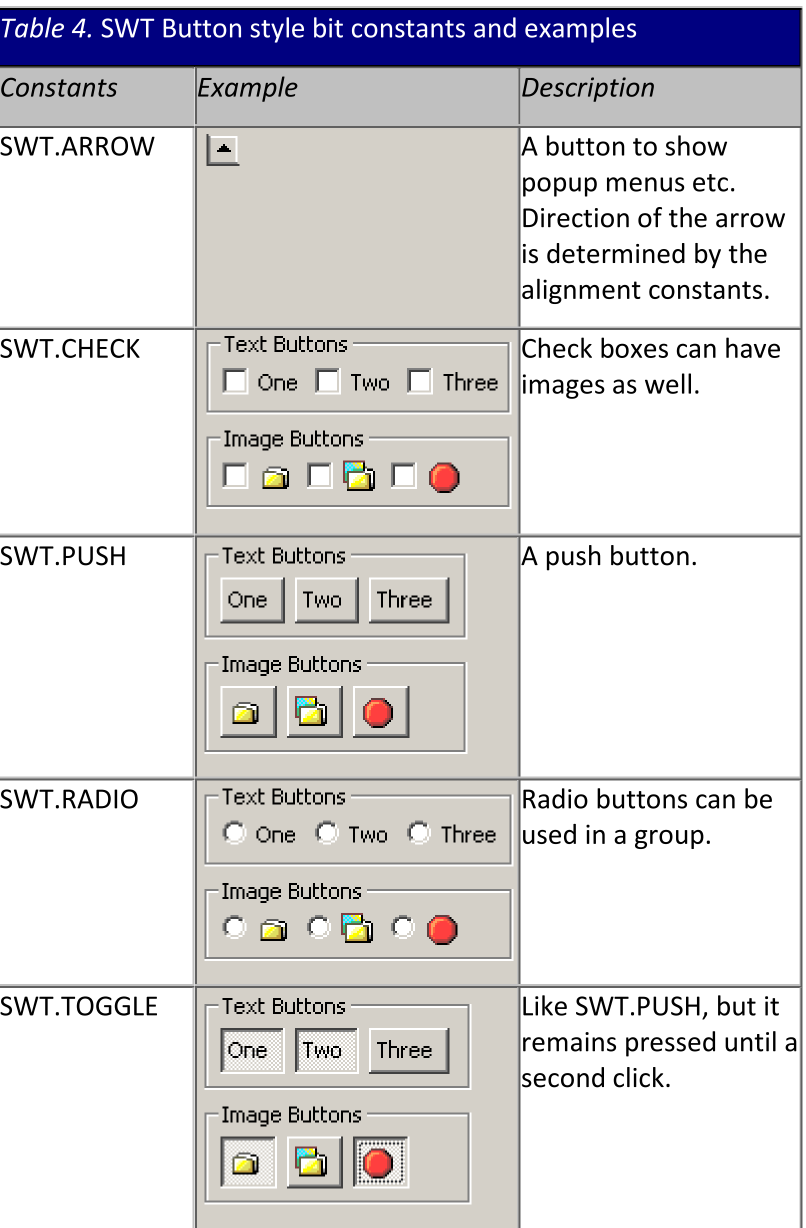


Table 4. SWT Button style bit constants and examples

Constants	*Example*	*Description*
SWT.ARROW		A button to show popup menus etc. Direction of the arrow is determined by the alignment constants.
SWT.CHECK	Text Buttons: One, Two, Three; Image Buttons	Check boxes can have images as well.
SWT.PUSH	Text Buttons: One, Two, Three; Image Buttons	A push button.
SWT.RADIO	Text Buttons: One, Two, Three; Image Buttons	Radio buttons can be used in a group.
SWT.TOGGLE	Text Buttons: One, Two, Three; Image Buttons	Like SWT.PUSH, but it remains pressed until a second click.

8.2.2. Slider, Scale, and ProgressBar widgets

Scale represents a range of selectable continues values. The range can be specified by the setMinimum() and setMaximum() methods of the Scale class. You can get the selection value by using the getSelection() method. A scale can only have one selected value at a given time. That is to say, it is not possible to have multiple selections.

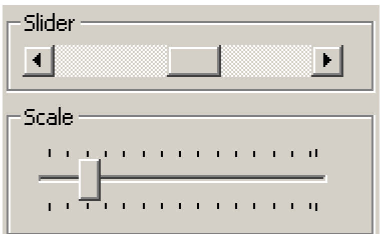


Figure 6. *Slider and Scale widgets inside a Group.*

Depending on the parameters passed to the constractor, it is possible to create different scale and slider widgets. The slider and scale constants are shown in *Table 5*.

Table 5. SWT slider & scale style bar constants	
SWT.HORIZONTAL SWT.VERTICAL	Shows horizontal or vertical widget

Optionally, you can use the SWT.BORDER constant to create a border around a scale widget. This constant has no effect on the slider widget.

Source 4. `Slider` widget example

```
 final Slider slider =
   new Slider(shell,SWT.VERTICAL);
 slider.setMinimum(0);
 slider.setMaximum(100);
 slider.setIncrement(5);
 slider.setPageIncrement(10);
 slider.setSelection(25);
 slider.addSelectionListener(
  new SelectionAdapter()
  {
    public void
widgetSelected(SelectionEvent e)
    {

System.out.println("Selection:"+

slider.getSelection());
    }
  }
 );
```

The ProgressBar widget is similar to the Slider and Scale widgets, but it is not selectable. It shows the progress of a task. You can use the SWT.SMOOTH and SWT.INTERMINATE constants with the ProgressBar widget.

8.2.3. Text widget

A Text widget can be used to show or to edit text. Alternatively, you can use a StyledText widget to display the text by using a different font and color. The StyledText widget allows you to set the foreground, background color, and font for a given range within a text block.

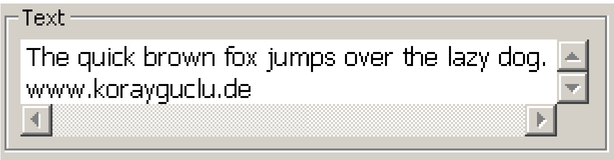


Figure 7. *Text widget.*

It is possible to create a Text widget with the constants shown in *Table 6*. A Text widget is a scrollable widget. Therefore, the SWT.H_SCROLL and SWT.V_SCROLL constants can be used to add scroll bars to a Text widget.

Table 6. SWT Text style bit constants	
SWT.MULTI SWT.SINGLE	Shows a single line or multi line widget
SWT.READ_ONLY	Creates a read-only widget
SWT.WRAP	Wraps the text

Source 5 is a simple Text widget usage example.

Source 5. `Text` widget example

```
Text text =
  new Text(shell, SWT.MULTI|SWT.WRAP);
```

8.2.4. List widget

A List widget can be used to display a list of selectable string values. In the case of a selection, the List object sends a notification event to its listeners. The type of a selection can be single or multiple selections. The type of the selection is determined by the SWT.SINGLE or SWT.MULTI constants. The List widget is a scrollable widget. Therefore, the SWT.H_SCROLL and SWT.V_SCROLL constants can be used to add scroll bars to a Text widget.

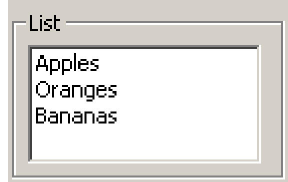


Figure 8. *List widget.*

The following code snippet shows a simple List widget example.

Source 6. `List` example

```
 final List list = new
List(shell,SWT.MULTI);
 for (int i = 1; i < 11; i++)
 {
   list.add(i+".)www.korayguclu.de");
 }
 list.addSelectionListener(
   new SelectionAdapter()
   {
     public void
widgetSelected(SelectionEvent e)
     {
       List list = (List)
e.getSource();
       String[] str =
list.getSelection();
       for (int i = 0; i <
str.length; i++)
       {

System.out.println("Selection:
"+str[i]);
       }
     }
  }
 );
```

8.2.5. Sash widget

A Sash widget can be used to display composite widgets in resizable areas. The following figure shows a Sash widget example.

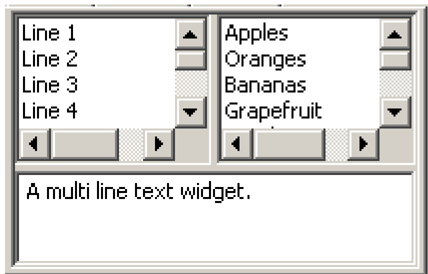


Figure 9. *Sash widget.*

A basic sash example can be seen below.

Source 7. `Sash` example

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

Sash sash = new Sash(shell, SWT.VERTICAL);

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

8.3. Composite Widgets

Composite widgets can contain other composite widgets. The Composite class is the parent class of the composite widgets.

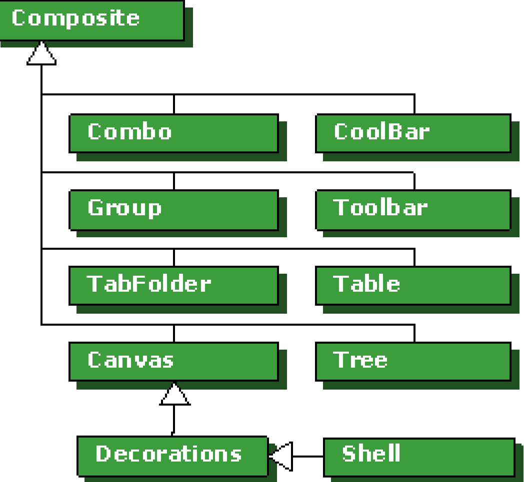


Figure 10. *Composite can contain other composite classes.*

Composite classes can contain other composite classes. This containment is built up by using the constructor of a composite widget class. In contrast to Swing, there is no add() method; instead, you must use constructors to build up a containment structure.

As can be seen from *Figure 10*, the Shell class is also a composite class. That is to say, the Shell object can contain other composite classes.

Composite widgets are Scrollable, which means that it is possible to add scrolls to the composite widgets by using the SWT.H_SCROLL and SWT.V_SCROLL constants.

8.3.1. Table widget

A Table widget can display a set of String items or Images. In contrast to other composite widgets, it is not possible to add composite controls to the table widget. A sub component of a table widget must be of the TableItem type.

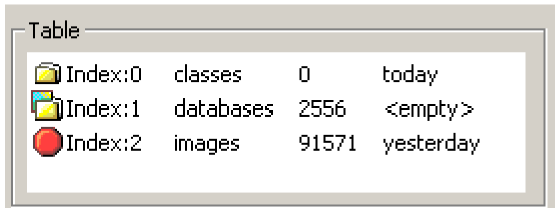


Figure 11. *Table widget.*

The constants in *Table 7* can be used with the table widget.

Table 7. SWT Table style bit constants	
SWT.MULTI SWT.SINGLE	Enables a single or multi selection
SWT.FULL_SELECTION	Enables full row selection
SWT.CHECK	Displays a check box at the beginning of each row

The code snippet in *Source 8* shows a table widget usage containing two columns.

Source 8. `Table` widget example

```
final Table table =
  new Table(shell,SWT.SINGLE);
TableColumn col1  =
  new TableColumn(table,SWT.LEFT);
col1.setText("Coloumn 1");
col1.setWidth(80);
TableColumn col2  =
  new TableColumn(table,SWT.LEFT);
col2.setText("Coloumn 2");
col2.setWidth(80);

TableItem item1 = new TableItem(table,0);
item1.setText(new String[]{"a","b"});
TableItem item2 = new TableItem(table,0);
item2.setText(new String[]{"a","b"});

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

8.3.2. Combo widget

A Combo widget allows users to select a value from a list of values or optionally enter a new value. The combo widget is similar to the List widget, but it uses a limited space.

Although the combo widget is a composite, it does not make sense to add child elements to it. Its elements must be of the String type. An element to a combo widget can be added by using the add(String element) method defined in the combo class.

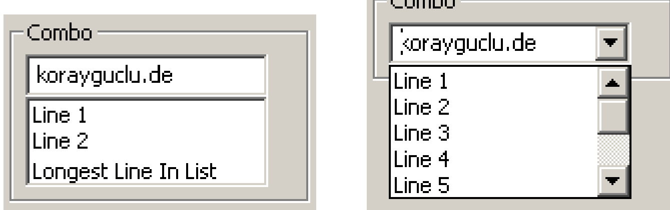


Figure 12. *Combo boxes in different styles.*

The following SWT constants can be used with the Combo widget.

Table 8. SWT Combo style bit constants	
SWT.DROP_DOWN	Drop-down combo box
SWT.READ_ONLY	Read-only combo box
SWT.SIMPLE	Simple combo box (not drop-down combo box). See Figure 11.

The following example shows a Combo widget's usage.

Source 9. `Combo` example

```
final Combo combo =
  new Combo(shell,SWT.DROP_DOWN);
for (int i = 1; i < 11; i++)
{
  combo.add(i+".) element ");
}
combo.setText("Text");
combo.addSelectionListener(
 new SelectionAdapter()
 {
   public void widgetSelected(SelectionEvent e)
   {
     System.out.println("Selection:"+
                         combo.getText());
   }
 }
);
```

8.3.3. Tree widget

A Tree widget represents a selectable hierarchy of items in a tree. Altough the Three class is a composite, it is not allowed to add composite classes to the Three class. Sub items of a Tree class must be of the ThreeItem type.

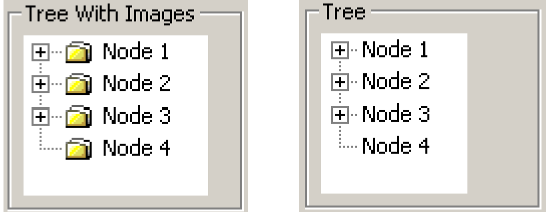


Figure 13. *Tree widgets in different styles.*

The following table shows a list of Tree widget constants.

Table 9. SWT Combo style bit constants	
SWT.SINGLE SWT.MULTI	Allows single or multiple selections
SWT.CHECK	shows a check box at the begining of each node

A simple Tree widget example is shown below.

Source 10. `Tree` example

```
final Tree tree =
  new Tree(shell,SWT.SINGLE);
for (int i = 1; i < 11; i++)
{
  final TreeItem item1  =
   new TreeItem(tree,SWT.NULL);
  item1.setText("node "+i);
  for (int j = 1; j < 6; j++)
  {
    final TreeItem item11 =
     new TreeItem(item1,SWT.NULL);
    item11.setText("node "+i+"."+j);
  }
}

tree.addSelectionListener(
 new SelectionAdapter()
 {
  public void widgetSelected(SelectionEvent e)
  {
    System.out.println("Selection:"+
                       tree.getSelection()[0]);
  }
 }
);
```

8.3.4. TabFolder

The TabFolder widget allows users to select a page from a set of pages. Although it is a composite, it is not allowed to add composite widgets. A widget that is to be added to a TabFolder must of the TabItem type. The content of a tab can be set by using the TabItem's setControl(Control control) method.

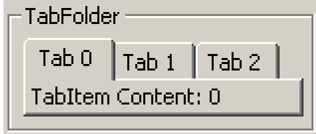


Figure 14. *TabFolder widget*

A simple TabFolder example is shown below.

Source 11. `TabFolder` example

```
final TabFolder tabFolder = new TabFolder( shell, SWT.BORDER);
for (int i=1; i<5; i++){
  // create a TabItem
  TabItem item = new TabItem( tabFolder, SWT.NULL);
  item.setText( "TabItem " + i);
  // create a control
  Label label = new Label( tabFolder, SWT.BORDER);
  label.setText( "Page " + i);
  // add a control to the TabItem
  item.setControl( label );
}
```

8.3.5. CoolBar widget

The CoolBar widget provides an area in which you add items on a dynamically positionable space.You can add one or more ToolBar widgets to a CoolBar. A CoolBar can contain one or more CoolItems. Although CoolBar is a composite widget, it is not allowed to add other composite classes. Sub elements of a CoolBar must be of the CoolItem type.

Figure 15. *Coolbar widget*

The following example shows a CoolBar widget's usage.

Source 12. `CoolBar` example

```
CoolBar coolBar =
  new CoolBar(shell, SWT.BORDER);
coolBar.setLayoutData(
  new FillLayout());
// create a tool bar which it
// the control of the coolItem
for (int k = 1; k <3; k++)
{
  ToolBar toolBar =
    new ToolBar(coolBar, SWT.FLAT);
  for (int i = 1; i < 5; i++)
  {
    ToolItem item =
      new ToolItem(toolBar, SWT.NULL);
    item.setText("B"+k+"."+i);
  }
  // Add a coolItem to a coolBar
  CoolItem coolItem =
    new CoolItem(coolBar, SWT.NULL);
  // set the control of the coolItem
  coolItem.setControl(toolBar);
  // You have to specify the size
  Point size =
    toolBar.computeSize( SWT.DEFAULT,
                         SWT.DEFAULT);
  Point coolSize =
    coolItem.computeSize (size.x, size.y);
  coolItem.setSize(coolSize);
}
```

8.4. A summary of controls having items

Some controls accept sub components as items. For example, a Composite component accepts composite components. Some components need only items. Such components are listed in *Table 10*.

Table 10. Components having items

Widget	Item	Description
CoolBar	CoolItem	Items are selectable, dynamically positionable areas of a CoolBar
Menu	MenuItem	Items are selections under a menu
TabFolder	TabItem	Items are Tabs in a TabFolder
Table	TableItem TableColumn	Items are Rows in a table
ToolBar	ToolItem	Items are Buttons on the tool bar
Tree	TreeItem	Items are nodes in a tree

Conclusion

SWT is the core of the Eclipse user interface. The Eclipse platform is based on the SWT library. To extend your SWT knowledge, you can download SWT examples from the SWT Web site.

Resources

Links

- *A First Look at Eclipse Plug-In Programming,* by Koray Guclu.
- Eclipse uses the Common Public License: http://www.eclipse.org/legal/cpl-v10.html
- Eclipse download site: http://www.eclipse.org/downloads/
- Articles published on the Eclipse Web site: http://www.eclipse.org/arcticles/
- Java download Web site: http://java.sun.com/downloads/index.html
- Eclipse user interface guidelines: http://www.eclipse.org/articles/Article-UI-Guidelines/Index.html
- Quality Eclipse: http://www.qualityeclipse.org/
- SWT Standard Widget Toolkit: http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html

Books

- Eric Clayberg, Dan Rubel, 2004. *Eclipse: Building Commercial-Quality Plug-Ins*, Pearson Education; June 25, 2004.
- David Gallardo, Ed Burnett, Robert McGovern, 2003. *Eclipse in Action: A Guide for Java Developers*, Manning Publications.
- Sherry Shavor, Jim D'Anjou, Scott Fairbrother (2003) *The Java Developer's Guide to Eclipse*, Addison-Wesley Professional.

Constructing SWT Layouts

By Koray Guclu

"I saw the angel in the marble and carved until I set him free."—Michelangelo

Learning layout managers takes time, but once you get accustomed to using them, you can create good looking user interfaces. You also can use GUI builders to create the GUI easily. I use SWT (Standard Widget Toolkit), a cross platform GUI developed by IBM and part of the Eclipse environment. If you are not familiar with this tool please see my earlier article on programming with SWT.

I prefer to use the GUI builders to create an initial look and I configure the UI manually. If you do not have a good understanding of the layout internals, you will limit yourself to the capabilities of the GUI builder that you are using.

Layouts

A layout automatically controls the position and size of widgets inside a Composite. In SWT, the size of a widget inside a composite is not automatically set. Each composite requires a layout to show its children controls. If the layout is not set, SWT will not be able to set the size and position of the controls inside a composite and our controls will not be visible. We have to set a layout for each composite in order to display the children controls.

Layout Manager classes are inherited from an abstract Layout class. Some Layout Managers allow us to set different properties for each control. Those layout classes have an addition object which can be set separately for each control inside a composite. The Control class provides a standard method, setLayoutData(Object obj), to set this addition parameter. You have to set an appropriate layout data object for each control; otherwise, it throws a classcast exception.

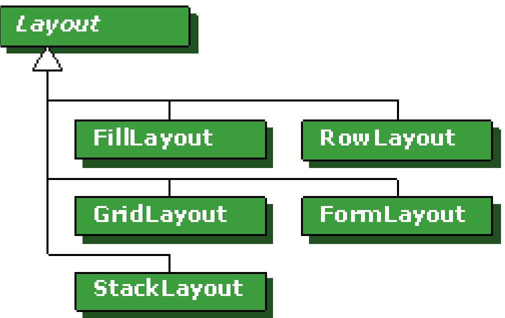


Figure 1. *SWT Layout managers.*

Layout classes have properties that affect all the components within a composite. Some layout classes support layout data objects to set different properties for each control within a composite. Layout properties can be set by using a layout's member variables. If you are familiar with Swing, you would probably search for methods to set and get these member variables. In SWT, this is different; you read and write to those variables directly.

You can use the following SWT application template to test the code snippets shown in this article.

Listing 1. SWT application template

```
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class SWTTemplate
{
  public static void main(String args[])
  {
    Display display = new Display();
    Shell shell = new Shell(display);
    // ------------------------
    // Your code comes to here.
    // ------------------------
    shell.pack();
    shell.open();
    while( !shell.isDisposed())
    {
      if(!display.readAndDispatch())
      display.sleep();
    }
    display.dispose();
  }
}
```

tip *Running SWT applications*

If you are new to SWT programming, you can read my previous article "SWT Programming with Eclipse" from http://www.developer.com/java/other/article.php/3330861.

FillLayout

FillLayout is the simplest layout. If there is only one subcomponent, it fills the all available parent area. If there are more than one component, it forces all components to be the same size in a single row or a column. The width and hight of the subcomponents are determined by the widest or the highest widget in a composite. There are no options available to control either the spacing, margins, or wrapping.

The type variable specifies how controls will be positioned within the composite. The type variable can be set to SWT.HORIZONTAL (the default) or SWT.VERTICAL to position the controls either in a single row or a column. This variable is public; you can either directly set the variable or pass the variable to the constructor.

Listing 2. FillLayout example

```
  FillLayout fillLayout = new
FillLayout(SWT.HORIZONTAL);
  // or ..
  FillLayout fillLayout = new FillLayout();
  fillLayout.type = SWT.HORIZONTAL;
```

The FillLayout might be used in a task bar, in a tool bar, or in a group or composite having only one child. Besides that, it might be used to stack the check boxes or radio buttons in a group.

Listing 3. FillLayout example

```
  // ...
  FillLayout fillLayout = new
FillLayout(SWT.VERTICAL);
  shell.setLayout(fillLayout);
  for (int i = 0; i < 3; i++)
  {
    Button button =new Button(shell, SWT.PUSH);
    button.setText("A"+i);
  }
  // ...
```

The width and height of a control within a composite are determined by the parent composite. Initially, the height of any control is equal to the highest, and the width of a control is equal to the widest.

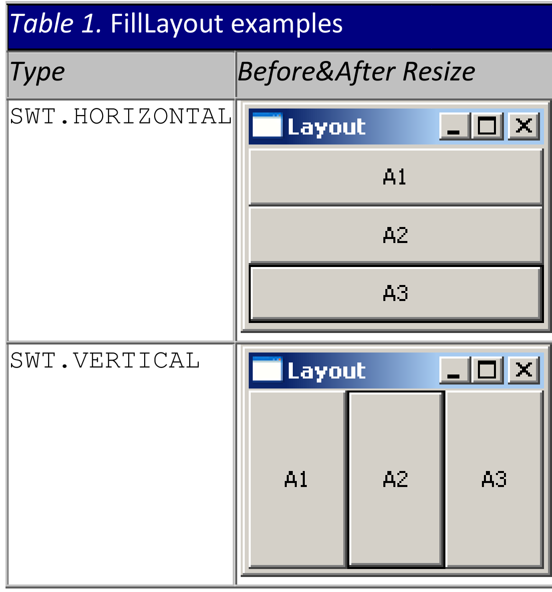


Table 1. FillLayout examples

Type	*Before&After Resize*
SWT.HORIZONTAL	Layout: A1, A2, A3
SWT.VERTICAL	Layout: A1, A2, A3

Advantages:

- Simplest layout
- Positions the components in rows or columns

RowLayout

RowLayout is very similar to FillLayout. It positions the controls, similar to FillLayout, in rows or columns. In addition to that, RowLayout provides configuration fields to control the position of a control within a composite as seen in *Listing 4*.

Figure 2 shows a simple use of RowLayout. You can use the preceding code to create the following example. All values are default; nothing is changed. As you can see, it wrapped the row automatically when there is not enough space left in the row.

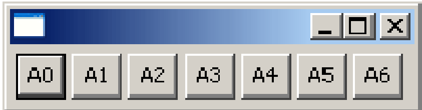


Figure 2. *RowLayout Example with default properties*

If you do not want to make any configuration changes, you can use the default values by creating a RowLayout using its default constructor.

The configurable properties of RowLayout are listed in the following table. The variables are member fields in RowLayout. You can access them directly.

Table 2. `RowLayout` properties

Variable	*Default*	*Description*
`justify`	false	Spreads the widgets across the available space within a composite.
`marginBottom` `marginLeft` `marginRight` `marginTop` `spacing`	3	Number of pixels around widgets. Spacing represents number of pixels between widgets.
`pack`	true	Forces all components to be the same size within a composite.
`type`	SWT.HORIZONTAL	Positions the widgets in rows or columns.
`wrap`	true	Wraps the widgets in row/column if there is not enough space on the row/column

Figure 6, shown below, shows the fields that do not depend on the size of the composite. *Figures 3*, *4,* and *5* show the properties that depend on the size of the composite.

Figure 3 shows the effect of the wrap property. The wrap is set to false and shell window is resized. As it is shown, there is no wrapping after resize when the wrap is set to false. In contrast to *Figure 2* above, the controls on the composite do not wrap.

Figure 3. *RowLayout.wrap=false*

The pack property forces all components to be the same size within the composite. This property sets the width of the controls to the widest and height of the components to the highest. *Figure 4* shows both cases.

Figure 4. *RowLayout.pack=true, false*

The justify property spreads the widgets across the available space within a composite. *Figure 5* shows the effect of the justify property.

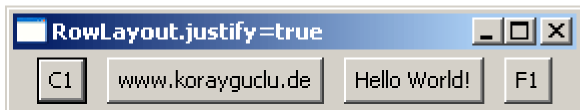


Figure 5. *RowLayout.justify=false, true*

Figure 6 shows some properties listed in *Table 2*.

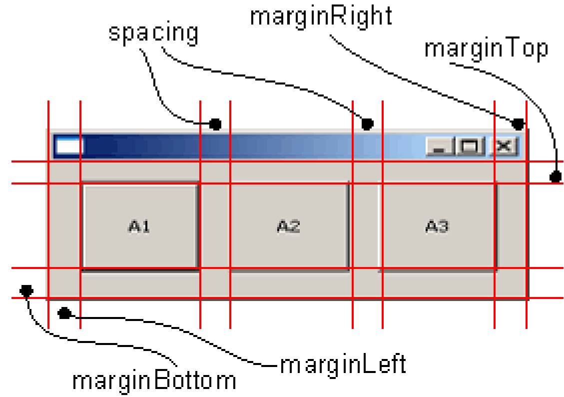


Figure 6. *RowLayout properties*

RowLayout is similar to FillLayout, but has the following key advantages:

- If the number of widgets do not fit in a row, it wraps the widgets.
- It provides configurable margins.
- It provides configurable spaces.
- It provides RowData object.

3.1. RowData

Each widget can have different RowData objects in RowLayout. You can specify different a height and width for each widget in the composite having RowLayout.

Table 3. `RowData` properties		
Variable	*Default*	*Description*
`height` `width`	0	Height and width of the widget

RowData only has two properties. There are only the width and height properties that we can use to set the width and height of each widget. If you want to change the size of a widget, you should create a RowLayout for the composite and you should create a RowData object for each widget. You can specify different RowData objects for each widget.

Listing 5. RowData Example

```
  // ...
  RowLayout rowLayout = new RowLayout();
  shell.setLayout(rowLayout);
  for (int i = 1; i < 6; i++)
  {
    Button button = new Button(shell,
SWT.PUSH);
    button.setText("A" + i);
    RowData rowData = new
RowData(i*15,i*25);
    button.setLayoutData(rowData);
  }
  // ...
```

Listing 5 shows a simple RowData example. This code will generate buttons having different heights and widths.

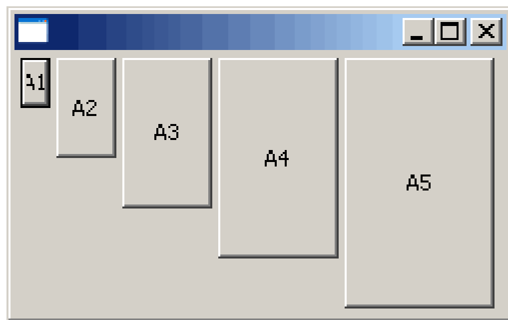


Figure 7. *RowData Example*

GridLayout

GridLayout is the most useful and flexible layout.The GridLayout lays out the widgets in grids. Configureable properties of the GridLayout are listed in *Table 4*.

The properties listed in *Table* 4 affect all the layout behavior. The GridLayout provides a data object as well. If you need to set a different property for a cell, you need to create a GridData object and set this as the control's layout data object.

The makeColumndEqualWidth property in *Table 4* makes all the controls the same size, as shown in *Figure 8*. This property makes the columns the same size. If you want the widgets also to be the same size, you should need to use GridData object's horizontal/vertical alignment properties, as shown in *Figure 12*.

Figure 8. *GridLayout.makeColumnnsEqualWidth=false,true*

The numColumns field must be set. This field specifies the number of columns in the GridLayout. If the number of components added is bigger than the number of the components in a composite, GridLayout adds the component to a new row.

For example, five components added to a GridLayout in *Figure 9*. The GridLayout in *Figure 9* only has four columns. For that reason, button A5 will be added to a the second row.

Figure 9 shows the location of the properties on the UI that are listed in *Table 4*.

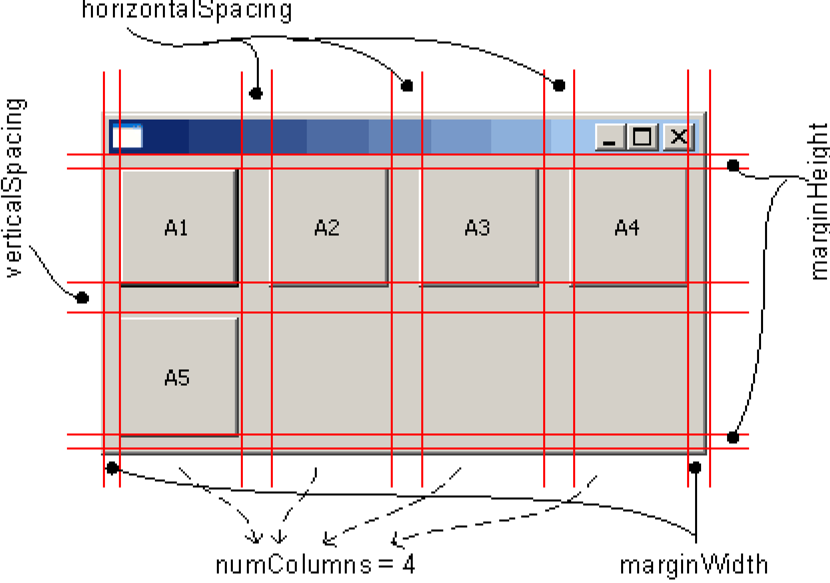


Figure 9. *GridLayout Example*

Initially, each column can have a different width. If you want to force all columns to be the same size, you must set the `makeColumnsEqualWidth` variable to true, as shown in *Figure 8*.

Listing 6. GridLayout example

```
// ...
GridLayout gridLayout = new GridLayout();
shell.setLayout(gridLayout);
gridLayout.numColumns = 5;
for (int i = 1; i <= 7; i++)
{
  Button button = new Button(shell, SWT.PUSH);
  button.setText("A" + i);
}
// ...
```

The preceding code snippet creates a new grid layout and sets the number of columns to 5. Afterwards, seven components are added to the composite. The figure below shows the position of the widgets.

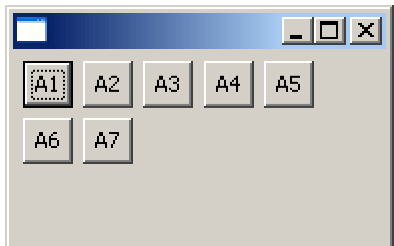


Figure 10. *GridLayout Example using default properties*

Advantages:

- It is the most powerful layout
- It puts widgets in rows and columns
- It provides configurable margins
- It provides configurable spaces
- It provides a GridData object

GridData

Each widget within a composite having GridLayout can have GridData set a different property for each widget. Configureable properties of the GridData object are listed in *Table 5*.

Table 5. `GridData` properties

Variable	*Default*	*Description*
`grabExcessHorizontalSpace` `grabExcessVerticalSpace`	false	If true, after resize, the widget will grow enough to fit the remaining space.
`heightHint` `widthHint`	SWT.DEFAULT (indicates that no minimum width is specified.)	Specifies a minimum width/height for the column.

horizontalAlignment verticalAlignment	GridData.BEGINNING(possible values are BEGINNING, CENTER, END, FILL)	Specifies how controls will be positioned horizontally/vertically within a cell.
horizontalIndent	0	Specifies the number of pixels of indentation that will be placed along the left side of the cell.
horizontalSpan verticalSpan	1	Specifies the number of column/row cells that the control will take up.

The effect of the first property listed in *Table 5* is illustrated below. The grabExcessHorizontalSpace (or grabExcessVerticalSpace) property can be hard to understand at first. For that reason, the area that is affected by this property is highlighted in the picture below. As you can see, if this property is set to true, the width of the grids will be as large as possible. If you want a widget to fill the horizontal space, you need to use this in combination with the alignment property shown in *Figure 12*.

Figure 11. *GridData.grabExcessHorizontalSpace=false,true*

The horizontalAlignment (or verticalAlignment) property sets the alignment of the control. You can use this in combination with the grabExcessHorizontalSpace property.

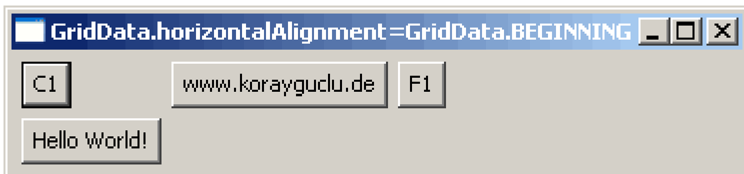


Figure 12. *GridData.horizontalAlignment = GridData.BEGINNING, GridData.CENTER, GridData.END, GridData.FILL*

The effect of the horizontalSpace is shown below.

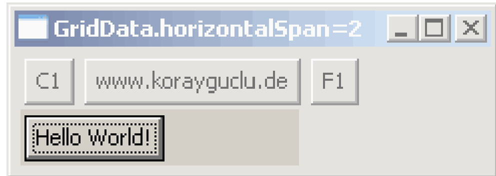


Figure 13. *GridData.horizontalSpan=1,2*

FormLayout

FormLayout is a very flexible layout, like GridLayout, but it works in a completely different way. In GridLayout, you have to plan everything ahead and build your user interface. In contrast to GridLayout, FormLayout is independent from the complete layout. The position and size of the components depend on one Control.

The location of the properties listed in *Table 6* can be seen in *Figure 9*.

FormData and FormAttachment

Each widget within a composite having FormLayout can have FormData to set the different layout properties for a widget.

The FormData object has the properties listed in *Table 7*.

A FormData object can have 0 or 4 FormAttachment objects. The form attachment can have controls (see *Figure 15*). The FormAttachment object can have the properties listed in *Table 8*.

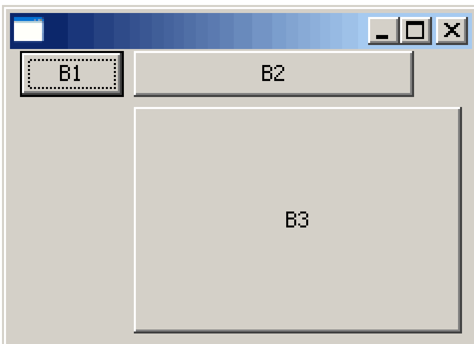


Figure 14. *FormLayout Example*

The above example shows a form layout created by using the code snippet below.

Source 7. `FormLayout` example

```
FormLayout layout= new FormLayout();
shell.setLayout (layout);
Button button1 = new Button(shell, SWT.PUSH);
Button button2 = new Button(shell, SWT.PUSH);
Button button3 = new Button(shell, SWT.PUSH);
button1.setText("B1");
button2.setText("B2");
button3.setText("B3");
FormData data1 = new FormData();
data1 .left     = new FormAttachment(0,5);
data1 .right    = new FormAttachment(25,0);
button1.setLayoutData(data1);
FormData data2 = new FormData();
data2.left      = new FormAttachment(button1,5);
data2.right     = new FormAttachment(90,-5);
button2.setLayoutData(data2);
FormData data3 = new FormData();
data3.top       = new FormAttachment(button2,5);
data3.bottom    = new FormAttachment(100,-5);
data3.right     = new FormAttachment(100,-5);
data3.left      = new FormAttachment(25,5);
button3.setLayoutData(data3);
```

Figure 15 shows the location of the properties and location of the buttons.

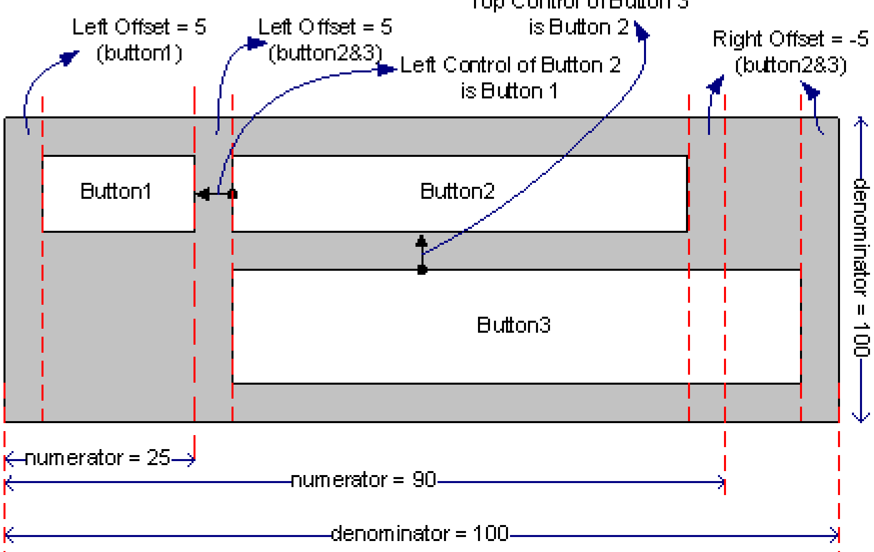


Figure 15. *FormLayout Example*

StackLayout

StackLayout is different from the other layout classes. StackLayout displays only one Control at a time; however, other layout classes attempt to display many Controls at a time. StackLayout is used in property pages, wizards, and so forth.

The StackLayout has the properties listed in *Table 9*.

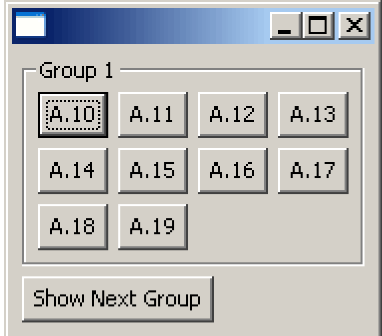


Figure 16. *StackLayout Example*

Figure 16 shows a simple stack layout example. We click on the Show Next Group object each time to change the top control.

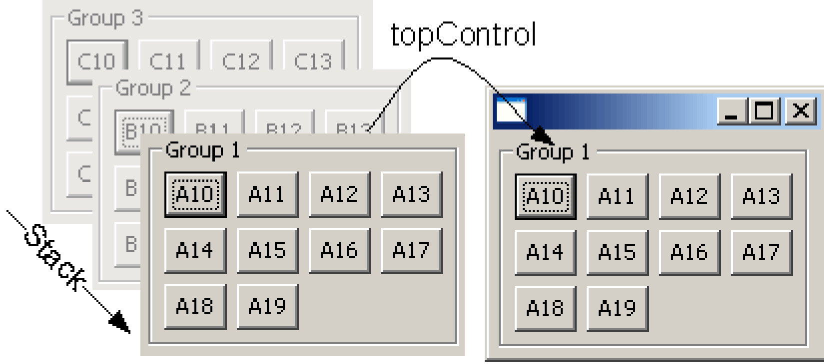


Figure 17. *StackLayout Example*

Figure 17 shows an example of StackLayout. A stack of controls exists. It is allowed to display only one control at a time. The highlighted picture on the right shows the topControl (active) control.

Source 8. `StackLayout` example

```
  // ...
  shell.setLayout(new GridLayout());

  final Composite parent = new Composite(shell,
SWT.NONE);
  parent.setLayoutData(new GridData());

  final StackLayout layout = new StackLayout();
  parent.setLayout(layout);

1 final Group[] group = new Group[3];
  for (int k = 0; k < group.length; k++)
  {
    group[k] = new Group(parent, SWT.NONE);
    group[k].setText("Group " + (k + 1));

    GridLayout gridLayout = new GridLayout();
    gridLayout.numColumns = 4;
    group[k].setLayout(gridLayout);

    Character character = new Character((char) ('A' +
k));
    for (int i = 10; i < 20; i++)
    {
      Button bArray = new Button(group[k], SWT.PUSH);
      bArray.setText(character + "." + i);
    }
  }
2 layout.topControl = group[0];

  Button b = new Button(shell, SWT.PUSH);
  b.setText("Show Next Group");
  final int[] index = new int[1];
  b.addListener(SWT.Selection, new Listener()
  {
    public void handleEvent(Event e)
    {
      index[0]          = (index[0] + 1) % 3;
3     layout.topControl = group[index[0]];
4     parent.layout();
    }
  });
  // ...
```

We have three groups and we add button controls to those groups. Afterwards, we need to set the 2 initial topControl. We create an action listener when the button is clicked. The action listener 3 changes the top control and calls the layout() method of the 4 parent control. This step is very important: If you do not call the layout method of the 4 parent control, you won't be able to see any change.

Resources

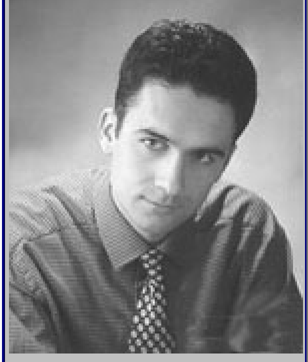
Links

- *SWT Programming with Eclipse*, by Koray Güclü, http://www.developer.com/java/other/article.php/3330861
- *A First Look at Eclipse Plug-In Programming* by Koray Güclü, http://www.developer.com/java/other/article.php/3316241.
- *Understanding Layouts in SWT*, by Carolyn MacLeod.
- *Eclipse User Interface Guidelines*, by Nick Edgar, Kevin Haaland, Jin Li, and Kimberley Peter.
- Java Tutorial - Creating a GUI with JFC/Swing.
- Eclipse uses the Common Public License.
- Eclipse download site: http://www.eclipse.org/downloads/
- Eclipse Java documentation.
- Java download Web site: http://java.sun.com/downloads/

Books

- David Gallardo, Ed Burnett, Robert McGovern (2003) *Eclipse in Action: A Guide for Java Developers*, Manning Publications.
- Sherry Shavor, Jim D'Anjou, Scott Fairbrother (2003) *The Java Developer's Guide to Eclipse*, Addison-Wesley Professional

About the Author



Koray Güclü

He is working as a freelance author and software architect. He is currently finishing his book on Software Architectures and Design Patterns. His main interest areas are Software Architectures, Data Warehouses, and Database Modeling.

www.korayguclu.de

Swing and SWT: A Tale of Two Java GUI Libraries

By Mauro Marinilli

In this article, we will talk about Java J2SE graphical user interface (GUI) libraries. We will adopt a programmer's viewpoint in our discussion. We assume the reader is familiar with the basics of the Swing library and wants to explore the SWT technology.

We will discuss two Java GUI libraries:

- Swing—The reference GUI toolkit for J2SE.
- SWT—This library has been developed by IBM as a part of the Eclipse platform.

Eclipse is an open-source, IBM-sponsored, fully extensible IDE, built using Java and SWT. SWT originated as Eclipse's GUI library, but it can be used outside it as a an alternative GUI library to both Sun's AWT and Swing.

In the following section, we will introduce the basics of SWT. We will assume the reader is familiar with Swing. Finally, we compare SWT and Swing.

SWT

SWT (Standard Widget Toolkit) is a graphics library and a widget toolkit integrated with the native window system (especially with Windows but Linux and Solaris are supported as well). Despite the tight integration with the native target platform, SWT is an OS-independent API. SWT can be seen as a thin wrapper over the native code GUI of the host operating system.

At a higher level of abstraction, also a part of the Eclipse platform, lies JFace. This is a GUI library, implemented using SWT, that simplifies common GUI programming tasks. JFace is independent of the given window system, in both its API and implementation, and is designed to work with SWT, without hiding it. For brevity, we will discuss only SWT here.

Introduction

The design strategy of SWT was focused on building a simple, essential library that would produce GUI applications closely coupled to the native environment. SWT delegates to native widgets for common components (such as labels, lists, tables, and so on) as AWT does, while emulating in Java more sophisticated components (for example, toolbars are emulated when running on Motif) similarly to Swing's strategy.

SWT has been designed to be as inexpensive as possible. This means (among the other things) that it is native-oriented. Anyway, it differs from AWT in a number of details. SWT provides different Java implementations for each platform, and each of these implementations calls natively (through the Java Native Interface, JNI) the underlying native implementation. The old AWT is different in that all platform-dependent details are hidden in C (native) code and the Java implementation is the same for all the platforms.

Resources

An important difference from normal Java programming is the way OS-dependent objects are managed in SWT. Swing emulates a large part of such objects (such as widgets, for instance) in Java, leaving the disposal job to the JRE garbage collector. This saves a lot of complexity for the programmer but this lack of control can lead to some unexpected issues, especially with cross-platform development.

SWT designers chose a different approach, obliging the developer to explicitly dispose of OS-dependent objects in the application code. SWT has been designed with efficiency in mind, so handling explicitly OS resources becomes an occasion to promote efficient programming and not just a necessity. Resource disposal is needed to free the OS resources used by the SWT application. Such OS resources need to be explicitly de-allocated by invoking the `dispose` method.

In practice, disposing of objects is a delicate business and it can lead to unpredictable results whenever another object tries to access an already disposed item.

The Basic Structure of an SWT Program

As already said, an SWT program relies upon the native platform resources (wrapped by the `Display` class, as we will see later on) both in terms of allocated resources (such as colors, images, and so on) and as regards the event mechanism. As regards event handling, in SWT there is only one thread that is allowed to handle native user interface events.

We now see the basic structure of an SWT program. This will help us understand SWT's basic architecture.

There are two basic classes used in any SWT application, the `Display` and `Shell` classes. Instances of the `Display` class are responsible for managing the connections between SWT and the underlying operating system, enforcing the SWT models (for colors or images, for example). The `Shell` class instead represents windows managed by the platform-dependent desktop manager.

Typically, an SWT program will first create a local resources manager (an instance of the `Display` class) and attach all the needed widgets to one or more `Shell` objects. Listing 1 shows this typical mechanism.

Listing 1: A snippet of code showing the use of the `Display` and `Shell` classes.

```
00:    Display display = new Display();
01:    Shell shell = new Shell(display);
02:    //GUI code here
03:    shell.open();
04:    while (!shell.isDisposed()) {
05:      if (!display.readAndDispatch())
06:        display.sleep();
07:    }
08:    display.dispose();
09:  }
10: }
```

To fully understand the code in Listing 1, one should understand the `readAndDispatch` method of the `Display` class. Such a method reads an event from the operating system's event queue, dispatching it appropriately. It returns `true` if there is additional work to do, or `false` if the caller can sleep until another event is placed on the event queue.

The loop at lines 4-9 in Listing 1 is typical of SWT applications. It takes care of forwarding all underlying platform events to the SWT application until the shell is disposed and the program can exit the event loop.

Basic Controls

Some of the basic components (or controls, as they are called in SWT) are the following:

- `Button`. This component is the well-known button component used in toolbars, forms, and so forth.
- `ComboBox`. This widget is the well-known combo box component.
- `Label`. This component represents a (non-selectable) object that displays a string or an image.
- `List`. This widget represents a basic list component.
- `ProgressBar`. It shows a progress indicator.
- `Sash`. It is the Swing equivalent of a JSplitPane. It is a widget that can be dragged to resize two areas in a GUI.

- `Scale`. This component implements an editable GUI item representing a range of continuous numeric values.
- `Slider`. This component represents an editable object that stands for a range of discrete, numeric values.
- `Table`. This component represents a basic table.
- `Text`. This component represents a basic text area.
- `Tree`. This component represents a basic tree widget.
- `StyledText`. This component represents a text area with styled fonts and other advanced attributes.

SWT vs. Swing

We conclude this article comparing these two technologies. Which is better, and when?

Swing provides a larger set of features, it is much more elegant, and gives an higher abstraction level (that turns helpful when developing complex GUIs with custom components). In general, SWT is easier to use at first, but when it comes to building complex GUIs, Swing usually results are easier to use and more flexible. In Table 1, support for the main components is shown for the three main GUI libraries.

Table 1: Comparing visual components

Component	SWT	Swing	AWT
Button	X	X	X
Advanced Button	X	X	
Label	X	X	X
List	X	X	X
Progress Bar	X	X	
Sash	X	X	
Scale	X	X	
Slider	X	X	
Text Area	X	X	X
Advanced Text Area	X	X	
Tree	X	X	
Menu	X	X	
Tab Folder	X	X	
Toolbar	X	X	X
Spinner	X	X	
Spinner	X	X	
Table	X	X	X
Advanced Table	X	X	

At first, SWT may seem simpler to use than Swing because it spares developers from a lot of sophisticated issues (such as the elaborated Swing class hierarchy, pluggable look and feel, the Model-View-Controller approach, and so on). Anyway, one potential problem with SWT is in the need for resource management. Swing (and AWT) follows the Java paradigm of automatic resources disposal, while in SWT de-allocating native resources needs to be accomplished explicitly by the programmer. Explicitly de-allocating the resources could be a step back in development time (and costs) at least for the average Java developer. This is a mixed blessing. It means more control (and more complexity) for the SWT developer instead of more automation (and slowness) when using Swing.

Although SWT supports non-natively rendered widgets (similarly to Swing), the default widget implementations are based on native peers (similarly to the old AWT). In addition, most of the Eclipse peers do not allow for inheritance. (Obviously, if you're writing Eclipse plugins, you have no choice but to use SWT.)

Recapping, whenever one needs a tight integration with the native platform, SWT can be a plus. Similarly, if one wants to take advantage of the Java language but needs to deploy only on Windows (or some of the few other supported platforms), SWT (and the Eclipse platform) is a better choice than Swing. But, be careful about the hidden costs associated with learning to use the Eclipse IDE and the SWT library. Also, the explicit de-allocation mechanism adopted in SWT could be useful in some situations and uselessly complex in others. If you don't need cross-platform development and your target OS is supported (Windows, for example), you may consider SWT, especially if your customers have old, limited machines where Swing use can be prohibitive. For all other scenarios, Swing provides the conservative choice.

Conclusions and References

In this article, we introduced SWT and compared it with Swing. My intent was to give a first idea of this library and its main characteristics. We concluded contrasting briefly SWT against Swing.

In the following are listed some (among the many) resources available for the interested reader.

- http://www.eclipse.org/—The home page of the Eclipse project.
- http://gcc.gnu.org/java/—The home page of the GCJ project.
- http://www.java.sun.com/products/plugin/—The official Sun Web site about Swing technology.
- http://www-106.ibm.com/developerworks/library/j-nativegui/—The site for Vogen, Kirk's article "Create native, cross-platform GUI applications."

About the Author

Mauro Marinilli is currently finishing his second book, on professional Java graphical user interfaces. He teaches CS courses at the University of Rome 3 in Italy while being active as a consultant and as an academic researcher in case-based reasoning, Human Computer Interaction and its applications to e-learning. You can email him at contact@marinilli.com.