Migrate your Swing application to SWT

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

What is this tutorial about?

Since its first release and its donation to the open source community in 2001, the Eclipse platform has been continually gaining importance in the tool-provider community. The Eclipse consortium already regroups more than 40 industry-leading companies that deliver or plan to deliver tools that can be plugged in the Eclipse platform, or products based on the Eclipse platform.

The advantages of the Eclipse platform for tool developers are obvious:

- For the first time, there is a high-performing, vendor-independent platform that has been widely accepted by the industry.
- The platform's highly modular nature and great extensibility allow a seamless integration of a variety of tools coming from different vendors. Users can get the best tools from different providers, and use them together without having to worry about interoperability.
- By providing tools as Eclipse plugins, tools providers can cover with one release all the Eclipse-based products on the market. They don't have to build a workbench around their tool and can concentrate their effort on the development of their core features.
- The consistent UI among the different Eclipse tools reduces the learning curve for users.

One of the reasons for the success of the Eclipse platform is the performance of its user interface compared to other Java applications. This level of performance was reached thanks to the Standard Widget Toolkit (SWT), a widget library that was developed as an alternative to Swing. SWT allows you to build cross-platform user interfaces that are as rich as Swing UIs and that perform as well as native UIs.

Although programmers who try SWT tend to be very enthusiastic about it, this toolkit does have a drawback: SWT is not compatible with AWT (the Abstract Window Toolkit) and Swing. Mixing SWT and AWT panels in the same application can, in the worst case, cause the JVM to crash on some platforms. Thus, if you want to deliver an existing Swing tool as an Eclipse plugin, you need to rewrite its UI with SWT. This task can be very tedious for complex UIs.

Considering the number of tools on the market that currently use a Swing UI, a bridge technology or method that would allow developers to port an existing application from Swing to SWT with a minimum of change would be in great demand. This is always a very hot topic in the discussion forums about Eclipse and SWT.

The purpose of this tutorial is to introduce a methodology for such a migration. The techniques presented here won't allow you to automatically port an existing application without any code modification, but they will show you how to do a manual migration of the Swing code with very few changes to the original code.

We will begin with a study of the main differences between AWT/Swing and SWT. We'll then examine migration techniques that can be used to successfully port Swing code to SWT with a minimum of change, and we'll compare each Swing component with its SWT equivalent in detail, and discuss the problems you might encounter in porting. Finally, we'll work through a concrete example where a Swing dialog is ported to SWT using the techniques we've presented.

This tutorial includes sample code applying the described methods on the standard components of the Swing library. You are free to use and modify this code in your own projects.

Should I take this tutorial?

Before you take this tutorial, you should have a good knowledge of the Swing API, as well as a basic knowledge of SWT. This tutorial was written for people who want to migrate a Swing application to SWT, or for Swing programmers who want to know which features available in Swing are also available in SWT, and what limitations they should expect. For this reason, this tutorial uses a lot of terms and comparisons that are relevant specifically to Swing development. It mainly focuses on how features available in Swing can be programmed in SWT, not on features that are available in SWT but not in Swing.

This tutorial is neither an introduction to UI programming, nor an introduction to SWT. If you need an introduction to SWT, you will find relevant links in Resources on page 95 that you should read before taking this tutorial.

To complete this tutorial, you will need to install *Eclipse* 2.1 or an equivalent product (IBM *WebSphere Studio Application Developer* 5.1 for instance), which includes the SWT packages. You may also wish to review the *SWT development resources*.

Note that this tutorial is very comprehensive and will require significant time to complete. However, it serves as excellent reference material. I recommend you download the PDF after you complete the tutorial for offline viewing.

About the author

Yannick Saillet is a software engineer at the IBM Laboratory of Boeblingen in Germany. Yannick joined IBM Germany as software developer in 1998. He first worked for IBM Learning Services as a software engineer in several distributed learning projects. He joined the IBM Boeblingen Laboratory in 2000 and since that date has been active in the development of the DB2 Intelligent Miner products. He received his master degree from the ESSTIN (Ecole Superieure des Sciences et Technologies de l'Ingenieur de Nancy) at the University of Nancy in France.

For technical questions or comments about the content of this tutorial, contact Yannick Saillet at *ysaillet@de.ibm.com*, or click Feedback at the top

of any panel.

Section 2. The history of Swing and SWT

AWT and Swing

If you are reading this tutorial, you are probably quite familiar with AWT and Swing. In this section, we will refresh your memory on the history and the basic architecture of these libraries, so that you can better understand what makes SWT different.

AWT (the Abstract Windowing Toolkit) was the Java language's first widget library; it accompanied version 1.0 of the language in early 1995. The original idea was to define a set of widgets that were common to all platforms, and to map these widgets to the native components of the underlying windowing system on each particular platform.

For each widget available in AWT, there is:

- A public Java class, defining the public API of the component. These classes, defined in the package `java.awt`, are implemented once for all platforms.
- A native peer class relaying the API calls from the public class to the native widget. The native classes form the JNI layer for the native API of the windowing system and are reimplemented for each platform.

This approach originally seemed like a good idea. By introducing an abstraction layer between the native API of the platform and the application itself, it allowed developers to write user interfaces that could run on any platform, fulfilling Java technology's "Write Once, Run Anywhere" motto. Furthermore, porting AWT to a new platform would only involve porting the thin JNI layer for the new windowing system.

However, this architecture also had some major drawbacks. AWT did not perform well and had a lot of bugs. More seriously, AWT's functionality was too limited. Because the approach was to take only the least common denominator of all the windowing toolkits on the market, if a certain feature was not available on a single platform, it was excluded from AWT. For this reason, AWT doesn't provide such common components as trees, tables, tabbed panes, and rich text, although these components are nowadays quite standard and used in nearly every modern UI.

Swing came later and tried to solve this problem by providing a 100 percent pure Java emulated library of widgets. The term *emulated* here means that Swing makes no use of any native API to draw or create its widgets, but reimplements its own look and feel created from scratch using the Java language only.

The advantage here is that the created widgets are very flexible -- nearly everything can be customized -- and that the look and feel is exactly the same on all platforms. But Swing unfortunately has several drawbacks as well:

- The API is complicated -- that's the price to pay for flexibility.
- The performance is not good; because everything is emulated and drawn using basic `Graphics2D` calls, software and hardware optimizations from the native system are not possible.

- The look and feel of a Swing application are not exactly the same as those of a native application. The developers of Swing keep trying to reproduce the look and feel of systems like Windows, but they can't stay synchronized with new OS versions. Furthermore, the customization of the colors and font schemes of the underlying windowing system are difficult to propagate in the emulated widgets.

SWT and JFace

SWT (the Standard Widget Toolkit) is an alternative toolkit that was created by IBM and has become popular due to its use in the Eclipse platform. SWT has now been donated to the open source community along with the rest of the Eclipse platform.

SWT was created to solve the problems existing in AWT (lack of functionality) and Swing (inconsistency with the native look and feel, and poor performance). This has been achieved by using a solution that lies between the two extreme approaches represented by AWT (using the smallest set of common features) and Swing (emulating everything). Like Swing, SWT provides a rich collection of widgets with all the functionality required by a modern UI -- but like AWT, SWT also makes use of the native widgets and libraries of the underlying platform.

The collection of widgets provided by SWT includes all the components a UI programmer might need to build a modern user interface: trees, tables, progress indicators, sliders, tabbed panes, and so on. Although the internal implementation makes use of a proprietary API on each platform, the public API, against which a UI developer will program, is completely OS-independent and quite simple to use -- like AWT.

The reason why SWT can offer much more functionality than AWT is that it uses native widgets where possible, but emulates widgets that may not be available on a specific platform, just like Swing does for all widgets. For example, Motif doesn't provide any tree component, but Windows and GTK do. The implementations of the tree widget under Windows and GTK simply make use of the native widgets. The Motif implementation emulates a tree by combining several simpler widgets. The programmer using SWT doesn't notice the difference, because the public API is the same for all platforms. The emulated widget under Motif may not perform as well as a native widget would, but this performance issue would only concern this particular widget on this particular platform.

SWT is a standalone library. It doesn't make use of any AWT classes, and has no dependency on Eclipse itself. Thus, you can see SWT as an alternative to AWT or Swing. The advantages are obvious:

- Because, for the most part, it uses native components, SWT preforms much better than Swing.

- With SWT, you get a better integration with the underlying windowing system. The look and feel are that of the underlying system, and the color and font schemes of the system are used. A Java application using SWT cannot be distinguished from a native application.

- SWT has already been ported to most of the platforms on the market, so platform port is not an issue.

For more information on the design of SWT, read "SWT: The Standard Widget Toolkit, Part 1" by Steve Northover. A link to this article is available in Resources on page 95 .

JFace is a pure Java API that groups SWT widgets into a set of more complex components or frameworks with a higher level of functionality. SWT only provides the basic components comprising a user interface, such as buttons, lists, text fields, and so on. JFace provides the more complex dialogs and UI components that are quite often reused when building a UI. Examples of such components include wizard dialogs, preference dialogs and progress indicators.

Section 3. Differences between Swing and SWT

Graphical resources and garbage collection

Switching from AWT/Swing to SWT doesn't just mean learning a new API; it also requires former Swing programmers to change some of their habits and to care about new coding rules they didn't have to deal with in the Swing world.

SWT uses a completely different philosophy than AWT and Swing do when it comes to handling graphical resources. In AWT and Swing, you can, in most cases, rely on the JVM's garbage collector to free up graphical resources (image handles, colors, cursors, fonts, widgets, etc.) when these are not needed anymore. I emphasize the words "in most cases," because even in AWT this isn't always the case. For example, a `java.awt.Image` must be freed up explicitly by invoking the method `flush()` if you want the pixels to be freed. Programmers of applications making heavy use of images often fall into the trap of thinking that if the garbage collector finalizes the reference to an image, it will free up the platform resources assigned to it as well. Then they wonder where the memory leaks in their applications come from. There are some other examples of resources that have to be explicitly freed up in AWT -- `java.awt.Dialog` and `java.awt.Graphics` both have a `dispose()` method, for instance -- while other resources, such as fonts or colors, are automatically released by the garbage collector. This is quite confusing for programmers.

SWT uses a different approach: All SWT objects allocating platform resources (`Color`, `Cursor`, `Display`, `Font`, `GC`, `Image`, `Printer`, `Region`, `Widget`, and their subclasses) have to be explicitly discarded. The JVM's garbage collector will finalize unreferenced SWT objects, but it will not dispose of the platform resources used by them. Thus, if you delete all the references to one of these objects without having previously discarded it, you will have a memory leak. This sounds like a very constricting rule, but it is a clear rule and it is the price you pay for better UI performance.

To avoid resource leaks in an SWT application, you must follow this simple rule: *If you instantiate an object that consumes graphical resources, you have to dispose of it yourself.* Objects obtained from getters, diverse methods, or parameters should not be discarded by the code obtaining them, because the objects were not created there, and may be used by other parts of the application. The only exceptions are widgets: Disposing of a parent container will automatically dispose of all its children.

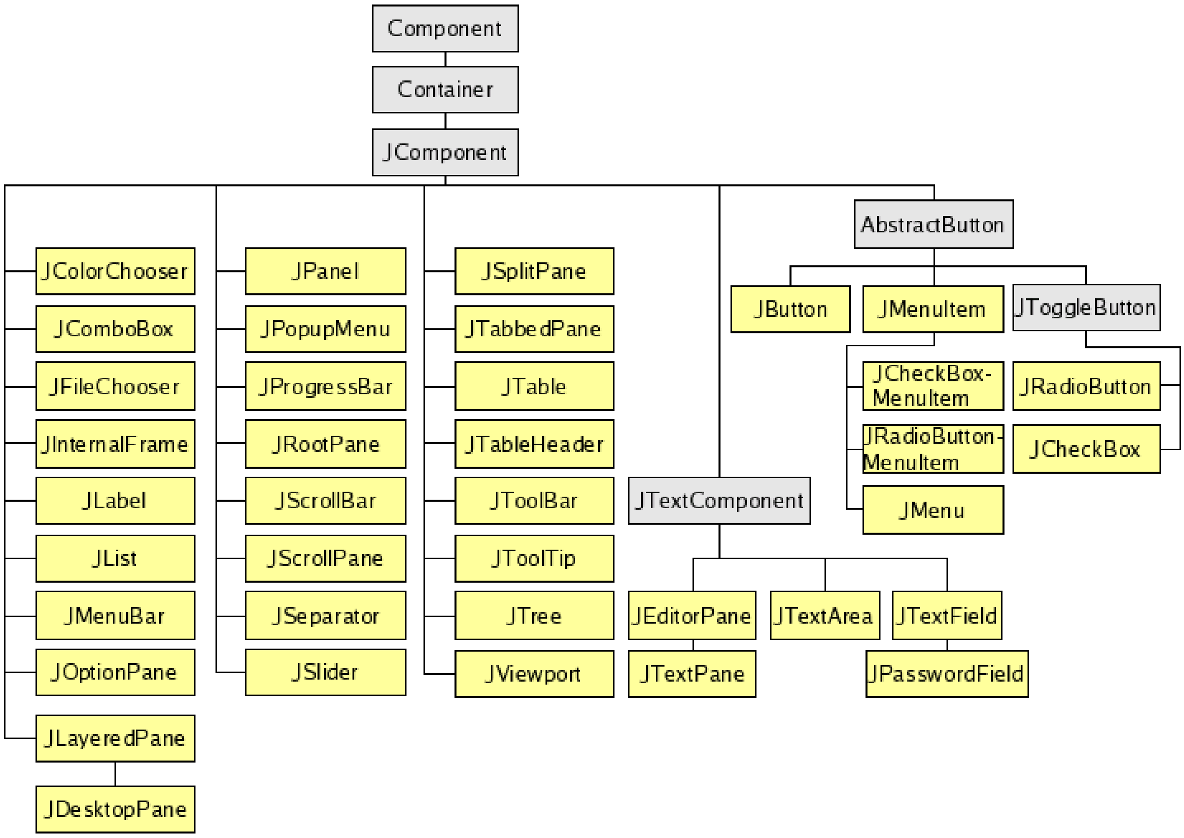
If you follow this rule, you won't have any problem with memory leaks of graphical resources.

Note that JFace provides helper classes and frameworks to help you to manage and discard resources (fonts and images) that may be shared by several components. These classes are contained in the package `org.eclipse.jface.resource`.

If you want to get a better understanding of the rules listed above, and the reasons why SWT doesn't behave like AWT when managing graphical resources, read "SWT: The Standard Widget Toolkit, Part 2" by Steve Northover and Carolyn MacLeod. A link to this article is available in Resources on page 95 .

The Swing component hierarchy

The most obvious difference between Swing and SWT is the component hierarchy. To facilitate the comparison between the Swing's and SWT component hierarchies, I've illustrated Swing's component tree in the following figure:

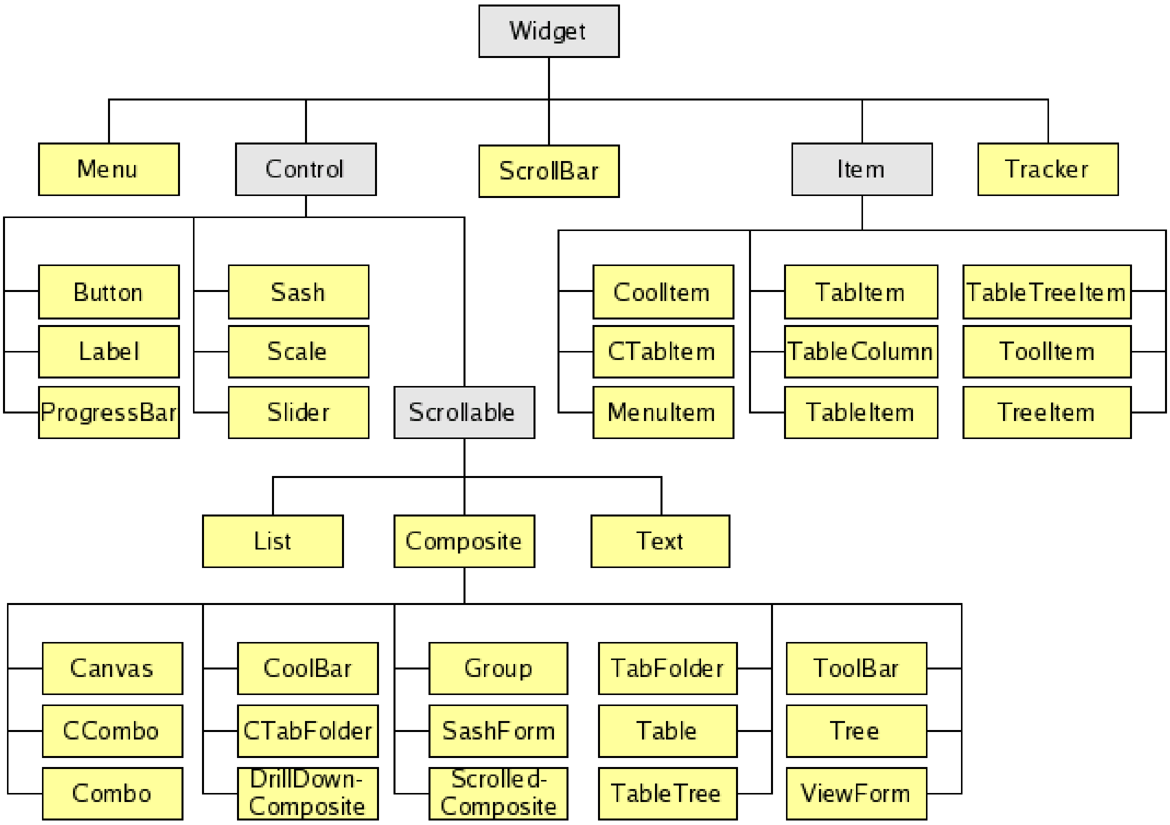


The boxes with a yellow background represent ready-to-use widgets that can be deployed in a user interface. The boxes with a blue background represent abstract classes that are not intended to be used directly.

As you can see, nearly all Swing components directly inherit from `JComponent`, which is itself a subclass of an AWT `Container`.

The SWT component hierarchy

Now let's take a look at SWT's component hierarchy:



As you can see, the number of available widgets here is pretty similar to what Swing offers, but the names and the hierarchy of the components is quite different.

- The superclass for all SWT components is `Widget`, which directly inherits from `Object`.
- The two most important subclasses of `Widget` are `Control` and `Item`. `Control` is the superclass for all widgets that can be added in a parent container and whose position and size can be set. `Item` is the superclass for components or sub-components that can only exist within another specific component, such as menu items, toolbar items, table rows or column, etc.
- Seven widgets directly inherit from `Control`. Six of these subclasses are simple components that don't allow children, such as buttons or labels. `Scrollable` is an abstract class, and is the superclass of all components that may be scrollable (tables, lists, text fields, and so on).
- `Composite` is an important class in the component hierarchy. It is the equivalent of AWT's `Container` and is the superclass of all components that allow children to be placed in them.

The correspondence between each Swing component and its equivalent in SWT will be introduced in Widgets on page 31 .

Containers and layouts

The equivalent of an AWT `Container` is an SWT `Composite`. As with a `Container`, you can add controls to a `Composite`, and set a layout manager that will relocate and resize the children as the parent composite is being resized.

However, there are some differences in this domain between AWT and SWT. If you look at the API documentation of `Composite`, keeping in mind that it is the equivalent of an AWT `Container`, you may be surprised to see that there is no direct equivalent for the methods `add(...)` or `remove(...)`, which in AWT allow you to add a child to or remove a child from its parent.

SWT controls are automatically added to their parent at construction time. When you construct an SWT control, the first parameter required by the constructor is always the reference to the parent composite. For this reason, `Composite` doesn't provide any `add(...)` method, as AWT's `Container` does. Although `Control` has a `setParent(Composite)` method that allows you to reparent a control -- that is, to remove it from its original parent and add it to a new parent -- this feature is not available for all widgets and all platforms, so you can't rely on it if your application has to be cross-platform. For example, Motif doesn't allow a control to be reparented. To test if this feature is supported by a particular platform or widget, you can use the method `Control.isReparentable()`. Invoking `setParent(Composite)` on a widget that is not reparentable will throw an exception.

Because the addition to the parent composite is done during the instantiation of a control, the order in which controls are instantiated defines the index the controls have in their parent. The index of a control in its parent may have an influence on the way the layout manager places it in the container. This can be an issue when porting existing Swing code, because in AWT/Swing, the order of instantiation of the children is not important -- in fact, a child can be instantiated before its parent. Only the order of addition of the children plays a role. When porting Swing code, you may have to change the order of creation of some widgets to get the same result as in Swing.

`Composite` doesn't provide a `remove(...)` method to remove a child as AWT's `Container` does. To remove a control from its parent, you have to dispose of it. However, you should be aware that a control that has been discarded can't be used anymore. There is no way to add a control to its parent again after it has been eliminated. You have to instantiate a new control again. Here, you don't have the flexibility of AWT, which allows you to remove a component, keep it instantiated offscreen, and later add it again to the same or a different parent.

Like AWT, SWT makes use of layout managers to place children of a container. The layout algorithms that are available are different, however. To get an overview of the SWT layout algorithms, read "Understanding Layouts in SWT" by Carolyn MacLeod and Shantha Ramachandran. A link to this article is available in Resources on page 95 .

As in AWT, some SWT layouts require you to set some layout constraints on widgets so that you can influence how the children of a container are going to be laid out. In AWT, you set this constraint by passing it as the second parameter to the method `Container.add(Component, Object)`. Because `Composite` doesn't provide any method to add a child, you have to set it by invoking a method named `setLayoutData(Object)` on the child component itself.

Data models and cell renderers vs. content providers and label providers

One of the most beautiful aspects of Swing's architecture is its strict adherence to the Model-View-Controller pattern. The clean separation between model, view, and controller can be above all observed in components like `JTable` or `JTree`, which use a data model:

- A data model provides, in an unformatted form, the raw content to be displayed by the component.
- The component uses a cell renderer to display the content of each cell in the component. Swing allows the cell renderer to be any kind of Swing component.
- The controller role -- modification of the model and of the presentation after a user interaction -- is assumed by the component itself.

SWT components don't have such a clean separation between model, view, and controller. If you create a table or tree using the SWT API only (that is, without using JFace), you'll probably miss the data models and cell renderers used in Swing. Creating a table using only the pure SWT API obliges you to create each row and each column like a standard control in a container, and to initialize them with rendered text and images. There is no support in SWT for data models.

Fortunately, on top of the standard SWT controls, JFace provides a framework that is comparable to the concepts used in Swing. To use this framework, you have to instantiate a JFace *viewer* on top of the basic SWT table or tree. There are different viewers specialized for each kind of control: `TableViewer` for a table, `TreeViewer` for a tree, etc. A viewer is a class that will extract data from a data model and automatically create and initialize the rows or items to display.

The equivalent of Swing's data model is in JFace called a *content provider* (see `org.eclipse.jface.viewers.IContentProvider`). Like a Swing `TreeModel` or `TableModel`, a content provider provides unformatted raw data that has to be displayed in the component. Unlike Swing's data models, JFace's content providers don't contain the data itself; instead, they extract that data from an input object that can be any kind of object. In this way, a JFace content provider acts as a data extractor: it knows how to extract data from a specific sort of input object, and provides a public interface used by the viewer to fill the underlying SWT component.

The equivalent of Swing's cell renderers in JFace is called a *label provider* (see `org.eclipse.jface.viewers.ILabelProvider`). Like Swing's renderers, the label provider defines how raw data provided by the content provider has to be displayed in the SWT component. JFace is here not as flexible as Swing is. In SWT/JFace, a cell of a tree or a table can only be represented by an icon and/or text. If you need custom rendering for some other kind of data, you have to display the renderer into an image and make the label provider return that image.

For more information and examples of how to use JFace viewers, read the related articles in Resources on page 95 .

Events

Like AWT and Swing, SWT lets your application react to user interactions by registering event listeners on components. There is not much difference in this area; the events thrown are all subclasses of `java.util.EventObject`, and the kind of events that are thrown, along with the listeners or adapters that are notified, are comparable to those in AWT and Swing.

Of course, the hierarchy of the events and their associated listeners is different. In Widgets on page 31 , we'll see what kind of events are thrown for SWT controls, and compare each to its Swing equivalent.

Section 4. Migrate your Swing code to SWT with minimal change

Migrate the layout managers

The *layout managers* -- the algorithms that define the location of the components of a container and how they are resized when the size of the container changes -- are the core of the UI design of a panel or dialog. Usually, when you design a panel, you first draw on a piece of paper the components that will compose the piece of GUI you are designing. Then you decide which layout managers are going to be used and, eventually, how the components will be grouped in invisible panels using other layout managers, so that the result looks like what you have originally designed. Thus, a GUI is typically made up of a combination of simple widgets and panels having their own layout and containing other widgets. When complex layout managers are used, each widget is additionally initialized with some layout information, which is interpreted by the specific layout manager in use.

Although the concept of a layout manager is quite common in most UI toolkits, each toolkit usually defines its own layout algorithms, which are not available in the other toolkits. AWT/Swing and SWT unfortunately confirm this rule: The most commonly used AWT layout managers, such as `BorderLayout`, `GridBagLayout`, and `FlowLayout`, have no direct equivalent in SWT. Of course, the layout managers provided by SWT are as powerful as those provided by AWT, but when you port an existing GUI, you'll need to design the layout of the UI again, so that you get the same layout with the new algorithms.

Thus, the layout managers used by the Swing application that you want to migrate are the first things that you should port to SWT. Most Swing applications always reuse a small number of layout managers. Porting them to SWT takes some extra work at the beginning of a project, but will save a lot of time during the migration of the GUI itself.

Porting an AWT layout manager to SWT doesn't present any technical problems because the methods to implement in order to create a new layout manager are quite similar in both toolkits. Creating an AWT layout manager -- a subclass of `java.awt.LayoutManager` -- requires you to implement the three following methods:

- `public Dimension minimumLayoutSize(Container parent)`: Computes the minimum size that a parent container should have when using this layout.

- `public Dimension preferredLayoutSize(Container parent)`: Computes the preferred size that a parent container should have when using this layout.

- `public void layoutContainer(Container parent)`: Sets the size and location of the children of the parent container.

Creating an SWT layout -- a subclass of `org.eclipse.swt.widgets.Layout` -- requires you to implement the two following methods:

- `protected Point computeSize(Composite parent, int wHint, int hHint, boolean flushCache)`: Computes the size that a parent composite should have when using this layout.

- `protected void layout(Composite parent, boolean flushCache)`: Sets the size and location of the children of the parent composite.

As you can see, the methods of an SWT layout are quite similar to the methods of an AWT layout manager. SWT has no equivalent for AWT's minimum size of a component. That means that all that you have to do is to port the algorithm of AWT's `preferredLayoutSize(Container)` into SWT's `computeSize(Composite, int, int, boolean)`, and to port the algorithm of AWT's `layoutContainer(Container)` into SWT's `layout(Composite, boolean)`.

If you own the source code of the AWT layout manager to port, you can easily do this with a couple of search-and-replace actions to adapt the layout algorithm to the SWT API.

You may think that I've made porting the standard AWT layout managers sound easier than it is. But here's some good news: I've already done the job for the standard AWT layout managers, so that you just have to concentrate on those layout managers that you wrote yourself.

You'll find the source code of the ported AWT layout managers in the following files of the j-swing2swtsrc.zip download in Resources on page 95 . Feel free to reuse this code in your projects, and eventually modify it to your needs.

For more information on the SWT layout, read "Understanding Layouts in SWT," by Carolyn MacLeod and Shantha Ramachandran. A link to this article is available in Resources on page 95 .

API mapping

After the layout managers you used in your Swing code have been ported to the SWT world, you will be confronted with the next problem: the differences existing between the Swing and SWT APIs. This is the most obvious problem when you port a GUI from one toolkit to the other: Nearly all the functionality you used in the Swing toolkit is also available in the SWT toolkit, but the class hierarchy, and the names of the methods and their syntax, are different in the new toolkit.

Your first strategy in tackling this problem may be to undertake a manual translation job: you analyse each line of code of your existing Swing GUI, search in the SWT documentation for an equivalent, and rewrite the code again with the new API

This strategy may be the fastest one if you only have a small amount of code to port, but if you plan to port a complete application with several dozen panels and dialogs, it can quickly turn into an astronomical amount of work.

There is a much better strategy to use. For each Swing component used by your application (see The Swing component hierarchy on page 8), you can write a wrapper class around the equivalent SWT component. Each wrapper class provides the same methods with the same syntax as the Swing component it emulates. Each of these methods invokes the equivalent method in the wrapped SWT component, ensuring the proper translation between the syntax used in Swing and that used in SWT.

The result of this work, which you should undertake before the migration of your code takes place, is a component hierarchy that is the mirror of the Swing component hierarchy, but has no dependency on any Swing or AWT class. The implementation of the Swing API exclusively invokes SWT methods.

This may seem like extra labor on your part, but it reduces a lot the work necessary to migrate your code: Because the SWT components can be controlled with an API that is a clone of the Swing API, you can migrate your code with simple search-and-replace operations.

The following code snippet shows you what such a wrapper class would look like. `SWTComponent` is the wrapper class on top of the wrapper class hierarchy. It corresponds to Swing's `JComponent`.

```
public class SWTComponent {
  (...)
  /**
   * SWT control to which this object is doing the API mapping.
   */
  protected Control control;
  (...)
  /**
   * Constructs a new API mapper on an existing SWT control.
   * @param control the SWT control whose API is mapped
   */
  public SWTComponent(Control control) {
    this.control = control;
    control.setData(KEY_SWT_COMPONENT, this);
  }

  /**
   * Returns the SWT control whose API is mapped by this object
   */
  public final Control getControl() {
    return control;
  }

  //--- emulation of the AWT/Swing methods ---

  public final int getHeight() {
    return control.getSize().y;
  }

  public final Point getLocationOnScreen() {
    return control.toDisplay(0, 0);
  }

  (...)

  public final SWTContainer getParent() {
    return (SWTContainer)getSWTComponent(control.getParent());
  }

  (...)

  public final boolean hasFocus() {
    return control.isFocusControl();
  }

  public final void requestFocus() {
    control.setFocus();
```

```
  }

  (...)

  public final void setPreferredSize(Dimension preferredSize) {
    getControl().setData(KEY_PREFERRED_SIZE, preferredSize);
  }

  public final Dimension getPreferredSize() {
    // check if a preferred size was set with setPreferredSize
    Dimension preferredSize =
      (Dimension)getControl().getData(KEY_PREFERRED_SIZE);
    // if not, compute it with computeSize
    if (preferredSize == null) {
      Point size = getControl().computeSize(SWT.DEFAULT, SWT.DEFAULT);
      preferredSize = new Dimension(size.x, size.y);
    }
    return preferredSize;
  }

  (...)

  //--- Helper methods ---

  /**
   * Returns the SWTComponent controlling a specific SWT control
   * @param control the SWT control
   * @return the SWTComponent assigned to it, or null if none.
   */
  public static SWTComponent getSWTComponent(Control control) {
    return (SWTComponent)control.getData(KEY_SWT_COMPONENT);
  }

}
```

This snippet is only a small part of the complete implementation. The complete source code for `SWTComponent` is in the j-swing2swtsrc.zip file in Resources on page 95 .

You'll notice that:

- The field `control` stores the reference to the wrapped SWT control.

- The method `getLocationOnScreen()` illustrates the emulation of the Swing API on top of SWT. This method emulates `java.awt.Component.getLocationOnScreen()`, the AWT method that returns the absolute coordinates of a component on the screen. SWT has a similar method, but with a different syntax: `Control.toDisplay(int, int)` converts coordinates in the coordinate system of the control to coordinates in the system of the screen. By passing `(0,0)` as parameters, you get the absolute coordinates of the component on the screen. With this method, you can use the AWT API on an SWT control, and you don't need to modify the existing code invoking Swing methods. Because the method is `final`, the compiler inlines the core of the method -- the code using the SWT API -- where it is invoked, so that you don't incur any performance penalty by using the wrapper class instead of rewriting your code with the SWT API.

- The method `setPreferredSize(Dimension)` stores the preferred size as user data in the SWT control. With `Widget.setData(String key, Object value)`, SWT lets you store any widget data in a kind of hashtable. This data can be retrieved at any time by invoking `getData(String key)` on the widget. Because SWT doesn't let you set the preferred size of the component, we use user data to store this information. The implementation of `getPreferredSize()` first checks to see if a preferred size was previously set with `setPreferredSize()`. If not, it invokes the method `computeSize(...)`, which is the equivalent of AWT's `getPreferredSize()`. The layout managers introduced in Migrate the layout managers on page 14 check for each component to lay out if a preferred size was stored as user data in the component.

Because `javax.swing.JComponent` is an abstract class without a constructor, the only constructor available in `SWTComponent` takes an already instantiated SWT control as its parameter. This constructor allows you to instantiate a wrapper class on any existing SWT control that may have been instantiated somewhere else in your application.

The following example shows you how a wrapper class can be instantiated around an existing SWT component. The object `button` is an SWT button created and initialized with the SWT API. By instantiating the class `SWTComponent` presented earlier in this section, with the SWT button passed as an argument in the constructor, you create a wrapper class that allows you to use the AWT/Swing API on the SWT component. When `getLocationOnScreen()` (from the Swing API) is invoked on the wrapper, the wrapper converts the call into its SWT API equivalent and invokes the corresponding SWT method on the wrapped SWT component. In this way, you can at any time use the Swing syntax of a method on an SWT component. The method `SWTComponent.getControl()` lets you retrieve the reference of the wrapped SWT component from the wrapper class. This can be useful if you need to invoke an SWT method and only have a reference to the wrapper class.

```
Button button = new Button(parent, SWT.PUSH);
(...)
SWTComponent wrapper = new SWTComponent(button);
// from here you can use the AWT/Swing API on the button...
Point pt = wrapper.getLocationOnScreen();
// ... or use the SWT API at your convenience
wrapper.getControl().addDisposeListener(listener);
```

The wrapper class for a non-abstract component would emulate the Swing constructors as well, so you can instantiate an SWT control and its wrapper class with a single invocation of the constructor of the wrapper class.

The following code snippet shows the wrapper class of Swing's `JLabel`:

```
public class SWTLabel extends SWTComponent {

  public SWTLabel(Label label) {
    super(label);
  }

  public SWTLabel(SWTContainer parent) {
    this(new Label(parent.getComposite(), SWT.NONE));
  }

  public SWTLabel(SWTContainer parent, String text) {
    this(parent);
    getLabel().setText(text);
  }

  public SWTLabel(SWTContainer parent, String text, int horizontalAlignment) {
    this(parent, text);
```

```
    setHorizontalAlignment(horizontalAlignment);
  }

  public Label getLabel() {
    return (Label)getControl();
  }

  public void setText(String text) {
    getLabel().setText(text);
  }

  public String getText() {
    return getLabel().getText();
  }
  (...)
}
```

This snippet is only a small part of the complete implementation. The complete source code of `SWTLabel` is in the j-swing2swtsrc.zip download in Resources on page 95 .

Because SWT requires that the parent of a control be passed as an argument when constructing a new control (remember, an SWT control is added automatically to its parent at creation time), the constructor of the wrapper class always requires one more parameter than its Swing equivalent: the reference to the parent container needs to be passed to the constructor.

The constructor of the wrapper class is the only part of the wrapper API that differs from its Swing equivalent. The migration of Swing code is easy, however. As an example, consider following Swing code:

```
JLabel label = new JLabel("Label Text", SwingConstants.CENTER);
```

It can be ported to SWT as follows:

```
SWTLabel label = new SWTLabel(parent, "Label Text", SwingConstants.CENTER);
```

As with the layout managers presented in the previous section, you will find wrapper classes for all the main Swing components included with the sample code provided with this tutorial (see Resources on page 95). Feel free to use these classes in your projects and eventually modify them to implement Swing methods I may not have implemented.

For more information on the individual Swing components' wrapper classes, read the panels describing the migration of the various components later in this tutorial -- see Widgets on page 31 to get an overview.

Trigger AWT/Swing event listeners from SWT

The code of a Swing application can usually be divided into three categories: *model, view,* and *controller,* as defined in the famous design pattern.

- Code belonging to the *view* defines what the GUI looks like, such as: how the widgets are arranged, with which properties (colors, font, etc.) they are initialized, and so on.

- Code belonging to the *model* is responsible for filling the content of the widgets -- how the tables, lists, trees, and similar components are filled.

- Code belonging to the *controller* category how the widgets and panels interact with each other -- what happens when a specific button is pressed, for example, or when a specific item in a table is selected.

With the migration of the layout managers and the API mapping of the widgets composing your panel, you have ported the code belonging to the view category. That means that if you comment out all the code of your application that has not yet been ported by using the migration techniques described in the previous panels, and then run your application, a GUI similar to the original Swing application should show up without content and logic because the widgets are still empty and don't trigger actions when the user interacts with them.

In this panel, we will focus on the *controller* code. In Swing, this is mainly made up of AWT/Swing event listeners, triggered by user interactions with the GUI.

The concept of event listeners, like the concept of layout managers, is something that is used by most of the modern GUI toolkits, but which is implemented differently in each toolkit. For example, to trigger an action when the user presses a button, Swing lets you register an `ActionListener` on that button, and implement the method `actionPerformed(ActionEvent)` in the listener itself. If you want to program the same behavior in SWT, you have to register a `SelectionListener` on the button, and implement the method `widgetSelected(SelectionEvent)` in the listener. Even if both toolkits throw the same kind of events, the class hierarchy and the API to catch those events is completely different. Thus, without a good migration technique, porting the controller code of a Swing application would be as tiresome as migrating the widgets themselves if you had to do it by hand.

To solve this problem, we will use the same technique that we used for the API mapping. As we saw in the previous section, this mapping was realized by constructing wrapper classes around SWT controls, with a public API that is similar to Swing's API. These wrapper classes can be improved so that they not only map the methods, but also the events.

To be able to do the event mapping, each wrapper class will store a list of AWT listeners and provide the `add/removeXXXListener(XXX)` methods defined in the AWT/Swing API of the widget to port. Additionally, the wrapper class will listen to the SWT events thrown by the wrapped SWT control. When an SWT event is detected -- for example, when a `SelectionEvent` is detected after a button has been pressed -- the corresponding AWT event is created and thrown to the AWT listeners that have been registered in the wrapper class.

By using this technique, you need not migrate your AWT/Swing listeners. The event mapping is programmed once in the wrapper classes, so the AWT/Swing listeners are notified when the user interacts with the SWT control.

The following code snippet illustrates how such an event mapping can be implemented. This is a snippet of the wrapper class corresponding to Swing's `AbstractButton`. The only event that is mapped here is the `SelectionEvent` of SWT, which has to be converted into an AWT `ActionEvent` and thrown to the registered `ActionListeners`.

```
public class SWTAbstractButton
  extends SWTComponent
  implements SelectionListener{

  public SWTAbstractButton(Button button) {
    super(button);
    button.addSelectionListener(this);
  }

  public Button getButton() {
    return (Button)getControl();
  }

  (...)

  public void setAction(Action action) {
    String actionName = (String)action.getValue(Action.NAME);
    if (actionName != null)
      getButton().setText(actionName);
    addActionListener(action);
  }

  public void addActionListener(ActionListener listener) {
    eventListenerList.add(ActionListener.class, listener);
  }

  public void removeActionListener(ActionListener listener) {
    eventListenerList.remove(ActionListener.class, listener);
  }

  //--------------- implementation of SelectionListener ------------------

  public void widgetDefaultSelected(SelectionEvent e) {}

  public void widgetSelected(SelectionEvent e) {
    // propagate an Action event to the ActionListeners
    ActionEvent actionEvent = null;
    EventListener[] actionListeners =
      eventListenerList.getListeners(ActionListener.class);
    for (int i = 0; i < actionListeners.length; i++) {
      if (actionEvent == null)
        actionEvent =
          new ActionEvent(
            this,
            ActionEvent.ACTION_PERFORMED,
            getButton().getText());

      ((ActionListener)actionListeners[i]).actionPerformed(actionEvent);
    }
  }
```

This snippet was only a part of the complete implementation. The complete source code of `SWTAbstractButton`is in the j-swing2swtsrc.zip download in Resources on page 95 .

The important parts of the code are formatted in bold:

- In the constructor `SWTAbstractButton(Button)`, the wrapper class registers itself as an SWT selection listener on the wrapped SWT component.
- AWT `ActionListeners` can be registered and deregistered in the wrapper class by

using the methods add/removeActionListener(ActionListener). The listeners are stored in a protected EventListenerList declared in the superclass SWTComponent. The list of listeners in SWTComponent is declared as follows:

```
protected EventListenerList eventListenerList = new EventListenerList();
```

- When the user presses the SWT button, the method widgetSelected(SelectionEvent) is invoked by SWT. The core of this method checks to see if some AWT ActionListeners are registered, builds an equivalent AWT ActionEvent, and notifies all the registered AWT listeners.

By implementing the event mapping in the wrapper classes, we make the migration of Swing code using listeners straightforward. Consider the following Swing code:

```
JButton button = new JButton("OK");
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    // do action
  }
});
```

It will be migrated as follows (the modified parts are in bold):

```
SWTButton button = new SWTButton(parent, "OK");
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    // do action
  }
});
```

The wrapper classes provided with this tutorial's sample code (see Resources on page 95) already perform event mappings for most of the events you may have used in your Swing application. If your project listens for events that are not mapped in the sample code, feel free to use the code as basis for implementing the missing features in your project.

For more information on the individual Swing components' wrapper classes, read the sections describing the migration of the various components later in this tutorial -- see Widgets on page 31 to get an overview.

Swing's models adapters: Reuse your Swing data models in SWT widgets

Now we are going to focus on the port of the data models used by your Swing application.

In Data models and cell renderers vs. content providers and label providers on page 12 , we saw that JFace's viewers allow you to separate the data model used to fill a widget from the widget itself, just as Swing's data models do. However, although the basic idea is the same, the API and the way JFace's viewers and content providers work is quite different from Swing's TableModel, TreeModel, or ListModels.

At first glance, it looks like these differences will oblige you to entirely migrate your Swing data models to JFace content providers. This conversion could be tedious, especially if your code uses customized model classes that extract the data from an external source.

Fortunately, it's not necessary to migrate the models themselves. We have seen previously that the content providers used by JFace's viewers are not data containers, but rather are data extractors. Content providers can be written to extract data from any kind of input object, including a Swing data model. It is quite easy to write a custom content provider to extract data from any kind of Swing data model and fill an SWT component with that data.

The following code snippet demonstrates how a JFace `TableViewer` can be filled with data from a Swing `TableModel`:

```
public class SWTTable extends SWTComponent
   implements TableColumnModelListener, ListSelectionListener, SelectionListener,
              PropertyChangeListener, ControlListener {

  /** SWT's TableViewer on the table component */
  private TableViewer tableViewer;

  /** Swing's TableModel */
 private TableModel model;
  (...)
  //---------------------------------------------

  public SWTTable(Table table) {
    super(table);
    table.addSelectionListener(this);
    tableViewer = new TableViewer(table);
 tableViewer.setContentProvider(new TableModelContentProvider());
    tableViewer.setLabelProvider(new TableModelLabelProvider());
    (...)
  }

  public SWTTable(SWTContainer parent) {
    this(parent, new DefaultTableModel());
  }

  public SWTTable(SWTContainer parent, TableModel model) {
    this(parent, model, null);
    setColumnModel(createDefaultColumnModel());
    createDefaultColumnsFromModel();
  }

  public SWTTable(SWTContainer parent, Vector rowData, Vector columnNames) {
    this(parent, new DefaultTableModel(rowData, columnNames));
  }
  (...)
  //----

  public final Table getTable() {
    return (Table)getControl();
  }

  public final TableViewer getTableViewer() {
    return tableViewer;
  }

  public final TableModel getModel() {
    return model;
  }
```

```
public final void setModel(TableModel model) {
    this.model = model;
    tableViewer.setInput(model);
    createDefaultColumnsFromModel();
  }

  public final void setValueAt(Object value, int row, int column) {
    getModel().setValueAt(value, row, convertColumnIndexToModel(column));
    TableViewer tv = getTableViewer();
    tv.refresh(tv.getElementAt(column), false);
  }
  (...)
  //------------------------------------------------------------------

  /**
   * Content Provider taking as input the content of the Swing TableModel
   */
 private class TableModelContentProvider
    implements IStructuredContentProvider {
    public Object[] cachedElements;

    /**
     * Takes as argument the current Swing TableModel used by the table
     * and returns an array of Vectors containing the content of the model.
     * Each vector represents a row in the model.
     */
    public Object[] getElements(Object inputElement) {
      if (cachedElements != null)
        return cachedElements;

      if (inputElement instanceof TableModel) {
        TableModel tm = (TableModel)inputElement;
        Vector[] rows = new Vector[tm.getRowCount()];
        for (int i = 0; i < tm.getRowCount(); i++) {
          rows[i] = new Vector(tm.getColumnCount());
          for (int j = 0; j < tm.getColumnCount(); j++) {
            rows[i].add(tm.getValueAt(i, j));
          }
        }
        cachedElements = rows;
      }
      return cachedElements;
    }

    public void dispose() {
      cachedElements = null;
    }

    public void inputChanged(Viewer viewer, Object oldInput, Object newInput) {
      cachedElements = null;
    }
  }

  /**
   * LabelProvider delegating the formatting to a SWTCellRenderer
   */
 private class TableModelLabelProvider implements ITableLabelProvider {
    public Image getColumnImage(Object element, int columnIndex) {
        return null;
    }

   public String getColumnText(Object element, int columnIndex) {
      if (element instanceof Vector) {
```

```
          Object item = ((Vector)element).get(columnIndex);
          if (item == null) return "";
          else return item.toString();
        } else return "";
      }

      public void addListener(ILabelProviderListener listener) {}
      public void dispose() {}
      public boolean isLabelProperty(Object element, String property) {
        return true;
      }
      public void removeListener(ILabelProviderListener listener) {}
    }
    (...)
}
```

This snippet is only a part of the complete implementation of the wrapper class for `JTable`. The complete source code for `SWTTable` is in the j-swing2swtsrc.zip download in Resources on page 95

The important part of the code is in bold:

- When the wrapper class is constructed on top of an SWT table, it automatically creates a JFace viewer on it and sets a customized content provider and a label provider.
- When `setModel(TableModel)` is invoked to set the Swing model, the model is passed to the content provider as new input, so that the rows of the table are reconstructed to display the new model.
- The inner class `TableModelContentProvider` is the most interesting part of the code. It does the conversion between the Swing TableModel API and the API of the JFace content provider. The method `getElements(Object)` returns an array of vectors; each vector contains the data for a single row of the model. For performance reasons, the extracted rows are cached until a new model is used.
- The inner class `TableModelLabelProvider` extracts (from the row vector provided by the content provider) the elements contained in each cell of the row, and converts them to the string to be displayed in the SWT table. We will see in the next panel how this label provider can be improved to have functionality similar to Swing's cell renderers.

If you look in the wrapper classes `SWTList` and `SWTTree`, you will see how a similar method can be used to adapt Swing `ListModels` and `TreeModels` to JFace viewers. These classes are available in the source code in Resources on page 95 .

Migrate Swing's cell renderers and editors

In Data models and cell renderers vs. content providers and label providers on page 12 we saw that the JFace equivalent for a Swing cell renderer is a *label provider.* Label providers are not as flexible as Swing cell renderers, because they don't allow you to use any kind of component to render a cell. In SWT, a table, tree, or list cell is basically represented as a label with an image and text. This means that complicated Swing renderers can't be migrated easily to SWT.

Fortunately, in most cases the renderers that are used in Swing applications are themselves some kind of labels, with text and an icon. This kind of renderer can be converted easily into a JFace label provider.

From my experience as a Swing programmer, the typical scenario in which customized cell renderers are used with tables, trees, or lists goes something like this:

- The data model contains non-`String` objects, which have to be represented in the application with a `String` that is different from the `String` returned by the `toString()` method. Typical examples are:
  - The data are `Numbers` or `Dates` and have to be formatted with a `NumberFormat` or a `DateFormat`, the formatting being locale dependent.
  - The data are objects that can't be easily represented with a string, and a custom icon has to be used. Typical examples are `Colors` or boolean values.
- A default cell renderer in Swing is subclassed (`DefaultListCellRenderer` for a list, `DefaultTreeCellRenderer` for a tree, or `DefaultTableCellRenderer` for a table). These default cell renderers are subclasses of `JLabel`. The newly created custom renderer simply converts the object to be formatted into a `String` and an icon, and sets them with `setText(String)` and `setIcon(Icon)`, like in a normal `JLabel`.

Here's an example of such a custom renderer:

```
TableCellRenderer customRenderer = new DefaultTableCellRenderer() {
  public Component getTableCellRendererComponent(JTable table,
                                                 Object value,
                                                 boolean isSelected,
                                                 boolean hasFocus,
                                                 int row,
                                                 int column) {
    if (value instanceof Date) {
      DateFormat formatter = DateFormat.getDateInstance(DateFormat.SHORT);
      String text = formatter.format((Date)value);
      setText(text);
    } else setText(value.toString());
    return this;
  }
};
```

Such a cell renderer is easy to rewrite as a JFace label provider:

```
ILabelProvider labelProvider = new LabelProvider() {
  public Image getImage(Object element) {
    return null;
  }

  public String getText(Object element) {
    if (element instanceof Date) {
      DateFormat formatter = DateFormat.getDateInstance(DateFormat.SHORT);
      return formatter.format((Date)value);
    } else return value.toString();
  }
```

```
};
```

As you can see, the migration of a single cell renderer to a JFace label provider doesn't present much difficulty for standard renderers, even if the code has to be slightly modified.

The problem is more complicated when you try to migrate a `JTable` using different renderers for each column. Swing's `JTable` allows you to set a different renderer for each column, as well as several default renderers, depending on the type of the data.

On the other hand, a JFace `TableViewer` uses a single `ITableLabelProvider` to format all the cells of a column. `ITableLabelProvider` is a subclass of `ILabelProvider` and provides two methods to return the text and image of a specific column for a specific row:

- `public Image getColumnImage(Object element, int columnIndex);`
- `public String getColumnText(Object element, int columnIndex);`

While the migration of several `TableCellRenderers` used by a `JTable` into a single `ITableLabelProvider` used by a `TableViewer` is technically possible, it can be tedious work to analyse the Swing code to find out which renderers are used by which column indices. A better solution is to:

- Create a class simulating the behavior of Swing's `TableCellRenderer`.
- Extend the wrapper class `SWTTable` so that it can store a separate renderer for each column and data type, like Swing's `JTable`.
- Create a central label provider that asks the table for the renderer to use for a specific cell, and delegate the formatting of a cell to it.

The following code snippet shows you how our renderer class could be implemented:

```
public class SWTCellRenderer
  implements TableCellRenderer {

  public Component getTableCellRendererComponent(
    JTable table,
    Object value,
    boolean isSelected,
    boolean hasFocus,
    int row,
    int column) {
    return null;
  }

 public String getCellText(Object value, int row, int column) {
    return value.toString();
  }

  public Image getCellImage(Object value, int row, int column) {
    return null;
  }
}
```

Get the complete source code for this `SWTCellRenderer` from /swing2swt/components/SWTCellRenderer.java in the j-swing2swtsrc.zip download in Resources on page 95 . The class provides a dummy implementation of Swing's `TableCellRenderer`, so that an instance of `SWTCellRenderer` can be stored in an instance of Swing's `TableColumn`. The two methods `getCellText(Object, int, int)` and `getCellImage(Object, int, int)` have to be overridden so that the renderer can do the formatting that it should do.

The wrapper class for `Table` is modified as follows:

```
public class SWTTable extends SWTComponent
                      implements  TableColumnModelListener, ...
  (...)
  /** Swing's column model */
  private TableColumnModel columnModel;

  /**
   * Hashtable storing the cell renderers to use for each data types
   */
  private Hashtable defaultRenderers = null;
  (...)
  //---------------------------------------------

  public SWTTable(Table table) {
    super(table);
    table.addSelectionListener(this);
    tableViewer = new TableViewer(table);
    tableViewer.setContentProvider(new TableModelContentProvider());
 tableViewer.setLabelProvider(new TableModelLabelProvider());
    (...)
  }
  (...)

 public SWTCellRenderer getCellRenderer(int row, int column) {
    TableColumnModel cm = getColumnModel();
    if (cm != null) {
      Object renderer = cm.getColumn(column).getCellRenderer();
      if (renderer instanceof SWTCellRenderer)
        return (SWTCellRenderer)renderer;
    }
    return getDefaultRenderer(getColumnClass(column));
  }
  (...)

  public final SWTCellRenderer getDefaultRenderer(Class columnClass) {
    if (defaultRenderers == null || columnClass == null)
      return null;
    SWTCellRenderer renderer =
      (SWTCellRenderer)defaultRenderers.get(columnClass);
    if (renderer != null)
      return renderer;

    // if a renderer was not found for this specific class, try recursively
    // to find a renderer for one of the superclasses
    return getDefaultRenderer(columnClass.getSuperclass());
  }
  (...)

  public final void setDefaultRenderer(
```

```
    Class columnClass,
    SWTCellRenderer cellRenderer) {
    if (defaultRenderers == null)
      defaultRenderers = new Hashtable();
    defaultRenderers.put(columnClass, cellRenderer);
    getTableViewer().refresh();
  }
  (...)
  //---------------------------------------------------------------
/**
   * LabelProvider delegating the formatting to a SWTCellRenderer
   */
  private class TableModelLabelProvider implements ITableLabelProvider {
    public Image getColumnImage(Object element, int columnIndex) {
      if (element instanceof Vector) {
        Object item =
          ((Vector)element).get(convertColumnIndexToModel(columnIndex));
        if (item == null)
          return null;
        else {
          // get the renderer for this column
          int rowIndex = getRowIndex(element);
          SWTCellRenderer renderer = getCellRenderer(rowIndex, columnIndex);
          if (renderer != null)
            return renderer.getCellImage(item, rowIndex, columnIndex);
          else
            return null;
        }
      } else
        return null;
    }

    public String getColumnText(Object element, int columnIndex) {
      if (element instanceof Vector) {
        Object item =
          ((Vector)element).get(convertColumnIndexToModel(columnIndex));
        if (item == null)
          return "";
        else {
          // get the renderer for this column
          int rowIndex = getRowIndex(element);
          SWTCellRenderer renderer = getCellRenderer(rowIndex, columnIndex);
          if (renderer != null)
            return renderer.getCellText(item, rowIndex, columnIndex);
          else
            return item.toString();
        }
      } else
        return "";
    }

    public void addListener(ILabelProviderListener listener) {}

    public void dispose() {}

    public boolean isLabelProperty(Object element, String property) {
      return true;
    }

    public void removeListener(ILabelProviderListener listener) {}
  }
  (...)
}
```

For the complete source code for this `SWTTable`, see /swing2swt/components/SWTTable.java in the j-swing2swtsrc.zip download from Resources on page 95 . Like Swing's `JTable`, this object provides a `setDefaultRenderer(...)`, allowing you to register different renderers for different column types. Like `JTable`, it uses a Swing `TableColumnModel` to store a renderer in each column. The inner class `TableModelLabelProvider` searches for the cell renderer that has to be used for a specific column, and delegates the formatting of a cell value to it.

Yoy can use the same method for cell editors. The class `SWTCellEditor` (/swing2swt/components/SWTCellEditor.java) is a wrapper class allowing you to emulate the Swing API with JFace `CellEditors`.

Section 5. Widgets

Overview

In this section, we'll see how to translate each Swing component into a corresponding SWT component in our framework. The following table gives you an overview of the correspondence between the components in the two toolsets. For each Swing component listed in the first column, you can read in the second column the name of the equivalent SWT component, as well as the eventual style constants to use. The third column contains a link to the panel where the migration issues of the component are explained in detail.

Swing	SWT	Panel
JButton	Button (Style=SWT.PUSH)	JButton, JToggleButton, JCheckBox, and JRadioButton on page 33
JCheckBox	Button (Style=SWT.CHECK)	JButton, JToggleButton, JCheckBox, and JRadioButton on page 33
JCheckBoxMenuItem	MenuItem (Style=SWT.CHECK)	JMenu, JPopupMenu, and JMenuItem on page 48
JColorChooser	ColorDialog	JColorChooser on page 35
JComboBox	Combo or CCombo	JComboBox on page 36
JDesktopPane	No equivalent in SWT; use GEF if needed	JDesktopPane, JInternalFrame, JLayeredPane, and JRootPane on page 39
JEditorPane	StyledText	JEditorPane on page 39
JFileChooser	FileDialog or DirectoryDialog	JFileChooser on page 40
JInternalFrame	No equivalent in SWT; use GEF if needed	JDesktopPane, JInternalFrame, JLayeredPane, and JRootPane on page 39
JLabel	Label or CLabel	JLabel on page 42
JLayeredPane	No equivalent in SWT; use GEF if needed	JDesktopPane, JInternalFrame, JLayeredPane, and JRootPane on page 39
JList	List	JList on page 44
JMenu	Menu, or MenuItem (Style=SWT.CASCADE) if in a menu	JMenu, JPopupMenu, and JMenuItem on page 48
JMenuBar	Menu (Style=SWT.BAR)	JMenu, JPopupMenu, and JMenuItem on page 48
JMenuItem	MenuItem (Style=SWT.PUSH)	JMenu, JPopupMenu, and JMenuItem on page 48
JOptionPane	MessageDialog or InputDialog	JOptionPane on page 51

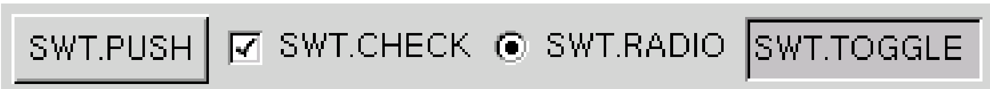
JPanel	Composite or Group	JPanel on page 53
JPasswordField	Text (Style=SWT.SINGLE); use setEchoChar(char)	JTextField, JTextArea, and JPasswordField on page 72
JPopupMenu	Menu (Style=SWT.POP_UP)	JMenu, JPopupMenu, and JMenuItem on page 48
JProgressBar	ProgressBar, ProgressIndicator, or ProgressMonitorDialog	JProgressBar on page 55
JRadioButton	Button (Style=SWT.RADIO)	JButton, JToggleButton, JCheckBox, and JRadioButton on page 33
JRadioButtonMenuItem	MenuItem (Style=SWT.RADIO)	JMenu, JPopupMenu, and JMenuItem on page 48
JRootPane	No equivalent in SWT; use GEF if needed	JDesktopPane, JInternalFrame, JLayeredPane, and JRootPane on page 39
JScrollPane	ScrolledComposite (Style=SWT.H_SCROLL | SWT.V_SCROLL)	JScrollPane and JViewport on page 57
JSeparator	Label (Style=SWT.SEPARATOR), or MenuItem (Style=SWT.SEPARATOR) if in a menu	JSeparator on page 60
JSlider	Slider or Scale	JSlider on page 60
JSplitPane	SashForm	JSplitPane on page 62
JTabbedPane	TabFolder or CTabFolder	JTabbedPane on page 64
JTable	Table	JTable on page 67
JTableHeader	No equivalent; use Table.setHeaderVisible(true)	JTable on page 67
JTextArea	Text (Style=SWT.MULTI)	JTextField, JTextArea, and JPasswordField on page 72
JTextField	Text (Style=SWT.SINGLE)	JTextField, JTextArea, and JPasswordField on page 72
JTextPane	StyledText	JEditorPane on page 39
JToggleButton	Button (Style=SWT.TOGGLE)	JButton, JToggleButton, JCheckBox, and JRadioButton on page 33
JToolBar	ToolBar or CoolBar	JToolBar on page 75
JToolTip	No equivalent; use Control.setToolTipText(String)	-
JTree	Tree	JTree on page 77

`JViewport`	`ScrolledComposite (Style=SWT.NONE)`	JScrollPane and JViewport on page 57

JButton, JToggleButton, JCheckBox, and JRadioButton

The equivalent of these four Swing components is a single SWT component: `Button`, shown in the image below. Instead of using different classes to represent the different types of buttons, SWT uses different type constants, which are passed as parameters in the constructor of the component:

- `SWT.PUSH` is used to create a push button like a `JButton`.
- `SWT.TOGGLE` is used to create a two-state push button like a `JToogleButton`.
- `SWT.CHECK` is used to create a checkbox like a `JCheckBox`.
- `SWT.RADIO` is used to create a radio button like a `JRadioButton`.



Text and icon

As in Swing, SWT buttons can contain text and/or an image. (Note that on some platforms, such as Motif, you can't display text and an image in the same button. If you try, the image will simply be ignored.) However, in SWT you can't define different images for the different states of the button as you can in Swing. The alignment of the text and image of the button can be defined in the constructor by combining the style `SWT.LEFT` or `SWT.RIGHT` with the type of the button, or by invoking `setAlignment(int)` after the creation of the button.

Keyboard navigation

Mnemonics -- the underlined characters that can be used as keyboard shortcuts to activate buttons -- are not set by invoking `setMnemonic(char)` as in Swing, but simply by adding an ampersand character (&) in the text of the button at the position before the mnemonic character, like so:

```
button.setText("&Execute");
```

Events

Where Swing's buttons throw three kinds of event -- an `ActionEvent`, indicating that an action has been performed, an `ItemEvent`, indicating that the state of a toggle button has changed, and a `ChangeEvent`, whose role is not really clearly defined -- SWT only uses one event: `SelectionEvent`.

To detect when a button is pressed, or when the state of a toggle button, check box, or radio button has changed, just use the `addSelectionListener(SelectionListener)` method and implement the interface `SelectionListener` or subclass `SelectionAdapter`. Then, implement the method `widgetSelected(SelectionEvent)`. To know the state of the button, just get the source of the event, cast it to `Button`, and invoke the `getSelection()` method.

The following code listing illustrates all of these code concepts in action:

```
//--- Creation of a push button with a left aligned text
Button button = new Button(parent, SWT.PUSH | SWT.LEFT);
button.setText("&Button Text");
// Trigger an action when the button is pressed
button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent event) {
      System.out.println("Button "+event.getSource()+" pressed");
    };
});

//--- Creation of a radio button
Button radioButton = new Button(parent, SWT.RADIO);
radioButton.setText("&RadioButton Text");
// Trigger an action when the state of the radio button changes
button.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent event) {
      Button b = (Button)event.getSource();
      if (b.getSelection()) System.out.println("Button "+b+" selected");
      else System.out.println("Button "+b+" deselected");
    };
});
```

Migrate existing Swing code

The migration of Swing buttons to SWT doesn't present any particular difficulty, as the two toolkits offer the same functionality. The sample code provided with this tutorial contains several wrapper classes that make migration easier:

- `SWTAbstractButton`
- `SWTButton`
- `SWTToggleButton`
- `SWTRadioButton`
- `SWTCheckBox`

These wrapper classes use the API and event mapping introduced in Migrate your Swing code to SWT with minimal change on page 14 , so the migration work you'll have to do is limited to the following simple steps:

- Search for occurrences of the Swing types and replace them with the new wrapper type.
- Search for constructors where a button is created and add the reference to the parent of the button in the arguments list.

Here's an example of such a migration. Consider the following Swing code:

```
Action myAction = ...;
JButton button1 = new JButton(myAction);
parent.add(button1);

JButton button2 = new JButton("Button 2");
button2.addActionListener(anActionListener);
parent.add(button2);

JCheckBox checkBox = new JCheckBox("CheckBox", true);
parent.add(checkBox);
```

Here's what this code would look like after being migrated to SWT:

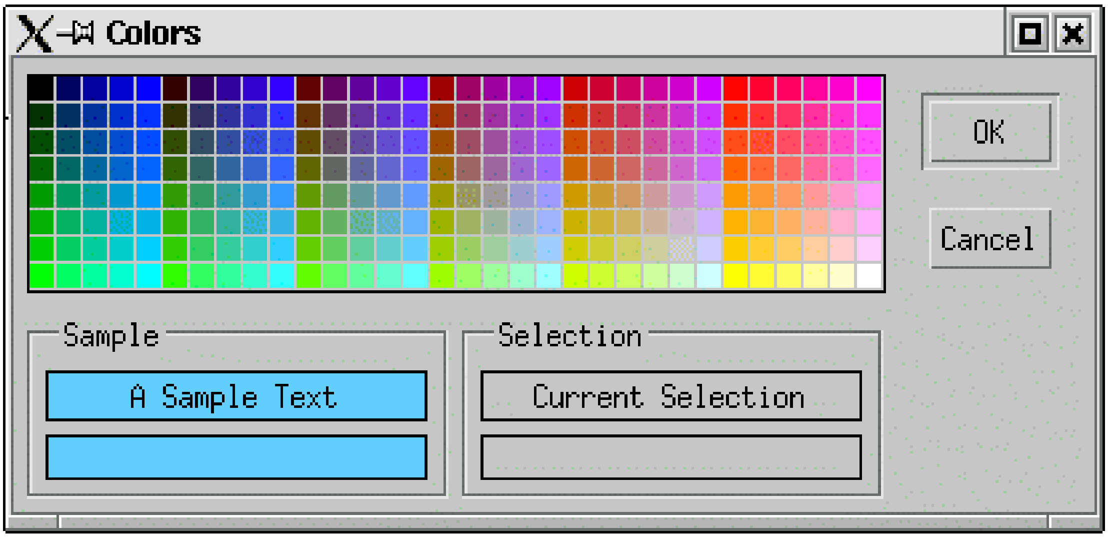
```
Action myAction = ...;
SWTButton button1 = new SWTButton(parent, myAction);
parent.add(button1);

SWTButton button2 = new SWTButton(parent, "Button 2");
button2.addActionListener(anActionListener);
parent.add(button2);

SWTCheckBox checkBox = new SWTCheckBox(parent, "CheckBox", true);
parent.add(checkBox);
```

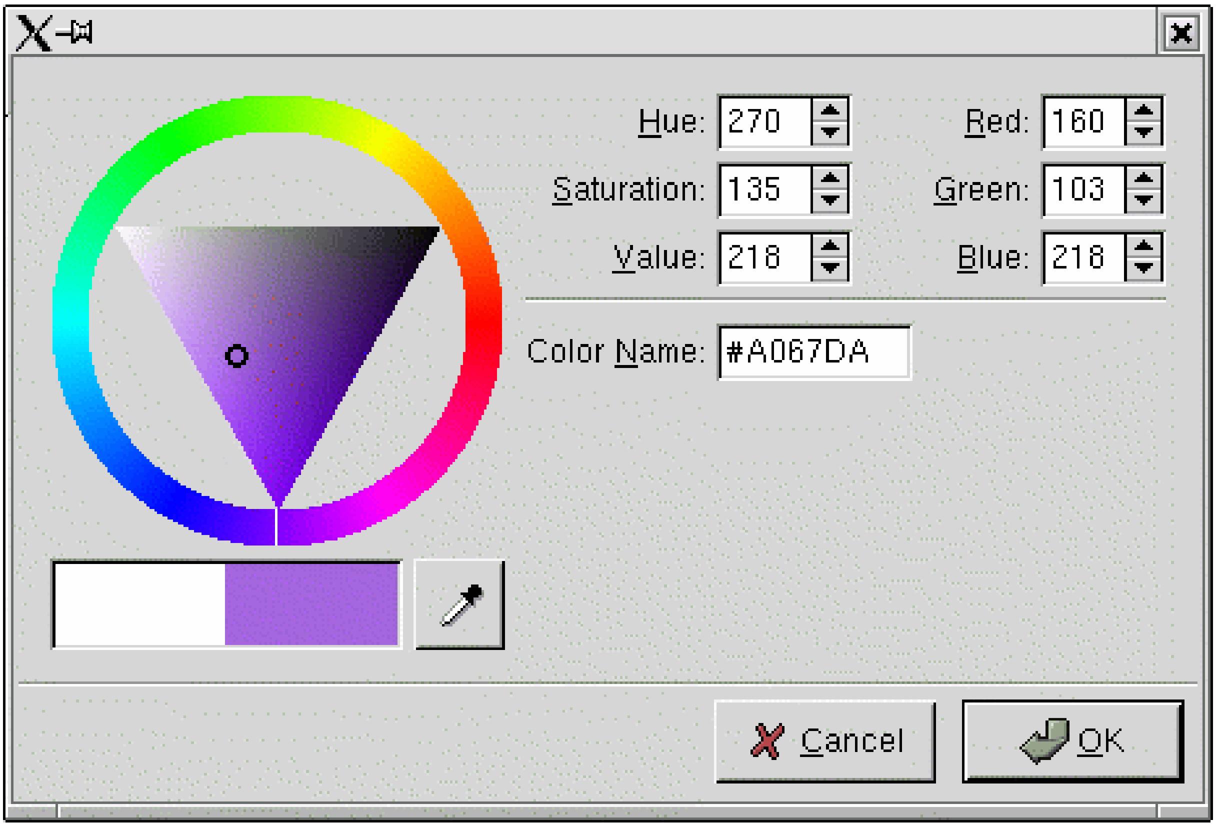
JColorChooser

SWT provides the standard dialog `ColorDialog` to choose a color. Its API is very simple and only contains three methods: `setRGB(RGB)`, `open()`, and `getRGB()`.

The dialog is a system dialog. This means that its look and feel is different for each platform, and that you can't customize it.

The following screenshots shows what the color dialog looks like under Motif and GTK, respectively:





SWT doesn't provide any color chooser control that can be embedded in a panel.

JComboBox

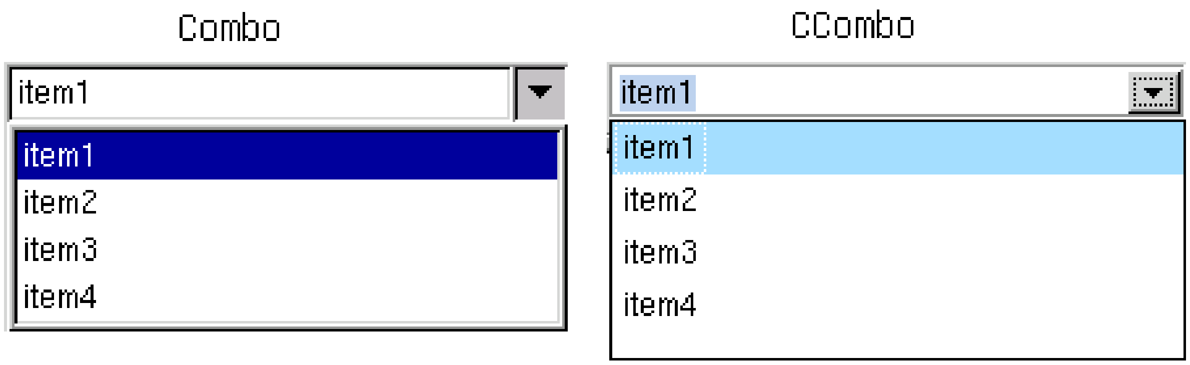
Combo boxes can be created in SWT either by using the component `Combo`, which is mapped to a native widget, or by using the customized widget `CCombo`, located in the `org.eclipse.swt.custom` package.

The APIs for both components are nearly identical. Most of the time, you will want to use `Combo` in order to have a native component with better performance and a standard look and feel. `CCombo` allows you to customize the look of the control and should only be used in special cases where a native component is not suitable.

There are two possible reasons why you might prefer a `CCombo` to a native `Combo`:

- *You need a combo box without any border:* Native `Combo`s are always drawn with a border. By using a `CCombo` with the style constant `SWT.FLAT`, you get a combo box without any border. This can be useful if the combo box is added to another component having its own border. To create a `CCombo` with a border similar to `Combo`, use the style constant `SWT.BORDER`.

- *You need a more compact combo box:* On some platforms, such as Motif, even the smallest native combo box is too large to be added to another component, such as a toolbar. Using a `CCombo` allows you to get a component whose minimum size is the same on all platforms and which is compact enough to fit in a toolbar.



Items

Unlike Swing's `JComboBox`, SWT combos can only contain normal `Strings` without any icons. The renderer mechanism of Swing that allowed you to put any kind of objects into a combo box model and render them in a customized way is not available in SWT. SWT doesn't use a separate model class to store the items and selection of the combo box as Swing does.

To set the items in the combo box, use the method `setItems(String[])`. To append or insert an item at a specific position, use `add(String)` or `add(String, int)`. To remove items, use one of the several `remove` methods.

Editable vs. read-only combos

As is true in Swing, an SWT combo is made up of a text field and a list. The text field can be either freely editable -- that is, the user can enter a value that is not available in the list -- or read only -- that is, the user can only select a value already available in the list. In Swing's `JComboBox`, you can control this feature by using the `setEditable(boolean)` method after the creation of the widget. In SWT, you have to use the style `SWT.READ_ONLY` in the constructor of the component if you want it to be read only.

Management of the selection

The currently selected item in a combo can be retrieved by using any of several methods:

- `getSelectionIndex()` returns the index of the currently selected item. If the combo is not read only, and the user enters text that is not in the list of the items, this method will return -1.

- `getText()` returns the current text of the field of the combo. If the combo is read only, it corresponds to the currently selected item in the list.

Be careful not to mix up `getSelectionIndex()` and `getSelection()`. The latter returns a `Point` containing the start and end position of the character selection of the text field of the combo. It is the equivalent of `Text.getSelection()` and has nothing to do with the item selection of the combo.

You can set the selection by using one of these methods:

- `select(int)` selects the item at a specific position.
- `setText(String)` sets the text to display in the field of the combo.

Do not mix up these methods with `setSelection(Point)`, which is the equivalent of `Text.setSelection(point)` and is used to set the character selection in the field of the combo.

Events

Two kind of events are thrown by an SWT combo:

- A `SelectionEvent` is thrown when the user chooses an item in the list of the combo. To detect a change in the selection, register a `SelectionListener` by using the method `addSelectionListener(SelectionListener)`. The listener method that is triggered by the event and should be implemented is `SelectionListener.widgetSelected(SelectionEvent)`.
- A `ModifyEvent` is thrown when the text in the field of the combo changes. This event is the same as the event thrown by the `Text` component. To learn more about `ModifyEvents`, read JTextField, JTextArea, and JPasswordField on page 72 . Note that, unlike `Text`, a combo box doesn't throw `VerifyEvents`.

The following code snippet illustrates SWT combo boxes in action:

```
//--- Creation of a read-only combo box containing 3 items
Combo combo = new Combo(parent, SWT.DROP_DOWN | SWT.READ_ONLY);
combo.setItems(new String[]{"item1", "item2", "item3"});
//--- Detect changes in the selection
combo.addSelectionListener(new SelectionAdapter() {
  public void widgetSelected(SelectionEvent e) {
    // trigger action
  }
});
//--- Get the current selected item
String selectedItem = combo.getText();
```

Migrate existing Swing code

The migration of existing Swing code is not problematic for combo boxes that only contain `String` items and don't use special renderers. If this is not the case, you have to replace the Swing renderer with a kind of label provider that converts the items into `Strings` before they are added in the combo. Icons are not supported.

The wrapper class `SWTComboBox`, included with the sample code provided with this tutorial, makes the migration easier. It uses the API and event mapping introduced in Migrate your Swing code to SWT with minimal change on page 14 , so that the migration work you'll have to do is limited to a few simple steps:

- Search for occurrences of the Swing type `JComboBox` and replace them with the new wrapper type `SWTComboBox`.
- Search for constructors where a combo box is created and add the reference to the parent of the combo box in the arguments list.

Let's look at a migration example. Consider the following Swing code:

```
String[] items = new String[]{"item1", "item2", "item3", "item4"};
JComboBox comboBox = new JComboBox(items);
comboBox.setAction(new AbstractAction() {
  public void actionPerformed(ActionEvent e) {
    // do action...
  }
});
parent.add(comboBox);
```

Here's what it would look like migrated to SWT:

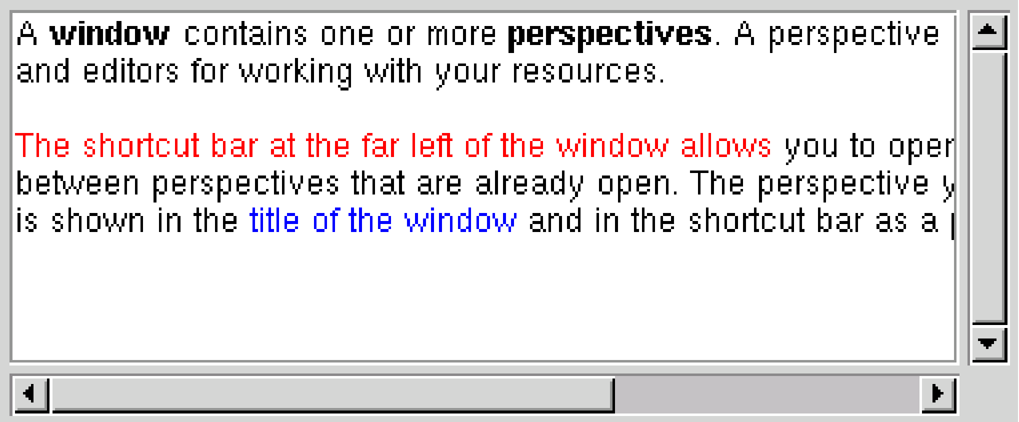
```
String[] items = new String[]{"item1", "item2", "item3", "item4"};
SWTComboBox comboBox = new SWTComboBox(parent, items);
comboBox.setAction(new AbstractAction() {
  public void actionPerformed(ActionEvent e) {
    // do action...
  }
});
parent.add(comboBox);
```

JDesktopPane, JInternalFrame, JLayeredPane, and JRootPane

Because SWT components are native components that don't support transparency, there is no direct SWT equivalent for Swing's `JRootPane` and `JLayeredPane`. As of version 2.1 of the toolkit, there are no multiple document interface (MDI) widgets in SWT like Swing's `JDesktopPane` or `JInternalFrame`. However, the Eclipse sub-project *GEF* provides some of the functionality of these components that is not available in the standard SWT library. GEF is a graphical library that can be used to build graphical SWT applications such as GUI designers and diagram editors. It provides a framework that allows you to build lightweight widgets with support for transparency and multiple layers, like those available in Swing. For more information on GEF, consult Resources on page 95 .

JEditorPane

With `StyledText`, SWT provides a component that is similar to Swing's `JEditorPane` and `JTextPane`. Like a `JEditorPane`, a `StyledText` is a widget that can be used to display and edit text with different font styles and colors, as illustrated below:



Unlike Swing's `JTextPane`, a `StyledText` can only display text. Things like images or tables are not supported. Additionally, SWT has no equivalent for Swing's `EditorKit`, which allows a Swing `JTextPane` to read or write documents in HTML or RTF format.

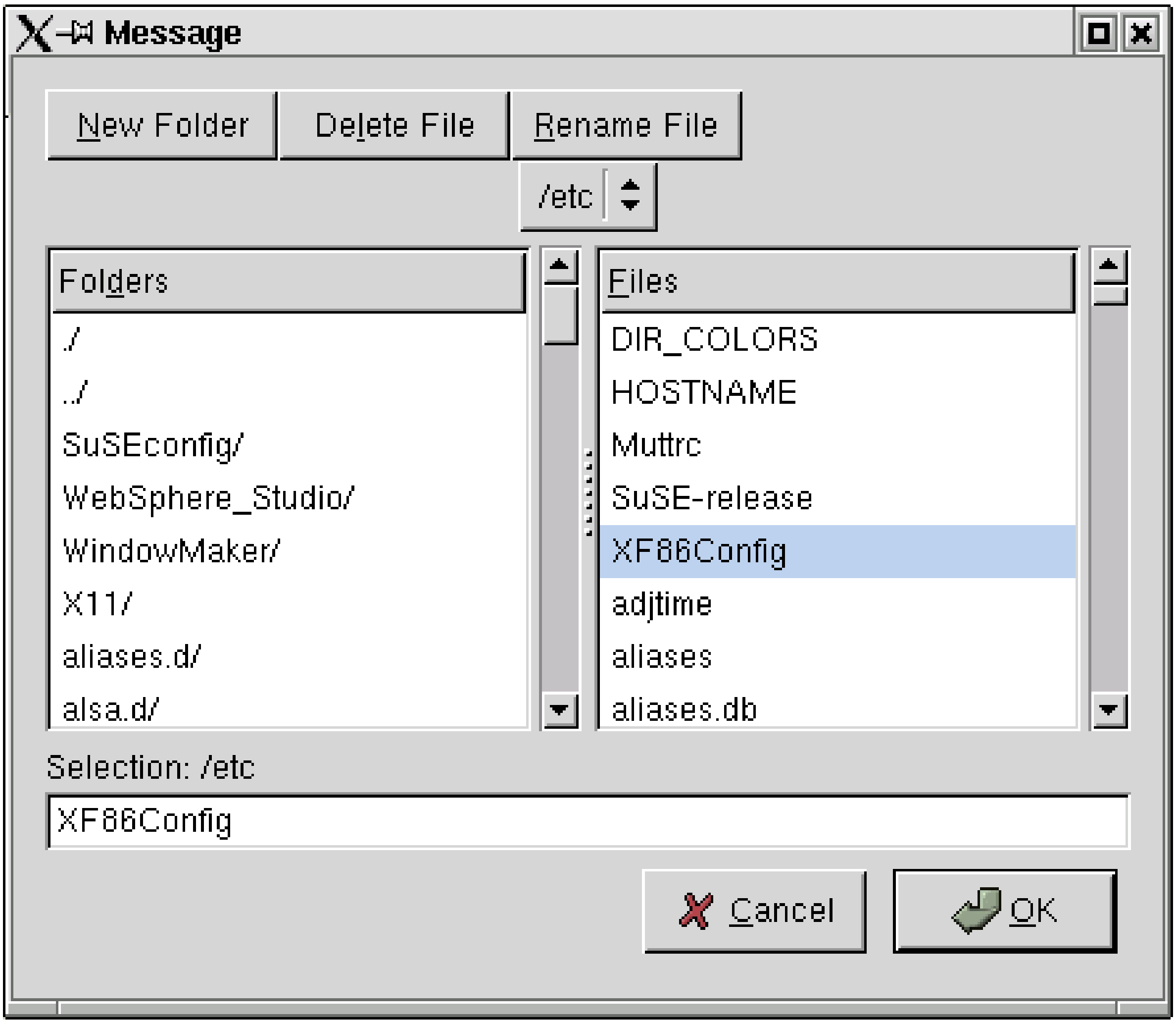
The API and usage of `StyledText` is not covered in detail in this tutorial. To learn more about it, you should read the articles "Getting your feet wet with the SWT `StyledText` widget" and "Into the deep end of the SWT `StyledText` widget" by Lynne Kues and Knut Radloff. You can find links to both in Resources on page 95 .

JFileChooser

SWT provides two dialogs to select files or directories.

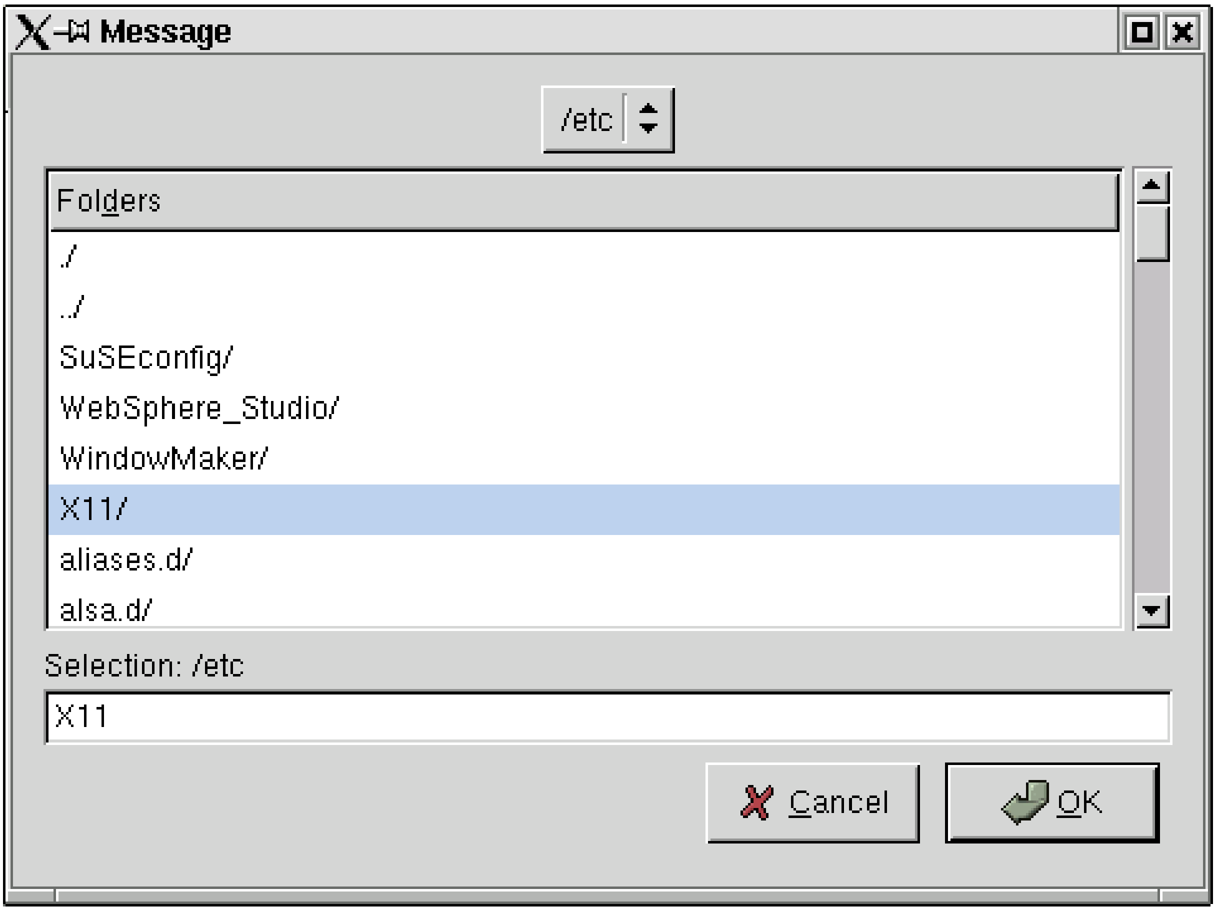
`FileDialog` is a dialog to select a file on the filesystem. You can choose whether the dialog should be used to open or save a file by using one of two type constants, `SWT.OPEN` and `SWT.CLOSE`. Some platforms use different dialogs for open and save operations. You can set the initial directory and filename by invoking `setFilterPath(String)` and `setFileName(String)`, respectively. You can get the selected file after the dialog has been closed by invoking `getFilterPath()` to have the directory of the selected file, or `getFileName()` to get the selected file name. The following code and figure illustrate `FileDialog` in action:

```
FileDialog dialog = new FileDialog(shell, SWT.OPEN);
dialog.setText("Title"); // title of the dialog
dialog.open();
File selectedFile = null;
if (dialog.getFileName()!=null)
    selectedFile = new File(dialog.getFilterPath(), dialog.getFileName());
```



`DirectoryDialog` is a dialog to select a directory on the filesystem. You can set the initial directory by invoking `setFilterPath(String)`. You can get the selection of the user by invoking `getFilterPath()`. The following code and figure show `DirectoryDialog` in action:

```
DirectoryDialog dialog = new DirectoryDialog(shell, SWT.OPEN);
dialog.setText("Title"); // title of the dialog
dialog.open();
File selectedDirectory = null;
if (dialog.getFilterPath()!=null)
    selectedDirectory = new File(dialog.getFilterPath());
```



The look and feel of these dialogs is platform specific. The screenshots shown above were taken under Linux and GTK.

SWT doesn't provide a file chooser component that you can embed in a panel. File and directory choosers are only available as standalone system dialogs, whose look and feel can't be customized.

Migrate existing Swing code

The migration of existing Swing code is not problematic as long as you use standard standalone dialogs to choose a file or a directory. If the `JFileChooser` of your Swing application is embedded in a panel, or if it has been customized to display a preview of the selected file, you will probably have to create your own SWT component.

JLabel

SWT provides two components that can be used as labels: `org.eclipse.swt.widget.Label` and `org.eclipse.swt.custom.CLabel`.

- `Label` uses a native widget of the underlying windowing system and has an API that is quite similar to the API of `Button` (see JButton, JToggleButton, JCheckBox, and JRadioButton on page 33). Like buttons, `Label`s are not very customizable; on some platforms, such as Motif, images and text can not be on the same label at the same time.

- `CLabel` is an emulated widget: It is not a single native widget, but a composition of simpler widgets. It provides more functionality than `Label`, such as support on all platforms for an image and text coexisting, support for additional borders (`SWT.SHADOW_IN` or `SWT.SHADOW_OUT`), and support for using an image or a gradient of color as background.

Most of the time you should use `Label`. It keeps application performance high and is more consistent with the underlying platform for simple labels displaying a simple text or image. For those rare cases in which a normal label is not sufficient (if you need a customized background, for instance), use `CLabel`.

Alignment

You can only set horizontal alignment for `Label` and `CLabel`. You can do this by using one of three styles, `SWT.LEFT`, `SWT.CENTER`, or `SWT.RIGHT`, in the constructor, or by invoking `setAlignment(int)`. You can't control the vertical alignment or the position of the text relative to the icon.

Note that `Label` accepts a style called `SWT.WRAP`, which is not available for `CLabel` and has no equivalent in Swing. When this style is used, the label text is wrapped on several lines if it is longer than the `Label`. `CLabel` uses a strategy similar to Swing's `JLabel` to shorten text that is too long for a label: it replaces a part of the text -- the middle part, unlike `JLabel` -- with an ellipsis (...) to symbolize that there is more to the text than the visible portion.

Mnemonics

As with buttons (see JButton, JToggleButton, JCheckBox, and JRadioButton on page 33), with SWT labels you do not set mnemonics with a special method, as you would in Swing, but by inserting an ampersand character (&) in the text just before the character to use as mnemonic. This functionality is however only available in `Label` and not in `CLabel`.

Borders

`Label` and `CLabel` use different border styles:

- `Label` accepts only one border style: `SWT.BORDER`. The look of the resulting border depends on the platform.
- `CLabel` ignores the style `SWT.BORDER` but accepts two other styles, `SWT.SHADOW_IN` and `SWT.SHADOW_OUT`. The look of these borders is platform independent.

The following code snippet illustrates SWT labels in action.

```
//--- Creation of a simple label with mnemonic on the 1st character
Label label = new Label(parent, SWT.NONE);
label.setText("&Label Text");
//--- Creation of a right aligned label with word-wrapping and border
Label label2 = new Label(parent, SWT.RIGHT | SWT.WRAP | SWT.BORDER);
label2.setText("Right Aligned Label");
```

Migrate existing Swing code

Because labels are simple, non-interacting components, porting from Swing to SWT should not cause any problem.

The wrapper class `SWTLabel`, included in the sample code provided with this tutorial, makes the migration easier. It uses the API and event mapping introduced in Migrate your Swing code to SWT with minimal change on page 14 , so the migration work you'll have to do is limited to a few simple steps:

- Search for occurrences of the Swing type `JLabel` and replace them with the new wrapper type `SWTLabel`.

- Search for constructors where a label is created and add the reference to the parent of the label in the arguments list.

Here's a migration example. Consider the following Swing code:

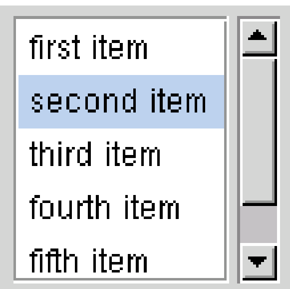
```
JLabel label = new JLabel("Label Text", SwingConstants.CENTER);
```

Here's how you would migrate this code to SWT:

```
SWTLabel label = new SWTLabel(parent, "Label Text", SwingConstants.CENTER);
```

JList

A list is one of the most common widgets that any toolkit must provide. So it is not really surprising that SWT provides nearly the same functionality as Swing's `JList` in a component named `List`, illustrated below:



Although the functionality provided by SWT lists is quite similar to that provided by Swing, there are some small differences you should be aware of.

First, SWT's `List` can only display its elements in a textual form. Icons are not supported, and all the items are displayed with the same background and foreground colors and in the same font. There is no renderer mechanism allowing you to represent an element with any kind of component, as you can in Swing. If you need to represent the elements of the list with icons or variable colors, or if you need a list of checkable items, you may want to use a `Table` with a single column. This will give you more flexibility in the representation of the items, and will offer you the same functionality. For more information on SWT's table, read JTable on page 67 .

In addition, SWT's `List` doesn't use a data model like Swing's `ListModel`. To fill the list, you simply set the items as strings with the `add(...)`, `setItem(String, int)` or `setItems(String[])` methods. However, if you need a separation between the data to display and the string used to represent these data in the list, you can use JFace's `ListViewer` with a content and label provider. The content provider supplies the elements of the list like Swing's `ListModel` does -- and these can be any kind of objects -- while the label provider converts these elements into string representations that are displayed in the `List` by the `ListViewer`. For more information on JFace's viewers, read Data models and cell renderers vs. content providers and label providers on page 12 .

Finally, like many other SWT components, SWT `Lists` are by nature scrollable and don't have to be put in a scroll pane in order to have scrollbars. To make the horizontal and/or vertical scrollbar appear, use in the constructor of the list a bitwise combination of the style constants `SWT.H_SCROLL` and/or `SWT.V_SCROLL`.

Management of the selection

As in Swing, a list can accept either a single selection or multiple selections. In Swing, you have to set this behavior in the `SelectionModel`. SWT lets you control this behavior in the constructor of the component by using one of two style constants: `SWT.SINGLE` or `SWT.MULTI`.

In fact, SWT doesn't have any equivalent for Swing's `SelectionModel`. The methods to set or get the selection in the list are found in the list itself, or in the `ListViewer`:

- SWT's `List` provides simple methods to set or get the selection. These methods work with either the indices of the items comprising the selection, or the displayed strings themselves. The API is easy to use and is simpler than Swing's `ListModel` API.

- JFace's `ListViewer` provides two methods, `getSelection()` and `setSelection(ISelection, boolean)`, that are inherited from `StructuredViewer` and work on a higher abstraction level. The `ISelection` object returned or used by these methods is in fact a `StructuredSelection` that provides an iterator or an array containing the selected elements as provided by the content providers, and is independent from their string representation or their representation order.

Borders

Lists are by default created without any border around them. However, you may often want to use the style `SWT.BORDER` to get a standard border around a list. The appearance of the border depends on the platform.

Events

An SWT list throws only one kind of event. A `SelectionEvent` is thrown to notify the listeners that a change has occurred in the selection. To detect a change in the selection, register a `SelectionListener` by using the `addSelectionListener(SelectionListener)` method. The listener method that is triggered by the event and should be implemented is `SelectionListener.widgetSelected(SelectionEvent)`.

The following code snippet shows SWT lists in action:

```
//--- Creation of a list containing 3 items
List list = new List(parent, SWT.BORDER | SWT.V_SCROLL | SWT.H_SCROLL
                             | SWT.MULTI);
list.setItems(new String[]{"item1", "item2", "item3"});
//--- Detect changes in the selection
list.addSelectionListener(new SelectionAdapter() {
  public void widgetSelected(SelectionEvent e) {
    // trigger action
  }
});
//--- Get the selected items
String[] selectedItems = list.getSelection();

//------ Example of use of a ListViewer ------

//--- Creation of a list displaying 3 java.util.Locale objects
ListViewer listViewer = new ListViewer(parent, SWT.H_SCROLL | SWT.V_SCROLL
                                               | SWT.MULTI | SWT.BORDER);
listViewer.setContentProvider(new ArrayContentProvider());
listViewer.setInput(new Locale[]{Locale.FRANCE, Locale.GERMANY, Locale.US});
//--- Use a label provider displaying the full localized name of the locales
//--- instead of their toString() representation
listViewer.setLabelProvider(new LabelProvider() {
  public String getText(Object element) {
    if (element instanceof Locale) return ((Locale)element).getDisplayName();
    else return element.toString();
  }
});
//--- Detect changes in the selection
listViewer.getList().addSelectionListener(new SelectionAdapter() {
  public void widgetSelected(SelectionEvent e) {
    // get the selection as an array
    StructuredSelection selection=(StructuredSelection)listViewer.getSelection();
    Object[] selectedElements = selection.toArray();
    // trigger action
  }
});
```

Migrate existing Swing code

The migration of existing Swing code is not problematic for lists that only contain `String` items and don't use special renderers. If this is not the case, you have to replace the Swing renderer with a label provider used in combination with a content provider and a `ListViewer`.

The wrapper class `SWTList`, included in the sample code provided with this tutorial, makes the migration easier. It uses the API and event mapping introduced in the section Migrate your Swing code to SWT with minimal change on page 14 , so the migration work you'll have to do is limited to a few simple steps:

- Search for occurrences of the Swing type `JList` and replace them with the new wrapper type `SWTList`.
- Search for constructors where a list is created. Add the reference to the parent of the list as the first argument in the constructor and a boolean indicating if only a single selection is allowed as the second argument.

- It is very likely that the Swing lists of your application are contained in `JScrollPanes`. Modify the code so that no `JScrollPane` is created and the lists are directly added to their parent.
- Convert Swing renderers into `SWTCellRenderers`.

Here's a migration example. Consider the following Swing code:

```
// --- simple list without special renderer
String[] items = new String[]{"item1", "item2", "item3", "item4"};
JList list1 = new JList(items);
list1.getSelectionModel().addListSelectionListener(new ListSelectionListener() {
  public void valueChanged(ListSelectionEvent e) {
    // do action
  }
});
parent.add(list1);

// --- list with customized renderers
Object[] locales = new Object[] {Locale.FRANCE, Locale.GERMANY, Locale.US};
JList list2 = new JList(locales);

ListCellRenderer cellRenderer = new DefaultListCellRenderer() {
  public Component getListCellRendererComponent(JList list,Object value,
      int index, boolean isSelected, boolean cellHasFocus) {
    JLabel label = (JLabel)super.getListCellRendererComponent(list, value,
                                             index, isSelected, cellHasFocus);
    if (value instanceof Locale)
      label.setText(((Locale)value).getDisplayName());
    return label;
  }
};
list2.setCellRenderer(cellRenderer);
parentContainer.add(list2);
```

Here's the same code migrated to SWT:

```
// --- simple list without special renderer
String[] items = new String[]{"item1", "item2", "item3", "item4"};
SWTList list1 = new SWTList(parent, true, items);
list1.getSelectionModel().addListSelectionListener(new ListSelectionListener() {
  public void valueChanged(ListSelectionEvent e) {
    // do action
  }
});
parent.add(list1);

// --- list with customized renderers
Object[] locales = new Object[] {Locale.FRANCE, Locale.GERMANY, Locale.US};
SWTList list2 = new SWTList(parent, true, locales);

SWTCellRenderer cellRenderer = new SWTCellRenderer() {
  public String getCellText(Object value, int row, int column) {
    if (value instanceof Locale) return ((Locale)value).getDisplayName();
    else return value.toString();
  }
};
```

```
list2.setCellRenderer(cellRenderer);
parentContainer.add(list2);
```

JMenu, JPopupMenu, and JMenuItem

SWT has a very simple API to create menus:

- The widget `Menu` is used to create menu bars, menus, and pop-up menus -- the equivalent of Swing's `JMenuBar`, `JMenu`, and `JPopupMenu`.
- The widget `MenuItem` is used to create all kinds of menu items -- the equivalent of Swing's `JMenuItem`, `JCheckBoxMenuItem`, and `JRadioButtonMenuItem`.

The type of the parent passed as a parameter when constructing a `Menu` defines the kind of menu that will be created:

- If the parent is of type `Decorations` -- in most cases it will be a `Shell`, which is the SWT equivalent of an AWT `Window` -- a menu bar will be created. In this case, you have to use the style `SWT.BAR`. Note that the menu bar is added to the window only after the `setMenuBar(Menu)` method has been invoked on the window.
- If the parent is a `Control`, the menu will be a pop-up menu. To display this menu, you have to set its location with `setLocation(int, int)` and then make it visible with `setVisible(boolean)`. Note that the coordinates passed to `setLocation(int, int)` are screen coordinates. Because pop-up menus are usually triggered by a mouse event on the parent component, and the click coordinates stored in the event are component coordinates, you have to convert them to screen coordinates by using `Control.toDisplay(int, int)`.
- If the parent is another `Menu`, the menu created will be a cascading menu. It has to be associated with a `MenuItem` in the parent menu that has the style `SWT.CASCADE`; you would associate it by invoking `setMenu(Menu)` on the `MenuItem`.

You can create different kinds of menu items by using different style constants when constructing your `MenuItems`:

- The style `SWT.PUSH` creates a normal menu item similar to Swing's `JMenuItem`.
- The style `SWT.CHECK` creates a menu item that works like a checkbox, similar to Swing's `JCheckBoxMenuItem`.
- The style `SWT.RADIO` creates a menu item that works like a radio button, similar to Swing's `JRadioButtonMenuItem`.
- The style `SWT.CASCADE` creates a menu item that opens a cascading menu, similar to Swing's `JMenu`. The menu opened by this menu item has to be set with the

`setMenu(Menu)` method.

* The style `SWT.SEPARATOR` creates a menu separator similar to Swing's `JSeparator`.

Text and icon

As in Swing, SWT menu items can contain text and/or an image. Note that some platforms, such as Motif, ignore images.

Keyboard navigation

Mnemonics -- the underlined characters that can be used as key shortcut to activate an item -- are set by adding an ampersand character (&) in the text of the item at the position before the mnemonic character, like so:

```
menuItem.setText("&Run");
```

Accelerators -- the key combination activating the action triggered by the menu item, such as Ctrl-C -- are set by using the method `MenuItem.setAccelerator(int)`. The parameter is a bitwise combination of SWT key constants -- `SWT.CONTROL`, `SWT.SHIFT`, `SWT.ALT` -- and a key character, like so:

```
menuItem.setAccelerator(SWT.CONTROL | 'C');
```

Events

`MenuItems` throw two kinds of events:

* An `ArmEvent` (use `addArmListener(ArmListener)` to receive it) is thrown when the mouse pointer enters the menu item, but before it has been clicked.

* A `SelectionEvent` (use `addSelectionListener(SelectionListener)` to receive it) is thrown when the menu item is selected.

`Menus` throw a `MenuEvent` (use `addMenuListener(MenuListener)` to receive it) when the menu is about to be shown or to be hidden.

The following code listing shows SWT menus in action:

```
//--- Creation of a menu bar
Menu menuBar = new Menu(shell, SWT.BAR);

// Create a sub menu "File" with 2 items "Open" and "Save"
MenuItem fileMenuItem = new MenuItem(menuBar, SWT.CASCADE);
fileMenuItem.setText("&File");
Menu fileMenu = new Menu(menuBar);
MenuItem openMenuItem = new MenuItem(fileMenu, SWT.PUSH);
openMenuItem.setText("&Open...");
openMenuItem.setImage(openImage);
MenuItem saveMenuItem = new MenuItem(fileMenu, SWT.PUSH);
saveMenuItem.setText("&Save");
```

```
saveMenuItem.setImage(saveImage);

// Create a sub menu "Edit" with 1 item "Copy"
MenuItem editMenuItem = new MenuItem(menuBar, SWT.CASCADE);
editMenuItem.setText("&Edit");
Menu editMenu = new Menu(menuBar);
MenuItem copyMenuItem = new MenuItem(editMenu, SWT.PUSH);
copyMenuItem.setText("&Copy");
copyMenuItem.setImage(copyImage);

shell.setMenuBar(menu);

//--- Create a pop-up menu in a control
Menu popupMenu = new Menu(control);

MenuItem item = new MenuItem(popupMenu, SWT.PUSH); // add an item "item1"
item.setText("item1");
new MenuItem(menu, SWT.SEPARATOR); // add a separator
item = new MenuItem(menu, SWT.PUSH); // add an item "item2"
item.setText("item2");

// create a cascading menu "sub-menu" containing 1 item "sub-item"
Menu subMenu = new Menu(popupMenu);
MenuItem subItem=new MenuItem(subMenu, SWT.PUSH);
subItem.setText("sub-item");

item = new MenuItem(popupMenu, SWT.CASCADE);
item.setText("sub-menu");
item.setMenu(subMenu);

//Displays the popup menu on a right-click on the control
control.addMouseListener(new MouseAdapter() {
  public void mouseDown(MouseEvent e) {
    if (e.button==3) {
      popupMenu.setLocation(control.toDisplay(e.x, e.y));
      popupMenu.setVisible(true);
    }
  }
});
```

Migrate existing Swing code

The migration of Swing menus to SWT doesn't present any particular challenge, because both toolkits have the same functionality in this area. The sample code provided with this tutorial contains several wrapper classes that make the migration easier:

- `SWTMenu`
- `SWTPopupMenu`
- `SWTMenuItem`
- `SWTRadioButtonMenuItem`
- `SWTCheckBoxMenuItem`

These wrapper classes use the API and event mapping introduced in Migrate your Swing code to SWT with minimal change on page 14 .

Let's look at a migration example. Consider the following Swing code:

```
Action myAction = ...;
```

```
JPopupMenu popupMenu = new JPopupMenu();
popupMenu.add(new JMenuItem(myAction));
popupMenu.addSeparator();
popupMenu.add(new JRadioButtonMenuItem("RadioButton"));

popupMenu.show(component, event.x, event.y);
```

Here's what the code looks like after migration to SWT:

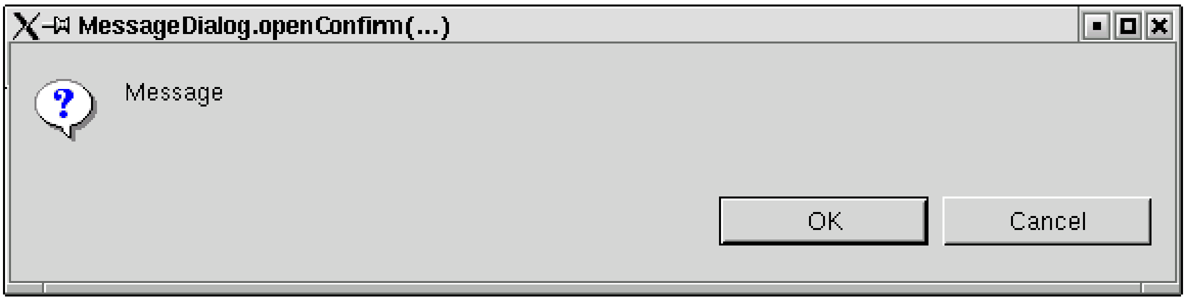
```
Action myAction = ...;
SWTPopupMenu popupMenu = new SWTPopupMenu(component);
popupMenu.add(new SWTMenuItem(popupMenu, myAction));
popupMenu.addSeparator();
popupMenu.add(new SWTRadioButtonMenuItem(popupMenu, "RadioButton"));

popupMenu.show(component, event.x, event.y);
```

JOptionPane

With the class `MessageDialog`, JFace provides a framework that is similar to Swing's `JOptionPane`. Both serve as the basis of all kinds of confirmation, error, and input dialogs.

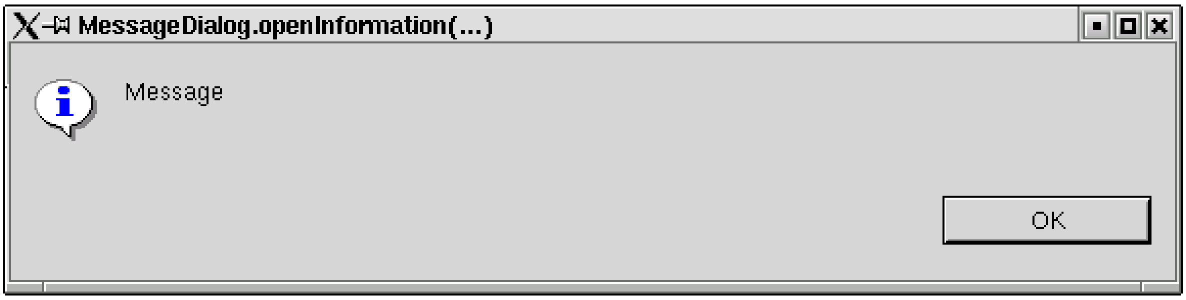
As with `JOptionPane`, you can subclass `MessageDialog` and create your own customized dialogs, but most of the time you will just use one of the static methods it provides:

- `MessageDialog.openConfirm(Shell, String, String)`: Open an OK/Cancel confirmation dialog.

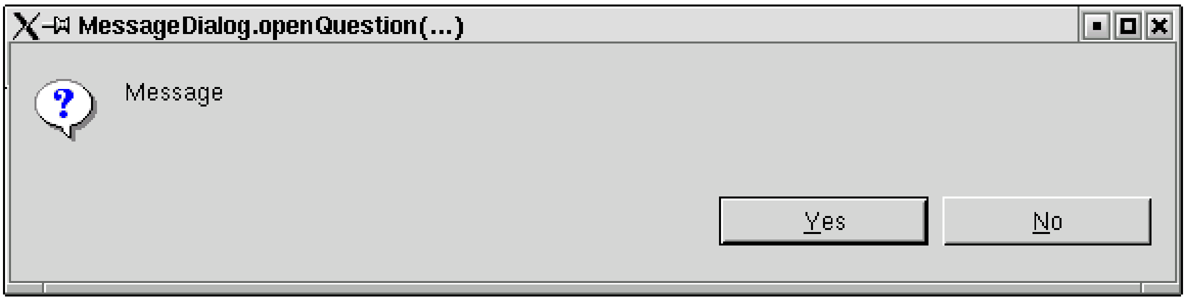


- `MessageDialog.openError(Shell, String, String)`: Open a dialog displaying an error message.

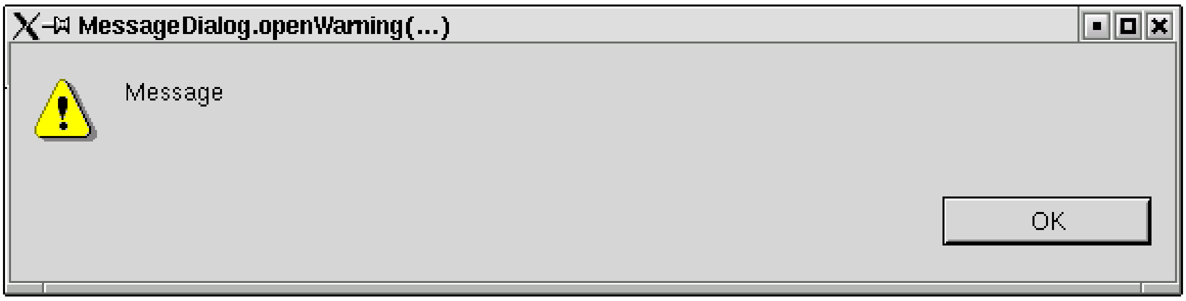
- `MessageDialog.openInformation(Shell, String, String)`: Open a dialog displaying a simple message.



- `MessageDialog.openQuestion(Shell, String, String)`: Open a Yes/No dialog asking the user to answer a question.

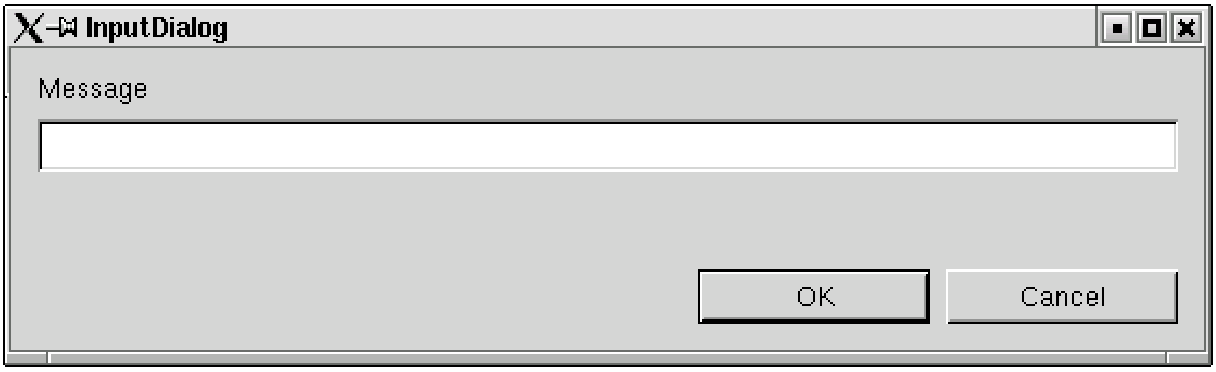


- `MessageDialog.openWarning(Shell, String, String)`: Open a dialog displaying a warning.



You can create an input dialog asking the user to enter a string by using `InputDialog`:

```
InputDialog dialog = new InputDialog(shell, "title", "message", null, null);
dialog.open();
String value = dialog.getValue();
```



Note that you can also use the JFace dialog `ErrorDialog` to display error messages. This dialog offers some additional functionality for displaying a stack trace or a detailed message.

Migrate existing Swing code

The migration of `JOptionPane` dialogs should not be problematic if you use static methods to open the standard dialogs. If you created a customized `JOptionPane`, you will have to create your own dialog by subclassing one of JFace's standard dialogs.

For an easier migration, you can use the helper class `SWTOptionPane`, provided in the sample code accompanying this tutorial. This class provides static methods that are similar to the static methods used in `JOptionPane` to open a standard dialog.

Let's look at a migration example. Consider the following Swing code:

```
if (JOptionPane.showConfirmDialog(parent, "Do you confirm?")
    == JOptionPane.YES_OPTION) ....
```

You could migrate it to SWT as follows:

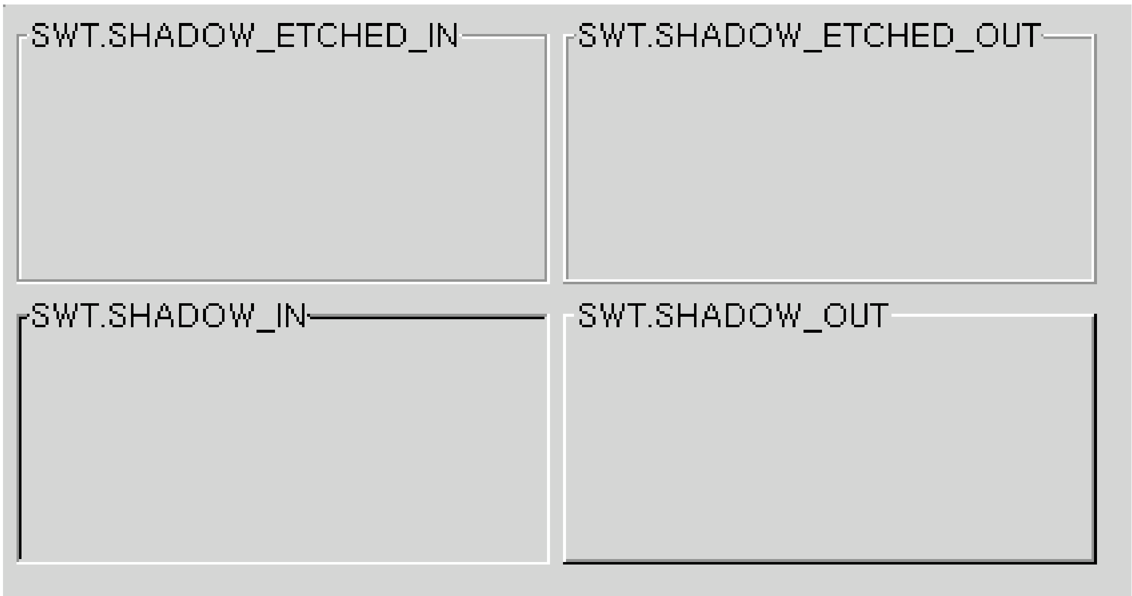
```
if (SWTOptionPane.showConfirmDialog(parent, "Do you confirm?")
    == SWTOptionPane.YES_OPTION) ....
```

JPanel

SWT provides two panel components that can be used to group together some controls of a UI, each with its own layout: `Composite` and `Group`.

`Composite` is comparable to `java.awt.Panel`. It is a container that can contain other controls and arrange them in a specific layout. You can't set a customized border for a `Composite` as you can for a Swing `JPanel`. Thus, a `Composite` is generally used as an invisible container to lay out controls in a specific way. Note that a `Composite` can display a basic border if it is created with the style `SWT.BORDER`. The appearance of the border depends on the underlying platform.

`Group` is a subclass of `Composite` and offers more possibilities to customize its appearance. A `Group` usually has a border around it and can have a title. Several types of borders are available. You can set one of them by using any one of several styles: `SWT.SHADOW_ETCHED_IN`, `SWT.SHADOW_ETCHED_OUT`, `SHADOW_IN`, and `SHADOW_OUT` (see the screenshot below). You can set a title for the group by using the method `setText(String)`. The title is then displayed in the border as it is in Swing's `TitledBorder`.



Use a `Composite` when you need an invisible panel to solve a specific layout problem. Use a `Group` when you need a visible panel with a border.

The following code snippet shows an example of SWT panels in action:

```
//--- Creation of an invisible panel
Composite composite = new Composite(parent, SWT.NONE);
composite.setLayout(new FlowLayout());
// add some controls
Button button = new Button(composite, SWT.PUSH);
...
//--- Creation of a titled panel with border
Group group = new Group(parent, SWT.SHADOW_ETCHED_IN);
group.setText("Group Title");
group.setLayout(new FlowLayout());
// add some controls
Button checkBox = new Button(group, SWT.CHECK);
...
```

Migrate existing Swing code

The migration of a `JPanel` is not problematic, as it is a passive component. However, you may encounter some problems if your panels make use of customized borders. In such a case, it may be easier to subclass `Composite` and create a customized control that can draw all kinds of borders.

The wrapper class `SWTPanel`, included with the sample code provided with this tutorial, makes the migration easier. It uses the API mapping introduced in Migrate your Swing code to SWT with minimal change on page 14 , so the migration work you'll have to do is limited to the following simple steps:

- Search for occurrences of the Swing type `JPanel` and replace them with the new wrapper type `SWTPanel`.

- Search for constructors where a panel is created and add the reference to the parent of the panel in the arguments list.

Let's look at a migration example. Consider the following Swing code:

```
JPanel p = new JPanel(new BorderLayout());
p.add(new JButton("button"), BorderLayout.CENTER);
parent.add(p);
```

Here's how you would migrate that code to SWT:

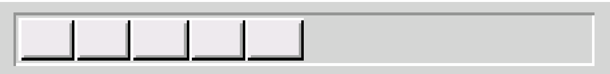
```
SWTPanel p = new SWTPanel(parent, new BorderLayout());
p.add(new JButton("button"), BorderLayout.CENTER);
parent.add(p);
```

JProgressBar

SWT and JFace provide two components that can be used to display the progress of a task: `ProgressBar` and `ProgressIndicator`.

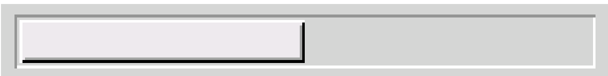
ProgressBar

`ProgressBar` is the basic progress component provided by SWT. Its API and functionality are quite similar to Swing's `JProgressBar`. The orientation of the bar must be defined in the constructor by using one of two styles: `SWT.HORIZONTAL` or `SWT.VERTICAL`. There is no way to change this orientation after the component has been created. A standard `ProgressBar` is illustrated in the figure below:



You can combine other style constants with the orientation styles:

- `SWT.SMOOTH` is a style that modifies the look of the progress indicator. When this style is used, task progress is represented as a plain bar that can take any value, instead of a chain of blocks that only grows when there is enough progress to display an additional block. A `ProgressBar` with the `SWT.SMOOTH` constructor would look like this:

  This style may be ignored by those platforms, such as Motif, that always display a smooth progress bar.

- `SWT.INDETERMINATE` is a style you can use when the length of the task represented by the progress bar is unknown. When this style is used, the progress bar displays a continuous animation showing that the task is still running. This is the equivalent of `JProgressBar.setIndeterminate(boolean)`, which was introduced in Swing in J2SE 1.4.

Like Swing, SWT provides methods to set the minimum, maximum, and current values of a progress bar. All of these values must be integers. Note that the equivalent of `JProgressBar.setValue(int)` is `ProgressBar.setSelection(int)` in SWT. Unlike Swing's bar, SWT's progress bar can't display customized text.

ProgressIndicator

`ProgressIndicator` is provided by JFace (in the package `org.eclipse.jface.dialogs`). It does the same thing as `ProgressBar` -- in fact, it is a `Composite` containing a `ProgressBar` -- but it uses a simplified API.

A `ProgressIndicator` only needs to be initialized with the single method `beginTask(int)`, which takes as its parameter the maximum progression value. When the task being monitored has progressed, you invoke `worked(double)` with the amount of new progress as a parameter. Be careful: This value does not represent the current progress value, like its equivalent in `ProgressBar.setSelection(int)`, but rather represents the relative amount of progress since the last invocation of `worked(double)`. To move the progress indicator to the end, invoke `sendRemainingWork()`. The method `done()` reinitializes the progress bar, indicating that no task is running.

Note that, unlike a `ProgressBar`, a `ProgressIndicator` can switch its state from a set amount of progress to an undetermined amount of progress after the component has been created. Invoke `beginAnimatedTask()` to switch to an undetermined progression, and `beginTask(int)` to switch back to a set amount of progress.

Because the constructor of `ProgressIndicator` doesn't accept any style as a parameter, the widget's orientation must be horizontal, and it is not possible to use the smooth progression mode as you can with `ProgressBar`.

Note that if you need a dialog that can both display the progress of a long task and give the user the option to cancel that task, you may want to use the JFace dialog `org.eclipse.jface.dialogs.ProgressMonitorDialog`.

The following code snippet shows SWT progress bars and indicators in action:

```
//--- Creation of an horizontal progress bar
ProgressBar progressBar = new ProgressBar(parent, SWT.HORIZONTAL);
progressBar.setMaximum(500); // set the maximum value to 500
progressBar.setSelection(100); // set the current value to 100

//--- Creation of a smooth progress bar with an automatic animation
new ProgressBar(parent, SWT.HORIZONTAL | SWT.SMOOTH | SWT.UNDETERMINATE);

//--- Creation of a progress indicator
ProgressIndicator progressIndicator = new ProgressIndicator(parent);
progressIndicator.beginTask(100);
progressIndicator.worked(10.0); // moves the bar of 10 units of work forward
progressIndicator.beginAnimatedTask(); // switch in automatic animation modus.
```

Migrate existing Swing code

The migration of a `JProgressBar` to a `ProgressBar` is not problematic, because a progress bar is a passive component, and the Swing and SWT components provide nearly the same functionality.

The wrapper class `SWTProgressBar`, included in the sample code provided with this tutorial, makes the migration easier. It uses the API and event mapping introduced in Migrate your Swing code to SWT with minimal change on page 14 , so the migration work you'll have to do is limited to a few simple steps:

- Search for occurrences of the Swing type `JProgressBar` and replace them with the new wrapper type `SWTProgressBar`.
- Search for constructors where a progress bar is created and add the reference to the parent of the label in the arguments list.

Let's look at a migration example. Consider the following Swing code:

```
JProgressBar progressBar = new JProgressBar(JProgressBar.HORIZONTAL, 0, 100);
parent.add(progressBar);
progressBar.setValue(50);
```

Here's what that code would look like migrated to SWT:

```
SWTProgressBar progressBar = new SWTProgressBar(parent, SWTProgressBar.HORIZONTAL, 0
parent.add(progressBar);
progressBar.setValue(50);
```

JScrollPane and JViewport

The SWT equivalent of Swing's `JScrollPane` or `JViewport` is the widget `org.eclipse.swt.custom.ScrolledComposite`, illustrated below:

Note that, unlike Swing, SWT doesn't require you to explicitly put lists, trees, tables, or text components in a scrollpane to make them scrollable. These components are made scrollable by creating them with the style constants `SWT.H_SCROLL` and `SWT.V_SCROLL`, like so:

```
//--- Creation of a multiline text area with both scrollbars
```

```
Text text = new Text(parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);

//--- Creation of a list with a vertical scrollbar only
List list = new List(parent, SWT.SINGLE | SWT.V_SCROLL);
```

Thus, you should only use a `ScrolledComposite` if you need to make one of the following scrollable:

- A canvas
- One of your customized widgets (based on a canvas)
- A composite containing other widgets

The use of a `ScrolledComposite` is similar to the use of the combination `JScrollPane/JViewport` in Swing. The viewed component can be set with the method `setContent(Control)`, and, as with `JViewport`, you can programmatically set the position of the visible area. The methods to do that are `setOrigin(Point)` and `getOrigin()`, the latter returning the current position of the viewer.

As you may have noticed, SWT doesn't make the distinction Swing makes between a `JScrollPane` and a `JViewport`. If you need the equivalent of a `JViewport` -- a "viewing hole" that can be moved programmatically to display a rectangular area of a larger component -- you just have to create a `ScrolledComposite` without scrollbars by using the style `SWT.NONE`.

Note that there is some functionality available in Swing but not in SWT for these components. There is no way to define row and column headers -- that is, vertical and horizontal components placed on the left-hand side or the top of the scrolling area. If you need this functionality, you will have to implement your own widget from a `Composite`.

Scrollbar policy

In SWT, you control the visibility of the scrollbars in a slightly different way than you do in Swing; using the styles `SWT.H_SCROLL` and/or `SWT.V_SCROLL` you can define at construction time whether scrollbars are to be used for horizontal scrolling, vertical scrolling, scrolling in both directions, or no scrolling at all. Then, the method `setAlwaysShowScrollBars(boolean)` allows you to define whether the enabled scrollbars are always shown or shown only when they are needed.

Size of the viewed component

In Swing, you set the size of a viewed component by invoking `setPreferredSize(Dimension)` on it. If the scrollpane is smaller than the preferred size, the view becomes its preferred size. If the scrollpane is larger, the view becomes the size of the scrollpane. SWT's `ScrolledComposite` provides two ways of defining the size of its content:

- If you simply invoke `setSize(int, int)` on the content, it will have a constant size. If the `ScrolledComposite` is smaller than its content, it will be scrollable. If it is larger, the scrollbars are disabled but the size of the content remains unchanged.
- If you invoke the methods `setExpandHorizontal(true)`, `setExpandVertical(true)`, and `setMinSize(int, int)` on the `ScrolledComposite`, the behavior will be similar to what it would be in Swing. If the

`ScrolledComposite` is smaller than the size defined by `setMinSize(int, int)`, the content will have that size and the scrollbars will be enabled. If the `ScrolledComposite` is larger, the content will be enlarged to its size. `setExpandHorizontal(true)`, `setExpandVertical(true)`, and `setMinSize(int, int)` must be invoked after the content is set with `setContent(Control)`.

The following code snippet shows an SWT `ScrolledComposite` in action:

```
//--- Creation of a ScrolledComposite displaying a child composite
ScrolledComposite scrolledComposite = new ScrolledComposite(
                            parent, SWT.H_SCROLL | SWT.V_SCROLL | SWT.BORDER);
Composite childComposite = new Composite(scrolledComposite, SWT.NONE);
childComposite.setSize(1000, 1000);
scrolledComposite.setContent(childComposite);
```

Migrate existing Swing code

Before migrating a `JScrollPane`, you should ask yourself if you really need a `ScrolledComposite`. Remember that widgets that are usually scrollable, such as text, lists, tables, or trees, only need the styles `SWT.H_SCROLL` and `SWT.V_SCROLL` to be scrollable. If you do need to port a `JScrollPane` or `JViewport`, you can use the following wrapper classes:

- `SWTScrollPane`
- `SWTViewport`

These classes use the API and event mapping introduced in Migrate your Swing code to SWT with minimal change on page 14 , so the migration work you'll have to do is limited to the following simple steps:

- Search for occurrences of the Swing types `JScrollPane` and `JViewport` and replace them with the new wrapper types `SWTScrollPane` and `SWTViewport`, respectively.
- Search for constructors where a scrollpane or a viewport is created and add the reference to the parent of the label in the arguments list. Note that if your Swing code uses the `JScrollPane(Component)` constructor with the component to view provided as the argument, in the ported code you'll have to explicitly set the component to view with `SWTScrollPane.setView(SWTComponent)`. This is because in SWT you can't create the component to view before its parent -- the scrollpane, in this case -- is constructed.

Let's look at a migration example. Consider the following Swing code:

```
JScrollPane scrollPane = new JScrollPane(componentToView);
scrollPane.getViewport().setViewPosition(new Point(100,100));
parent.add(scrollPane);
```

Here's what this code would look like ported to SWT:

```
SWTScrollPane scrollPane = new SWTScrollPane(parent);
scrollPane.setView(componentToView);
scrollPane.getViewport().setViewPosition(new Point(100,100));
parent.add(scrollPane);
```

JSeparator

Separators -- visual dividers used to separate widgets in a container, or a logical group of menu items in a menu -- are not represented in SWT by a unique class like Swing's `JSeparator`. *Menu separators* are created by instantiating a `MenuItem` having the style `SWT.SEPARATOR`. *Widget separators* in a panel or a toolbar are created by instantiating a `Label` having the style `SWT.SEPARATOR` combined with one of two style constants, `SWT.HORIZONTAL` or `SWT.VERTICAL`, that define the orientation of the separator. The orientation must be defined at construction time.

Here's what a separator would look like:

The following code snippet illustrates both kinds of separators in action:

```
//--- Creation of a menu separator
new MenuItem(parentMenu, SWT.SEPARATOR);

//--- Creation of a vertical separator in a parent composite
new Label(parent, SWT.SEPARATOR | SWT.VERTICAL);
```

Migrate existing Swing code

Because Swing's separators can't be customized, their migration to SWT is not problematic.

The wrapper class `SWTSeparator`, included with the sample code provided with this tutorial, makes the migration easier. It uses the API and event mapping introduced in Migrate your Swing code to SWT with minimal change on page 14 , so the migration work you'll have to do is limited to a few simple steps:

- Search for occurrences of the Swing type `JSeparator` and replace them with the new wrapper type `SWTSeparator`.
- Search for constructors where a separator is created and add the reference to the parent of the separator in the arguments list.

Let's look at a migration example. Consider the following Swing code:

```
//--- Add a vertical separator in a panel
panel.add(new JSeparator(SwingConstants.VERTICAL));
```

You could migrate this code to SWT as follows:

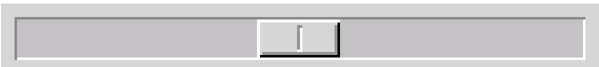
```
//--- Add a vertical separator in a panel
panel.add(new SWTSeparator(panel, SwingConstants.VERTICAL));
```

JSlider

Once again, SWT provides here two alternatives to replace a single Swing component. Both SWT components -- `Scale` and `Slider` -- have functionality that is quite similar to that of Swing's `JSlider`. In fact, when you read SWT's API documentation, you don't really get a sense of the difference between these two components: Both are used to select a numeric value within a bound of values, and both have nearly the same API and functionality. The only difference between `Scale` and `Slider` is in their look and feel:

- A `Slider` has at both ends arrows to increment or decrement the selected value, like a scroll bar does. The cursor, whose position represents the current value, has a variable width that can be programmatically set by using the method `setThumb(int)` -- this is the equivalent of Swing's `JSlider.setExtent(int)` method, with the difference being that SWT's `setThumb(int)` requires a positive, non-zero argument, where Swing's `setExtent(int)` accepts a zero argument.

- A `Scale` is simpler. It doesn't contain the arrows and its cursor has an invariable size. The rest of its API is exactly the same as `Slider`'s.



As you can with Swing's `JSlider`, you can set the minimum and maximum values for these components by invoking `setMininum(int)` and `setMaximum(int)`. The method to set the current value is in SWT named `setSelection(int)` and not `setValue(int)`.

SWT doesn't offer the flexibility to customize the look of the slider that Swing offers. There is no way to define whether gradations are displayed or not, or to display customized labels.

Events

`Sliders` and `Scales` only throw one type of event: A `SelectionEvent` is thrown each time the value of the slider or scale changes.

The following code snippet shows SWT sliders and scales in action:

```
// Create a slider
Slider slider = new Slider(parent, SWT.HORIZONTAL);
//set mininum, maximum, thumb, and increments value in a single line
slider.setValues(50, 0, 100, 30, 1, 10);
...
slider.setSelection(60); // change the current value

//Create a scale
Scale scale = new Scale(parent, SWT.HORIZONTAL);
slider.setMaximum(200);
slider.setSelection(50);
```

Migrate existing Swing code

The migration of existing Swing code is not problematic as long as you don't need a slider with customized labels.

The wrapper class `SWTSlider`, included with the sample code provided with this tutorial, makes the migration easier. It uses the API and event mapping introduced Migrate your Swing code to SWT with minimal change on page 14 , so the migration work you'll have to do is limited to a few simple steps:

- Search for occurrences of the Swing type `JSlider` and replace them with the new wrapper type `SWTSlider`.
- Search for constructors where a slider is created and add the reference to the parent of the slider in the arguments list.

Let's look at a migration example. Consider the following Swing code:

```
JSlider slider = new JSlider();
slider.addChangeListener(new ChangeListener() {
  public void stateChanged(ChangeEvent e) {
      // do action
  }
});
parent.add(slider);
```

You can convert this code to SWT as follows:

```
SWTSlider slider = new SWTSlider(parent);
slider.addChangeListener(new ChangeListener() {
  public void stateChanged(ChangeEvent e) {
      // do action
  }
});
parent.add(slider);
```

JSplitPane

The equivalent of Swing's `JSplitPane` in SWT is the customized widget `org.eclipse.swt.custom.SashForm`, illustrated in the figure below:

Where Swing's `JSplitPane` only allows you to divide a panel into two parts with a divider between them, a `SashForm` is a composite that can be split into as many parts as its number of children. A draggable divider is added between children. As in Swing, you can control the orientation of the split (horizontal or vertical) by passing one of two style constants, `SWT.HORIZONTAL` or `SWT.VERTICAL`, in the constructor, or by invoking the method `setOrientation(int)`.

Because a `SashForm` can be divided into more than two parts, there are no `setRightComponent(Component)` or `setLeftComponent(Component)` methods as there are for `JSplitPane`. The order of creation of the child components defines their position on the screen, as with a `FlowLayout`:

- If the `SashForm` is horizontal, the children will be laid out from left to right.
- If the `SashForm` is vertical, the children will be laid out from top to bottom.

The position of the divider can be set by invoking the method `setWeights(int[])`. This method expects an array containing as many integers as the number of children in the `SashForm`. Each of these values defines the relative width or height (depending on the orientation) of the children. Note that this method must be invoked after all the children of the `SashForm` are created.

SWT's `SashForm` has a feature that is not available in Swing's `JSplitPane`. You can programmatically maximize one child of the `SashForm` by invoking the method `setMaximizedControl(Control)`. This operation can by reversed by invoking the same method with `null` as its parameter.

The following code snippet shows an example of an SWT `SashForm` in action:

```
//--- Creation of a horizontal SashForm containing two children composites
SashForm sashForm = new SashForm(parent, SWT.HORIZONTAL);
Composite leftComposite = new Composite(sashForm, SWT.NONE);
Composite rightComposite = new Composite(sashForm, SWT.NONE);
//--- Set the position of the divider to 1/3
sashForm.setWeights(new int[] {1,2});
```

Migrate existing Swing code

The migration from Swing to SWT is not really problematic here, because both toolkits offer similar functionality. TThe wrapper class SWTTabbedPane, included in the sample code provided with this tutorial, makes the migration easier. However, because `SashForm` uses its children's creation order to decide where those children are to be placed, you may have to change the order of some lines of code. Note that it is not possible in SWT to pass the children as arguments in the constructor as you can in `JSplitPane`, because the child components need an already constructed parent container in order to be constructed themselves.

To migrate Swing code with the help of `SWTSplitPane`, you should proceed as follows:

- Replace all the references to the class `JSplitPane` with the class `SWTSplitPane`.
- Search for any invocation of a constructor of `JSplitPane` and replace it with the constructor of `SWTSplitPane`, passing the parent of the split pane as the first parameter and the orientation as the second parameter.
- Search for the code creating the two children of the split pane, migrate it to SWT, and move it or reorder it so that the children are created just after the `SashForm`, the left- or topmost component being the first child to be created.
- Remove any invocation of `setLeftComponent()`, `setRightComponent()`, `setTopComponent()`, or `setBottomComponent()`.

Let's look at a migration example. Consider the following Swing code:

```
JPanel leftPanel = new JPanel();
JPanel rightPanel = new JPanel();
JSplitPane splitPane = new JSplitPane(
    JSplitPane.HORIZONTAL_SPLIT, leftPanel, rightPanel);
splitPane.setDividerLocation(0.3);
```

Here's what this code would look like after being migrated to SWT:

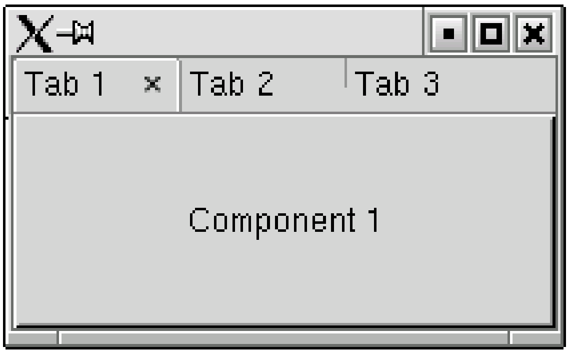
```
SWTSplitPane splitPane = new SWTSplitPane(parent, SWTSplitPane.HORIZONTAL_SPLIT);
SWTPanel leftPanel = new SWTPanel(splitPane);
SWTPanel rightPanel = new SWTPanel(splitPane);
splitPane.setDividerLocation(0.3);
```

JTabbedPane

SWT provides two components that can be used to replace a Swing `JTabbedPane`.

`TabFolder` uses a native widget from the underlying platform. Its functionality is quite similar to Swing's `JTabbedPane`, with one limitation: you can't modify the placement of the tabs. The look and feel of the widget is not customizable. This is what a `TabFolder` would look like:

`CTabFolder`, from the package `org.eclipse.swt.custom`, is an emulated widget that provides more functionality and possibilities for customization than `TabFolder`. The look and feel of the widget is the same on all platforms. By using in the constructor one of two style constants, `SWT.TOP` or `SWT.BOTTOM`, you can control the position at which the tabs are displayed. (Note that SWT doesn't offer the possibility of displaying the tabs on the sides of the component as Swing does.) Additionally, `CTabFolder` lets you set the height of the tabs and allows you to place a visual separator between two tabs by using the method `setInsertMark()`. There is one more feature that is specific to `CTabFolder` and is available neither in SWT's `TabFolder` nor in Swing's `JTabbedPane`: By adding a `CTabFolderListener` on the component, the tabs become *closeable.* Once a listener is registered, each tab has a button with a cross icon that automatically make the tab disappear when the user clicks on it. The listeners are then notified that a tab has been closed. This is the same behavior used in Eclipse when you close an editor by clicking on the close button in its tab. This is what a `CTabFolder` would look like:



Whether you use a `TabFolder` or a `CTabFolder` to replace a `JTabbedPane` depends on the level of customization required by your application. If the standard look and feel of the native `TabFolder` is good enough, it is better to use it to get the best performance. If the tabs have to be placed on the bottom of the component, of if you need a tab folder without border or more compact tabs whose height can be precisely defined, using a `CTabFolder` is the only choice you have. Both components have nearly the same API, so you can easily try out both in you application to find out which one better fit your needs.

Adding and removing items

SWT's method for adding and removing tabs or pages is different from Swing's. For each tab in the folder, you have to create a widget -- a `TabItem` or `CTabItem`, depending on whether you are using a `TabFolder` or a `CTabFolder`. A `TabItem` represents an empty tab in the folder. The constructor lets you indicate the index at which the tab should be inserted. To assign a control, a title, an icon, or a tooltip to a tab, you would invoke the methods `setControl(Control)`, `setText(String)`, `setImage(Image)`, and `setToolTipText(String)`, respectively. Successive calls of `TabFolder.setControl(Control)` let you change the content of a page without having to re-create the `TabItem`.

You can remove a tab by invoking the method `dispose()` on the `TabItem`. Once a tab has been discarded, it cannot be used anymore and has to be recreated if you need to add it again.

Note that discarding a `TabItem` doesn't dispose of the control that was associated to it with `setControl(Control)`. Thus, although a discarded `TabItem` is not reusable, its content can be reassigned to a new `TabItem`.

Events

A `SelectionEvent` is thrown each time a new tab is selected. To detect a change in the selection, register a `SelectionListener` by using the method `addSelectionListener(SelectionListener)`. The listener method that is triggered by the event is `SelectionListener.widgetSelected(SelectionEvent)`. Additionally, a `CTabFolder` throws a `CTabFolderEvent` to its `CTabFolderListener` when the user closes a tab by clicking on its close button. By setting the `doit` field of the event to `false`, you can programmatically forbid the user to close the tab.

The following code snippet illustrates SWT tabs in action:

```
//--- Creation of a native TabFolder containing 3 tabs, each tab contains a button
TabFolder tabFolder = new TabFolder(parent, SWT.NONE);

Button b1 = new Button(tabFolder, SWT.PUSH);
b1.setText("Component 1");
TabItem tabItem = new TabItem(tabFolder, SWT.NONE);
tabItem.setText("Tab 1");
tabItem.setControl(b1);

Button b2 = new Button(tabFolder, SWT.PUSH);
b2.setText("Component 2");
tabItem = new TabItem(tabFolder, SWT.NONE);
tabItem.setText("Tab 2");
tabItem.setControl(b2);

Button b3 = new Button(tabFolder, SWT.PUSH);
b3.setText("Component 3");
tabItem = new TabItem(tabFolder, SWT.NONE);
tabItem.setText("Tab 3");
tabItem.setControl(b3);

// Insert afterwards a new tab at index 1
tabItem = new TabItem(tabFolder, SWT.NONE, 1);
tabItem.setText("Inserted Tab");
tabItem.setControl(control);

//--- Creation of a CTabFolder with 2 tabs displayed on the bottom of the component
CTabFolder ctabFolder = new CTabFolder(parent, SWT.BOTTOM);

Button ba = new Button(ctabFolder, SWT.PUSH);
ba.setText("Component A");
CTabItem ctabItem = new CTabItem(ctabFolder, SWT.NONE);
ctabItem.setText("Tab A");
ctabItem.setControl(ba);
```

```
Button bb = new Button(ctabFolder, SWT.PUSH);
bb.setText("Component B");
ctabItem = new CTabItem(ctabFolder, SWT.NONE);
ctabItem.setText("Tab B");
ctabItem.setControl(bb);

// Make the tabs closeable
ctabFolder.addCTabFolderListener(new CTabFolderListener() {
  public void itemClosed(CTabFolderEvent event) {
    ...
  }
});
```

Migrate existing Swing code

The migration of existing Swing code is only problematic if in Swing you use a `JTabbedPane` whose tabs have to be placed on the side of the component. In any other case, you won't encounter any problem.

The wrapper class `SWTTabbedPane`, included in the sample code provided with this tutorial, makes the migration easier. It uses the API and event mapping introduced in Migrate your Swing code to SWT with minimal change on page 14 , so the migration work you'll have to do is limited to a few simple steps:

- Search for occurrences of the Swing type `JTabbedPane` and replace them with the new wrapper type `SWTTabbedPane`.
- Search for constructors where a tabbed pane is created and add the reference to the parent of the tabbed pane in the arguments list.

Let's look at a migration example. Consider the following Swing code:

```
JTabbedPane tabbedPane = new JTabbedPane();
tabbedPane.add("Tab 1", component1);
tabbedPane.add("Tab 2", component2);
tabbedPane.add("Tab 3", component3);
tabbedPane.setSelectedIndex(1);
parent.add(tabbedPane);
```

You can migrate this code to SWT like so:

```
SWTTabbedPane tabbedPane = new SWTTabbedPane(parent);
tabbedPane.add("Tab 1", component1);
tabbedPane.add("Tab 2", component2);
tabbedPane.add("Tab 3", component3);
tabbedPane.setSelectedIndex(1);
parent.add(tabbedPane);
```

JTable

SWT's equivalent for Swing's `JTable` is the component `Table`. It can be used in combination with JFace's `TableViewer`.

The use of a pure SWT table, without JFace's `TableViewer`, is, from the programmer's perspective, quite different from the use of Swing's `JTable`:

- The major difference between Swing's `JTable` and SWT's `Table` is that SWT doesn't make use of a data model like Swing's `TableModel`. In SWT, each table row is a widget of type `TableItem` that must be instantiated with the `Table` itself as parent. A `TableItem` can display text and an image for each column of the table. The content of the table is set in the `TableItems` themselves by invoking one of two methods, `TableItem.setImage(...)` and `TableItem.setText(...)`.

- SWT has no equivalent for Swing's `TableCellRenderer`. An SWT table can only display text and an image in each cell.

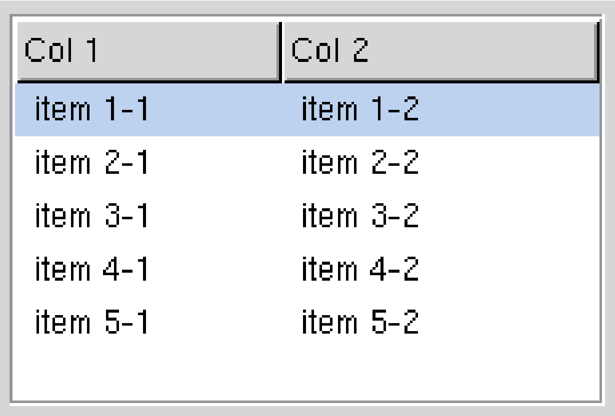
- Like the rows, the columns of a table are widgets that have to be instantiated as children of the `Table` as parent. The class for the column widgets is `TableColumn`. `TableColumns` can be instantiated with one of the three styles -- `SWT.LEFT`, `SWT.CENTER`, and `SWT.RIGHT` -- defining the alignment of the content of the table in the column. Note that some platforms, like GTK on Linux, ignore this constant. As with `TableItems`, you can set on a `TableColumn` text and an image with the methods `TableColumn.setText(String)` and `TableColumn.setImage(Image)`. The text and image of a column are displayed in the header of the table when it is visible.

- Because each table row is its own widget, an SWT table is not as scalable as Swing's `JTable`. SWT programmers are trying to solve this problem for future releases, but as of SWT 2.1 you have to keep in mind that very large tables (more than 10,000 rows) may present performance problems, mainly in the initialization time of the table.

The following code snippet shows the use of a pure SWT table, without a JFace viewer:

```
//--- Example of creation of a simple table without TableViewer
Table table = new Table(composite, SWT.BORDER | SWT.H_SCROLL | SWT.V_SCROLL | SWT.FULL_S
table.setHeaderVisible(true);

// Create 2 columns
TableColumn column1 = new TableColumn(table, SWT.LEFT);
column1.setText("Col 1");
column1.setWidth(100);
TableColumn column2 = new TableColumn(table, SWT.LEFT);
column2.setText("Col 2");
column2.setWidth(100);

// Create 5 rows
int nbColumns = table.getColumnCount();
for (int row=1 ; row<=5 ; row++) {
  TableItem tableItem = new TableItem(table, SWT.NONE);
  for (int col=0 ; col<nbColumns ; col++) {
    tableItem.setText(col, "item "+row+"-"+(col+1));
  }
}
```

And here's what such a table would look like:



JFace's TableViewer

Most of the time, however, you wouldn't create a table as shown above. Rather, you will use a JFace `TableViewer`. A `TableViewer` is a JFace viewer created on top of an SWT `Table`. It automatically creates and sets up the `TableItems` to represent a data model provided by a content provider in a text/icon form defined by a label provider. In this way, you have a mechanism that is much closer to Swing's `TableModel/TableCellRenderer` mechanism. For more information on JFace's viewers, read Data models and cell renderers vs. content providers and label providers on page 12 , or read the articles listed in the Resources on page 95 . For concrete examples showing how to use a `TableViewer`, you should focus on "Using the Eclipse GUI outside the Eclipse Workbench" by Adrian Van Emmenis, and "Building and delivering a table editor with SWT/JFace" by Laurent Gauthier.

Table items

If you use a JFace `TableViewer`, you don't have to care about the `TableItems` of the table, because those are automatically created by the viewer. However, in some cases it can be useful to work with the `TableItems` directly, even if they are automatically created.

By using the `Table`'s API, you can get the list of all the `TableItems`, or of those items that are selected. By invoking `setBackground(Color)` or `setForeground(Color)`, you can modify the colors of single rows. This is something that you can't do with the API of JFace `TableViewer` and its label provider.

Table columns and table headers

SWT has no equivalent for Swing's `TableColumnModel`. For each column, you have to create a `TableColumn` widget. You can decide whether a column is resizable or not by using the `TableColumn.setResizable(boolean)` method. There is no automatic resizing policy for the columns as there is in Swing. You have to set the width of each column by invoking `TableColumn.setWidth(int)`.

In SWT, the table header displayed on top of the table is not a separate widget like Swing's `JTableHeader`, but is a part of the `Table` itself. By default, the table header is not shown. You can make it visible by invoking `Table.setHeaderVisible(boolean)`. For each column, the table header can display a column name and an optional icon, though the icon may be ignored on some platforms. To set the name and icon to display for each column, you have to invoke the methods `setText(String)` and `setImage(Image)` on the `TableColumns`.

Management of the selection

SWT has no equivalent for Swing's `SelectionModel`. You can define whether or not multiple selection is allowed by using one of two style constants, `SWT.MULTI` or `SWT.SINGLE`, when constructing the table. You can't switch from one mode to the other after the table has been created.

SWT's `Table` doesn't support cell or column selection, as Swing's `JTable` does. Only rows can be selected. If you don't use the style constant `SWT.FULL_SELECTION`, only the first cell of the selected rows is displayed as being selected. Using `SWT.FULL_SELECTION`, you can select a complete row, as you can in Swing. If you don't want the selection to be displayed, you can use the style constant `SWT.HIDE_SELECTION`. You can set and get the selection programmatically in two different ways:

- SWT's `Table` provides simple methods to set or get the selection. These methods work with either the indices of the items composing the selection or with the `TableItems` themselves.

- JFace's `TableViewer` provides two methods, `getSelection()` and `setSelection(ISelection, boolean)`, that are inherited from `StructuredViewer`; they work on a higher abstraction level. The `ISelection` object returned or used by these methods is in fact a `StructuredSelection` that provides an iterator or an array containing the selected elements as supplied by the content providers, and is independent from their string representation or their representation order.

Cell editing

Like Swing's `JTable`, JFace's `TableViewer` allows cell editing. The concepts used by SWT/JFace here are pretty similar to those used in Swing. You can define for each column a `CellEditor` that allows you to use any kind of SWT component to edit the value of a cell. A small difference is that JFace requires that you also set a `ICellModifier` on the viewer. The cell modifier decides whether or not a cell is editable, and does the translation between the data model and the editor: it provides the value that will be edited to the cell editor, and modifies the data model once the editing is completed.

For more information about cell editing in a JFace `TableViewer`, read the article "Building and delivering a table editor with SWT/JFace" by Laurent Gauthier (see Resources on page 95 for a link).

Events

The only event thrown by a `Table` is a `SelectionEvent` that notifies the `SelectionListeners` when the selection has changed.

Migrate existing Swing code

The migration of existing Swing code for a `JTable` is not problematic for tables that use standard renderers -- icons and/or text -- and don't need single cell selection. Note that you may encounter scalability problems if your table has to display a very large number of rows.

The wrapper class `SWTTable`, included with the sample code provided with this tutorial, makes the migration easier by emulating the API of Swing, as introduced in Migrate your Swing code to SWT with minimal change on page 14 . This class is able to reuse an existing

Swing `TableModel` and `TableColumnModel`. To migrate existing code using the wrapper class, you'll need to follow these steps:

- Search for occurrences of the Swing type `JTable` and replace them with the new wrapper type `SWTTable`.
- Search for constructors where a table is created and add the reference to the parent of the table in the arguments list.
- It is very probable that the Swing tables of your application are contained in `JScrollPanes`. Modify the code so that no `JScrollPane` is created, and so the tables are directly added to their parent.
- Convert optional Swing renderers into `SWTCellRenderers`.

Let's look at a migration example. Consider the following Swing code:

```
//--- Create a data model containing 4x4 strings
Object[][] data = new Object[4][4];
for (int i=0 ; i<data.length ; i++) {
  for (int j=0 ; j<data[i].length ; j++)
    data[i][j] = (i+1)+"-"+(j+1);
}
// create the name of the columns
String[] columnNames = new String[4];
for (int i=0 ; i<columnNames.length ; i++) columnNames[i]="col"+(i+1);

// create a table to display the data
JTable table = new JTable(data, columnNames);

// add a selection listener on the table
table.getSelectionModel().addListSelectionListener(new ListSelectionListener() {
  public void valueChanged(ListSelectionEvent e) {
    // do action
  }
});

// create a custom renderer for the 1st column of the table
TableCellRenderer customRenderer = new DefaultTableCellRenderer() {
  public Component getTableCellRendererComponent(
    JTable table, Object value, boolean isSelected,
    boolean hasFocus, int row, int column) {
      super.getTableCellRendererComponent(table, value, isSelected, hasFocus, row, colu
      setText("custom:"+value.toString());
      return this;
  }
};
table.getColumnModel().getColumn(0).setCellRenderer(customRenderer);

parent.add(new JScrollPane(table));
```

Here's what this code would look like after migration to SWT:

```
//--- Create a data model containing 4x4 strings
Object[][] data = new Object[4][4];
for (int i=0 ; i<data.length ; i++) {
  for (int j=0 ; j<data[i].length ; j++)
```

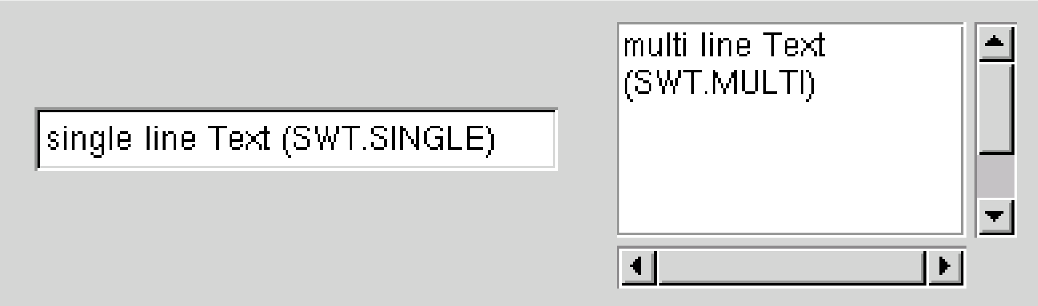
```
    data[i][j] = (i+1)+"-"+(j+1);
}
// create the name of the columns
String[] columnNames = new String[4];
for (int i=0 ; i<columnNames.length ; i++) columnNames[i]="col"+(i+1);

// create a table to display the data
SWTTable table = new SWTTable(parent, data, columnNames);

// add a selection listener on the table
table.getSelectionModel().addListSelectionListener(new ListSelectionListener() {
  public void valueChanged(ListSelectionEvent e) {
    // do action
  }
});

// create a custom renderer for the 1st column of the table
SWTCellRenderer customRenderer = new SWTCellRenderer() {
  public String getCellText(Object value, int row, int column) {
    return "custom:"+value.toString();
  }
};

table.getColumnModel().getColumn(0).setCellRenderer(customRenderer);

parent.add(table);
```

JTextField, JTextArea, and JPasswordField

For even its simplest text components, such as `JTextField` or `JTextArea`, Swing uses a pretty complicated API and class hierarchy. For normal text fields, SWT uses a much simpler design. All text fields, whether they are single-line fields, multiple-line areas, or password fields, are created by using the same component, `Text`, using different styles:

- Single-line text fields, like `JTextField`, are created by using the style `SWT.SINGLE`.
- Password fields, like `JPasswordField`, are in SWT normal text fields (with a style of `SWT.SINGLE`) on which the method `setEchoChar(char)` is invoked.
- Multiline text areas, like `JTextArea`, are created by using the style `SWT.MULTI`. You can combine this style with `SWT.WRAP` to build a text field whose lines are wrapped when they exceed the width of the component. Note that you don't need to put a text component in a scrollpane to make it scrollable; simply add the styles `V_SCROLL` and/or `H_SCROLL` if you want scrollbars to be added to the text component. Unlike a Swing `JTextArea`, an SWT text component is always scrollable with the keyboard, whether you use the styles `V_SCROLL` and `H_SCROLL` or not. These styles only define whether or not scrollbars appear.

All these elements are illustrated in the figure below.



Borders

Text components are by default created without any border around them. However, you may often want to use the style `SWT.BORDER` to create a text component surrounded by the standard border that the platform uses to draw text fields.

Selection and caret position

SWT's `Text` has an API for handling text selection that is quite similar to the API for Swing's `JTextComponent`. You can programmatically set the selection by using `setSelection(int, int)`, passing the start and end indices of the selection as arguments. Note that if you set the selection to a part of the text that is currently outside the visible area, the text component won't scroll automatically to the selected text. To do that, you have to additionally invoke the method `showSelection()`.

As in Swing, the currently selected text can be retrieved by invoking a method named `getSelectedText()`. However, the start and end indices of the selection can be retrieved by invoking a single method named `getSelection()`; this is not possible in Swing. This method returns a `Point` that contains in its x field the start index of the selection, and in its y field its end index. The unusual return type of this method is puzzling and leads some to think that it returns some screen coordinates. In fact, SWT simply misuses the `Point` object as a container object for two integer values.

Events

Like Swing, SWT gives you the capability to register listeners to notify you when the text of a text component is modified. The API for detecting such changes in a Swing text component is pretty complicated and not very intuitive -- you have to get the `Document` of the text object and register a `DocumentListener` on it. SWT offers a much simpler and more powerful way to detect such changes. SWT's `Text` throws a `VerifyEvent` to its `VerifyListeners` when its content is about to be changed. This event is thrown directly after the user presses the key provoking the change, but before the text is updated in the component. Thus, you can analyze the change that is going to take place and potentially modify or cancel that change before it occurs. That's why the event is called `VerifyEvent`: because it lets you verify whether or not the change should take place. `VerifyEvent` has four fields that you can use to analyse the change and eventually cancel it:

- `start` is a read-only field, which means that any changes you make in its value will be ignored. It indicates the index at which the text insertion or deletion will take place.
- `end` is also a read-only field. It indicates the end index of the modification. If its value is the same as `start`, text will be inserted. If its value is greater that `start`, text will be deleted.

- `text` contains the text that is going to be inserted or deleted. If text is going to be inserted, you can modify the value of this field to change the text to insert. If text is going to be deleted, changes you may make in this field will be ignored.
- `doit` is a field that you can set to `false` to cancel the event. In such a case, the change will be ignored and the text in the component remain unchanged.

`Text` next throws a `ModifyEvent` to its `ModifyListeners` after the text is in the component, assuming that the `VerifyEvent` was not canceled programmatically by setting its `doit` field to `false`.

Additionally, single-line text components throw a `SelectionEvent` event when the user presses **Enter**, just as Swing's `JTextField` throws an `ActionEvent` in the same situation. Note that the method from `SelectionListener` that is invoked is here is `widgetDefaultSelected(SelectionEvent)` and not `widgetSelected(SelectionEvent)`.

The following code snippet shows an example of SWT text fields in action:

```
//--- Create a single line text field
Text textField = new Text(parent, SWT.SINGLE | SWT.BORDER);
//--- Create a scrollable multiple line text area
Text textArea = new Text(parent, SWT.MULTI | SWT.BORDER
                                 | SWT.H_SCROLL | SWT.V_SCROLL);
```

Migrate existing Swing code

The migration of a `JTextField`, a `JTextArea`, or a `JPasswordField` to SWT should not be problematic, because the functionality is basically the same in both toolkits. However, SWT's API is much simpler than Swing's, so that minimal code change may be necessary. The deepest changes you will have to make will be in event handling if you use listeners on the document.

The following wrapper classes can facilitate the migration by emulating the Swing API under SWT:

- `SWTTextComponent`
- `SWTTextField`
- `SWTTextArea`
- `SWTPasswordField`

The Swing `DocumentChangeEvent` for the insertion and the deletion of text is also emulated. However, because SWT has no equivalent for Swing's `Document`, the methods `addDocumentListener(DocumentListener)` and `removeDocumentListener(DocumentListener)` are implemented in `SWTTextComponent` itself. `SWTTextComponent` implements a method called `getDocument()`, which returns the `SWTTextComponent` itself; thus, existing code to register listeners (`textField.getDocument().addDocumentListener(listener)`) need not be modified.

Let's look at a code migration example. Consider the following Swing code:

```
//--- Create a single line text field
JTextField textField = new JTextField("initial text");
textField.getDocument().addDocumentListener(aListener);
parent.add(textField);

//--- Create a text area
JTextArea textArea = new JTextArea("initial text");
textArea.getDocument().addDocumentListener(aListener);
parent.add(textArea);
```

Here's what that code would look like after being migrated to SWT:

```
//--- Create a single line text field
SWTTextField textField = new SWTTextField(parent, "initial text");
textField.addDocumentListener(aListener);
parent.add(textField);

//--- Create a text area
SWTTextArea textArea = new SWTTextArea(parent, "initial text");
textArea.addDocumentListener(aListener);
parent.add(textArea);
```

JToolBar

SWT provides two components that can be used to build a toolbar: `ToolBar` and `CoolBar`. Unlike other SWT controls, these two components are not alternatives to one another, but are designed to be used together

`ToolBar` is the basic toolbar component that lays out tool items -- usually buttons displaying an icon -- and optional separators. Its functionality is quite similar to Swing's `JToolBar`, except that SWT's `ToolBar` can't be made floatable like its Swing counterpart. The orientation of the toolbar -- horizontal or vertical -- can be defined by using one of two styles, `SWT.HORIZONTAL` and `SWT.VERTICAL`, in the constructor. Other styles allow you to modify the look of the bar:

- `SWT.BORDER` adds a border around the toolbar.
- `SWT.FLAT` makes the items flat. If you don't use this style, the items are represented as normal push buttons.
- `SWT.WRAP` wraps the items in several rows if there is not enough space to display them all. Note that this style is ignored by some platforms, such as GTK on Linux.

You can add items to the toolbar by creating `ToolItems`. The API of `ToolItem` is quite similar to the API of `Button`. By using in an item's constructor one of several styles -- `SWT.PUSH`, `SWT.CHECK`, `SWT.RADIO`, or `SWT.DROP_DOWN` -- you will create a normal push item, a check box, a radio button, or an item displaying a drop-down menu, respectively. By using the style `SWT.SEPARATOR`, you will create a visual separator between two items. By using the methods `ToolItem.setImage(image)`, `ToolItem.setHotImage(image)`, and `ToolItem.setDisableImage(image)`, you can define different icons to be displayed when the tool item is in its normal state, when the mouse pointer is on it, and when it is disabled, respectively. You can also add any SWT control to a toolbar -- it is for example quite usual to include a combo box in a toolbar to allow the user to change a font size or a zoom factor -- by creating a `ToolItem` with the style `SWT.SEPARATOR`, and then invoking the method `setControl(Control)` on it. You will then have to set its width by invoking the method `setWidth(int)` on the `ToolItem`. Note that the height of a `ToolBar` and all its `ToolItems` is defined by the platform and can't be changed.

A `CoolBar` is a multiline toolbar whose items can be moved and reordered by the user. A `CoolBar` usually contains several `ToolBars`. Each `ToolBar` is an atomic group of items that can be reordered by the user within the `CoolBar`.

You add items to the coolbar by creating `CoolItems` and setting SWT controls on them with the method `CoolItem.setControl(Control)`. Unlike `ToolItem`, `CoolItem` lets you set its width and height. The layout of the `CoolBar` -- the order of its items, their sizes, and the indices at which the row is wrapped -- can be set programmatically by using `setItemLayout(int[], int[], Point[])`. Keep in mind that each item in a `CoolBar` can be reordered by the user. Thus, you should avoid adding too many single components to the `CoolBar` and rather use it to contain several `ToolBars`, each of them defining a logical grouping of items.

The following code snippet illustrates SWT `CoolBars` and `ToolBars` in action:

```
//--- Create a CoolBar containing a ToolBar and a Combo
CoolBar coolBar = new CoolBar(parent, SWT.BORDER);

//- Create the ToolBar, representing the 1st group of items in the CoolBar
ToolBar group1 = new ToolBar(coolBar, SWT.FLAT);
ToolItem item = new ToolItem(group1, SWT.NONE); // add a 1st item
item.setImage(icon1);
item.setToolTipText("Action 1");

item = new ToolItem(group1, SWT.SEPARATOR); // add a separator

item = new ToolItem(group1, SWT.NONE); // add a second item
item.setImage(icon2);
item.setToolTipText("Action 2");

// add the ToolBar as 1st item in the CoolBar
CoolItem coolItem = new CoolItem(coolBar, SWT.NONE);
coolItem.setControl(group1);
```

```
coolItem.setSize(group1.computeSize(SWT.DEFAULT, SWT.DEFAULT));

//- Create a Combo to add as 2nd item in the CoolBar
Combo combo = new Combo(coolBar, SWT.DROP_DOWN);
combo.setItems(new String[]{"item1", "item2", "item3"});

coolItem = new CoolItem(coolBar, SWT.NONE);
coolItem.setControl(combo);
coolItem.setSize(combo.computeSize(SWT.DEFAULT, SWT.DEFAULT));
```

Migrate existing Swing code

The migration of existing Swing code for a `JToolBar` shouldn't present any problem. The wrapper class `SWTToolBar`, included in the sample code provided with this tutorial, makes the migration easier by emulating the API of Swing as introduced in Migrate your Swing code to SWT with minimal change on page 14 . To migrate existing code using the wrapper class, you'll need to take the following steps:

- Search for occurrences of the Swing type `JToolBar` and replace them with the new wrapper type `SWTToolBar`.
- Search for constructors where a toolbar is created and add the reference to the parent of the tabbed pane in the arguments list.

Let's look at a migration example. Consider the following Swing code:

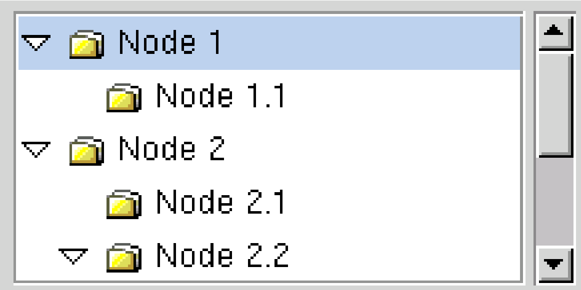
```
JToolBar toolBar = new JToolBar();
toolBar.add(anAction);
toolBar.add(aComponent);
toolBar.addSeparator();
toolBar.add(anotherAction);
parent.add(toolBar);
```

Here's what the code would look like migrated to SWT:

```
SWTToolBar toolBar = new SWTToolBar(parent);
toolBar.add(anAction);
toolBar.add(aComponent);
toolBar.addSeparator();
toolBar.add(anotherAction);
parent.add(toolBar);
```

JTree

SWT's equivalent for Swing's `JTree` is the component `Tree`. It can be used in combination with JFace's `TreeViewer`.



The constraints of using a pure SWT `Tree` without a JFace `TreeViewer` are the same as those for a table (see JTable on page 67). There is no data model and you have to create each tree node manually, as in the following example:

```
//--- Example of creation of a SWT Tree without TreeViewer
Tree tree = new Tree(parent, SWT.SINGLE | SWT.H_SCROLL | SWT.V_SCROLL);
// create a 1st root node "Node 1" containing 2 children "Node 1-1" & "Node 1-2"
TreeItem node1 = new TreeItem(tree, SWT.NONE);
node1.setText("Node 1");
TreeItem node11 = new TreeItem(node1, SWT.NONE);
node11.setText("Node 1-1");
TreeItem node12 = new TreeItem(node1, SWT.NONE);
node12.setText("Node 1-2");

// create a 2nd root node "Node 2" containing 2 children "Node 2-1" & "Node 2-2"
TreeItem node2 = new TreeItem(tree, SWT.NONE);
node2.setText("Node 2");
TreeItem node21 = new TreeItem(node2, SWT.NONE);
node21.setText("Node 2-1");
TreeItem node22 = new TreeItem(node2, SWT.NONE);
node22.setText("Node 2-2");
```

JFace's TreeViewer

Most of the time, you wouldn't create a tree as shown above, but would instead use a JFace `TreeViewer`. A `TreeViewer` is a JFace viewer created on top of an SWT `Tree`. The viewer automatically creates and sets up the `TreeItems` to represent a data model supplied by a content provider in a text/icon form defined by a label provider. In this way, you have a mechanism that is similar to Swing's `TreeModel/TreeCellRenderer` mechanism.

For more information on JFace's viewers, read Data models and cell renderers vs. content providers and label providers on page 12 , or read the articles listed in the Resources on page 95 . For concrete examples showing how to use a `TreeViewer`, you should read in particular "Using the Eclipse GUI outside the Eclipse Workbench" by Adrian Van Emmenis, and "How to use the JFace Tree Viewer" by Chris Grindstaff.

Tree items

If you use a JFace `TreeViewer`, you don't have to care about the `TreeItems` of the tree, because they are automatically created by the viewer. However, in some cases it can be useful to work with the `TreeItems` directly, even if they are automatically created.

By using the API of `Tree`, you can get the list of all the root `TreeItems` and navigate through all the items of the tree. By invoking `setBackground(Color)` or `setForeground(Color)`, you can modify the colors of single items. This is something that you can't do with the API of JFace's `TreeViewer` and its label provider.

Expand/collapse items

As in Swing, you can programmatically expand or collapse items. `TreeViewer` provides several methods to expand the tree up to a specific depth, or to expand or collapse the nodes corresponding to some specific elements in the data model. Check the API of the following methods:

- `AbstractTreeViewer.expandAll()`
- `AbstractTreeViewer.expandtoLevel(...)`
- `AbstractTreeViewer.setExpandedElements(Object[])`
- `AbstractTreeViewer.setExpandedState(Object, boolean)`

Another way to expand or collapse an item is to get the `TreeItem` of its node, and then invoke the method `setExpanded(boolean)` on it.

Editing

A limitation of JFace's `TreeViewer` is that it doesn't allow the editing of nodes, as its Swing equivalent does. If you really need to do this, SWT provides `TreeEditor`, which can be installed on top of an SWT `Tree`. If you use it in combination with a JFace `TreeViewer` and a content provider, you will have to write some code to modify the data model once the editing of a node is completed.

If you want an example showing how to use `TreeEditor`, look at the code snippets at the dev.eclipse.org site (see Resources on page 95 for a link).

Management of the selection

SWT has no equivalent for Swing's `SelectionModel`. You can define whether multiple selection is allowed or not by using one of two style constants, `SWT.MULTI` or `SWT.SINGLE`, when constructing the tree. You can't switch from one mode to the other after the tree has been created. You can set and get the selection programmatically in two different ways:

- SWT's `Tree` provides simple methods to set or get the selection. These methods work with the `TreeItems` populating the tree.

- JFace's `TreeViewer` provides two methods, `getSelection()` and `setSelection(ISelection, boolean)`, that are inherited from `StructuredViewer` and work on a higher abstraction level. The `ISelection` object returned or used by these methods is in fact a `StructuredSelection`. This object provides an iterator or an array containing the selected elements as provided by the content provider, and is independent from their string representation or their representation order.

Events

An SWT Tree throws two kind of events:

- A `SelectionEvent` is thrown to notify the listeners that a change occurred in the selection. To detect a change in the selection, register a `SelectionListener` by using the `addSelectionListener(SelectionListener)` method. The listener method that is triggered by the event and should be implemented is `SelectionListener.widgetSelected(SelectionEvent)`.

- A `TreeEvent` is thrown each time a node is expanded or collapsed. You can receive this event by registering a `TreeListener` on the `Tree`.

Migrate existing Swing code

The migration of existing Swing code for a `JTree` doesn't present any problem as long as you don't need complex renderers that can't be realized with a JFace label provider.

The wrapper class `SWTTree`, included with the sample code provided with this tutorial, makes the migration easier by emulating the API of Swing as outlined in Migrate your Swing code to SWT with minimal change on page 14 . You don't have to port your original Swing `TreeModel`.

To migrate existing code using the wrapper class, follow these steps:

- Search for occurrences of the Swing type `JTree` and replace them with the new wrapper type `SWTTree`.
- Search for constructors where a tree is created and add the reference to the parent of the tree in the arguments list.
- The Swing trees of your application are probably contained in `JScrollPanes`. Modify the code so that no `JScrollPane` is created and the trees are added directly to their parent.
- Convert any Swing renderers into `SWTCellRenderers`.

Let's look at a migration example. Consider the following Swing code:

```
TreeModel model = ...;
JTree tree = new JTree(model);
tree.setRootVisible(false);
tree.expandRow(0);
tree.addTreeSelectionListener(new TreeSelectionListener() {
  public void valueChanged(TreeSelectionEvent e) {
    // do action
  }
});
parent.add(new JScrollPane(tree));
```

After migration, the equivalent SWT code would look like this:

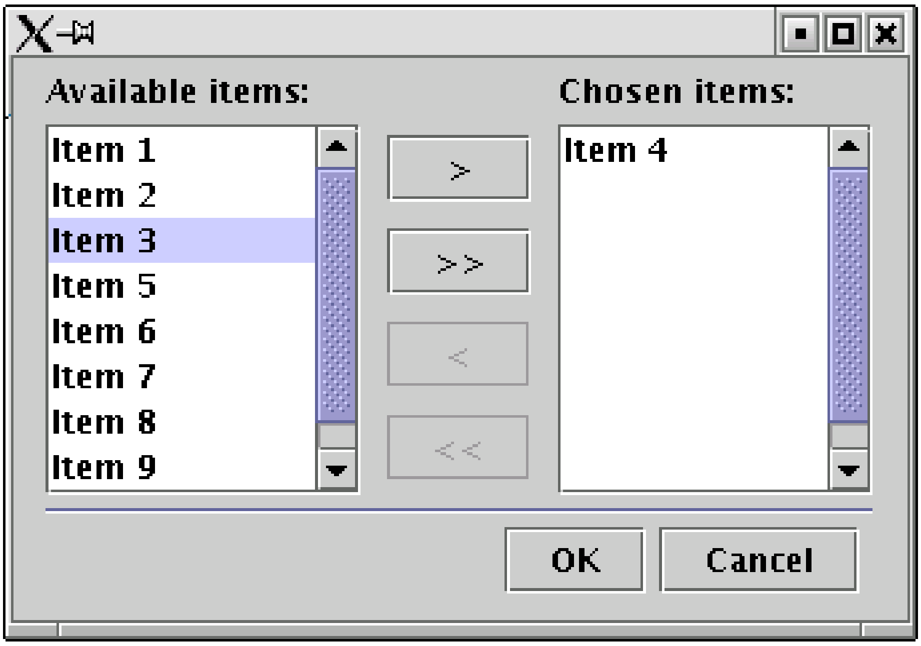
```
TreeModel model = ...;
SWTTree tree = new SWTTree(parent, model);
tree.setRootVisible(false);
tree.expandRow(0);
tree.addTreeSelectionListener(new TreeSelectionListener() {
  public void valueChanged(TreeSelectionEvent e) {
    // do action
  }
});
parent.add(tree);
```

Section 6. Complete example: Migrating a Swing dialog

Our sample dialog

In this section, we are going to apply the migration techniques and the wrapper classes introduced in this tutorial to migrate a complete Swing panel to SWT.

The following screenshot shows the Swing panel that we are going to migrate to SWT:



This is a fairly common dialog. It allows the user to select one or more items from a list of available items. The list on the left-hand side is a list of available items that have not been chosen by the user yet; the list on the right-hand side is the list of the items that have been chosen by the user. From top to bottom, the buttons between the two lists allow the user to:

- Move the selected items from the list on the left side to the list on the right side.
- Move all the items from the list on the left side to the list on the right side.
- Move the selected items from the list on the right side to the list on the left side.
- Move all the items from the list on the right side to the list on the left side.

The status of the buttons (enabled or disabled) depends on the selection and on whether or not the lists are empty:

- The buttons to move the selected items from one list to the other are enabled only when at least one item in the source list is selected.

- The buttons to move all the items from one list to the other are enabled only when the source list contains at least one item.

The **OK** and **Cancel** buttons on the bottom of the panel trigger two methods (`performOK()` and `performCancel()`) that can be overloaded.

This panel is a simple but quite useful example, illustrating the typical Swing-to-SWT migration issues we discussed earlier. The layout of the component is realized by using a complex arrangement of invisible panels using different layout managers. The AWT layout managers used by this panel are `FlowLayout`, `BorderLayout` and `GridLayout`. The components of the dialog interact with each other through event listeners: a change in the selection of the lists modifies the status of the buttons, and the buttons modify the content of the lists.

The content of the lists is defined by using customized `ListModels`.

If you wanted to migrate such a panel to SWT without using the migration techniques presented in this tutorial, it would be probably quickest to rewrite the whole panel from scratch, because almost none of the existing code could be reused; the layout managers, the events and the API of the components are different. In the following panels, we'll see how to use our migration techniques to make that migration a lot easier.

Source code of the Swing panel

Here is the source code of the Swing panel presented on the previous panel (See the SwingSamplePanel.java file in the j-swing2swtsrc.zip download available in Resources on page 95 .)

The class contains a main method that allows you to test the panel without having to integrate it in an application. To compile and run this sample, follow these steps:

1. Save the file SwingSamplePanel.java in a local directory.
2. Compile it by using the command `javac SwingSamplePanel.java`.
3. Launch it by using the command `java -classpath . SwingSamplePanel`.

```
import java.awt.*;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.*;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

public class SwingSamplePanel extends JPanel implements ListSelectionListener {

  private JList leftList, rightList;
  private JButton selectButton, selectAllButton;
  private JButton deselectButton, deselectAllButton;

  private DefaultListModel leftListModel = new DefaultListModel();
  private DefaultListModel rightListModel = new DefaultListModel();

  public SwingSamplePanel() {
    JPanel content = new JPanel(new BorderLayout(5, 5));
    add(content);
    content.add(BorderLayout.SOUTH, createButtonsPanel());
    content.add(BorderLayout.CENTER, createSoshPanel());
```

```
    initContent();
  }

  protected JComponent createSoshPanel() {
    JPanel mainPanel = new JPanel(new BorderLayout(5, 5));

    JPanel leftPanel = new JPanel(new BorderLayout(5, 5));
    leftPanel.add(BorderLayout.NORTH, new JLabel("Available items:"));
    leftList = new JList(leftListModel);
    leftList.setPreferredSize(new Dimension(100, 150));
    leftList.getSelectionModel().addListSelectionListener(this);
    leftPanel.add(new JScrollPane(leftList));
    mainPanel.add(BorderLayout.WEST, leftPanel);

    JPanel centerPanel = new JPanel(new BorderLayout());
    mainPanel.add(centerPanel);

    JPanel p1 = new JPanel();
    centerPanel.add(BorderLayout.SOUTH, p1);
    JPanel p2 = new JPanel(new BorderLayout());
    p1.add(p2);
    JPanel buttonsPanel = new JPanel(new GridLayout(0, 1, 10, 10));
    p2.add(BorderLayout.NORTH, buttonsPanel);

    selectButton = new JButton(">");
    selectButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Object[] selectedItems = leftList.getSelectedValues();
        for (int i = 0; i < selectedItems.length; i++) {
          rightListModel.addElement(selectedItems[i]);
          leftListModel.removeElement(selectedItems[i]);
          updateButtonsState();
        }
      }
    });
    buttonsPanel.add(selectButton);

    selectAllButton = new JButton(">>");
    selectAllButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Object[] items = leftListModel.toArray();
        for (int i = 0; i < items.length; i++) {
          rightListModel.addElement(items[i]);
          leftListModel.removeElement(items[i]);
          updateButtonsState();
        }
      }
    });
    buttonsPanel.add(selectAllButton);

    deselectButton = new JButton("<");
    deselectButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Object[] selectedItems = rightList.getSelectedValues();
        for (int i = 0; i < selectedItems.length; i++) {
          leftListModel.addElement(selectedItems[i]);
          rightListModel.removeElement(selectedItems[i]);
          updateButtonsState();
        }
      }
    });
    buttonsPanel.add(deselectButton);

    deselectAllButton = new JButton("<<");
```

```
    deselectAllButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Object[] items = rightListModel.toArray();
        for (int i = 0; i < items.length; i++) {
          leftListModel.addElement(items[i]);
          rightListModel.removeElement(items[i]);
          updateButtonsState();
        }
      }
    });
    buttonsPanel.add(deselectAllButton);

    JPanel rightPanel = new JPanel(new BorderLayout(5, 5));
    rightPanel.add(BorderLayout.NORTH, new JLabel("Chosen items:"));
    rightList = new JList(rightListModel);
    rightList.setPreferredSize(new Dimension(100, 150));
    rightList.getSelectionModel().addListSelectionListener(this);
    rightPanel.add(new JScrollPane(rightList));
    mainPanel.add(BorderLayout.EAST, rightPanel);

    updateButtonsState();
    return mainPanel;
  }

  protected JComponent createButtonsPanel() {
    JPanel buttonsPanel = new JPanel(new BorderLayout());
    buttonsPanel.add(BorderLayout.NORTH, new JSeparator());
    JPanel subPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
    buttonsPanel.add(BorderLayout.CENTER, subPanel);

    JButton okButton = new JButton("OK");
    okButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        performOK();
      }
    });
    subPanel.add(okButton);

    JButton cancelButton = new JButton("Cancel");
    cancelButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        performCancel();
      }
    });
    subPanel.add(cancelButton);

    return buttonsPanel;
  }

  protected void initContent() {
    for (int i = 0; i < 10; i++) {
      leftListModel.addElement("Item " + (i + 1));
    }
    rightListModel.addElement("Item 11");
  }

  protected void performOK() {
    System.out.println("OK performed");
  }

  protected void performCancel() {
    System.out.println("Cancel performed");
  }
```

```
  private void updateButtonsState() {
    selectButton.setEnabled(!leftList.getSelectionModel().isSelectionEmpty());
    selectAllButton.setEnabled(!leftListModel.isEmpty());
    deselectButton.setEnabled(!rightList.getSelectionModel().isSelectionEmpty());
    deselectAllButton.setEnabled(!rightListModel.isEmpty());
  }

  // Implementation of ListSelectionListener
  public void valueChanged(ListSelectionEvent e) {
    if (e.getSource() == leftList.getSelectionModel()
      || e.getSource() == rightList.getSelectionModel()) {
      updateButtonsState();
    }
  }

  public static void main(String[] args) {
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(new SwingSamplePanel());
    frame.pack();
    frame.setVisible(true);
  }

}
```

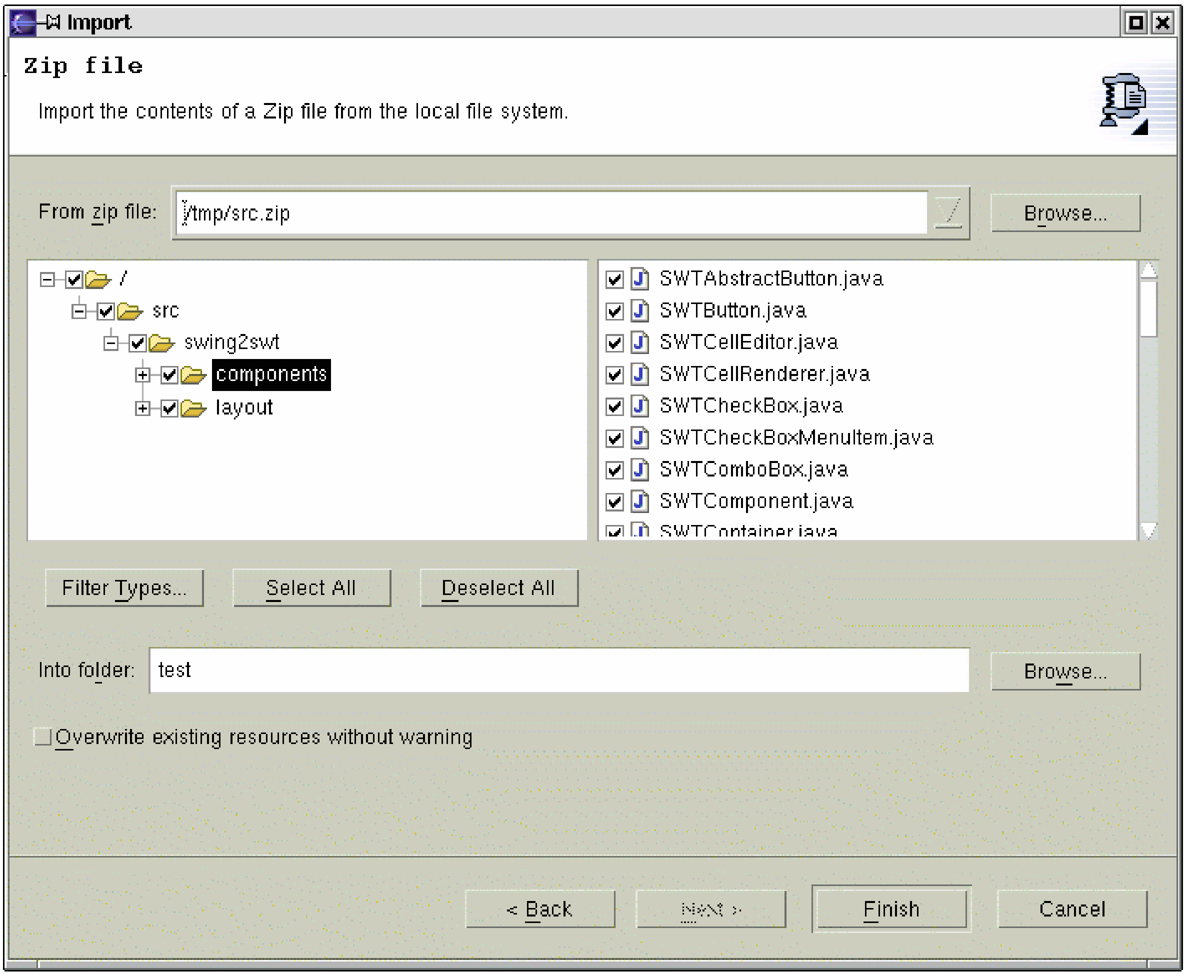
Set up your build and run environment

In this panel, we are going to set up a Java project in Eclipse, which is able to compile and run a standalone SWT/JFace application using the wrapper classes provided with this tutorial. The version of Eclipse I use in this tutorial is 2.1.

1. Create a new Java project in Eclipse called `test`. Use the subdirectory `./src/` to the source files.

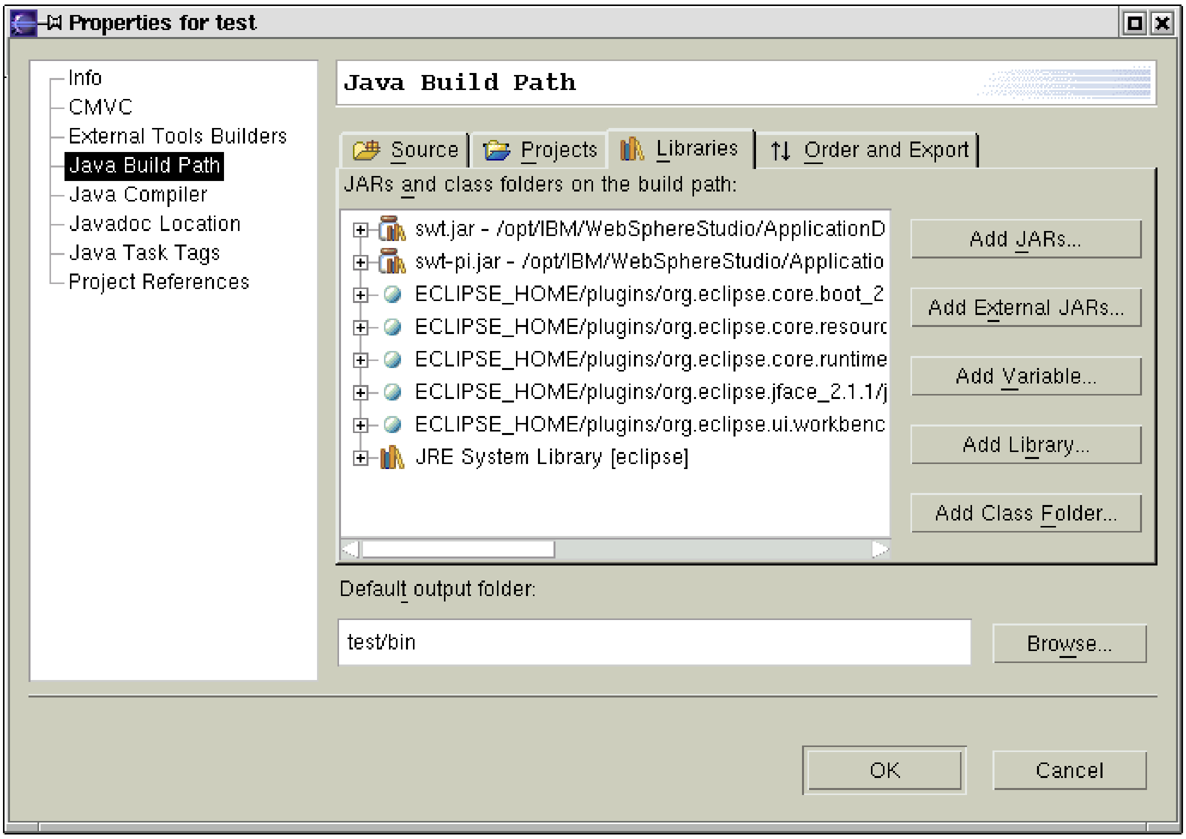
2. Download j-swing2swtsrc.zip from Resources on page 95 , which contains the sample code provided by this tutorial, into a local directory.

3. Import the content of the zip file in the new created Java project by using **File=>Import...=>Zip File.** Then, import all the files contained in downloaded zip file into the project directory.

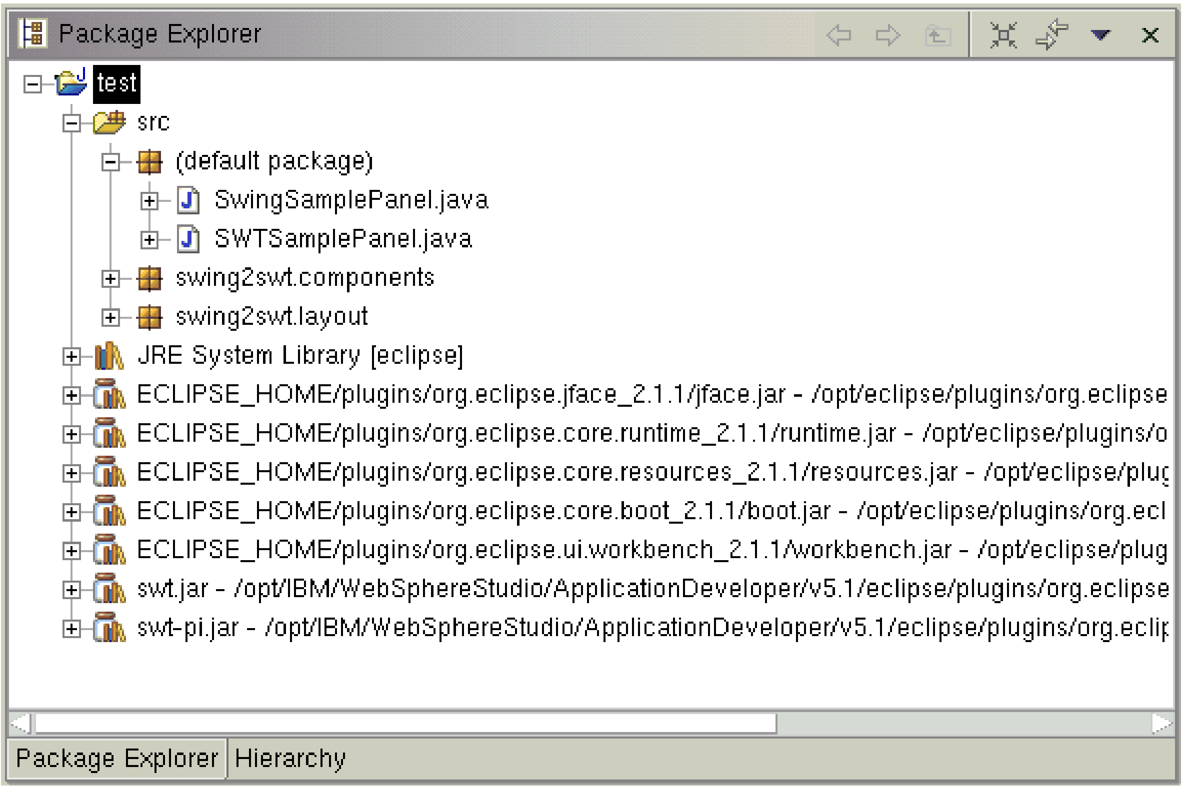


4. The imported classes (the Java packages `swing2swt.components` and `swing2swt.layout`) should be compiled. You will get some compilation errors, because the JAR files for SWT and JFace are not in the classpath yet.

5. Right-click on the Java project and open its properties. Go in the category *Java Build Path* and add the following JAR files, taken from the installation directory of Eclipse:
   - boot.jar from the plugin `org.eclipse.core.boot`.
   - resources.jar from the plugin `org.eclipse.core.resources`.
   - runtime.jar from the plugin `org.eclipse.core.runtime`.
   - jface.jar from the plugin `org.eclipse.jface`.
   - workbench.jar from the plugin `org.eclipse.ui.workbench`.
   - swt.jar from the plugin `org.eclipse.swt.XXX` (where XXX is the name of the platform you use); some platforms (such as GTK) provide several JAR files for SWT. Add all of them.



Once the JAR files are added in the classpath, the compilation errors should disappear. The next screenshot shows what the package explorer should look like at this point:



For more information on launching SWT/JFace applications outside Eclipse, read the developerWorks article "Using the Eclipse GUI outside the Eclipse workbench" by Adrian Van Emmenis. You can find a link in Resources on page 95 .

Migrate the Swing code to SWT

By using the wrapper classes and layout managers provided with this tutorial, you can migrate the Swing code with a sequence of search-and-replace actions.

First, copy the original class `SwingSamplePanel` into a new class named `SWTSamplePanel`.

Next, in the new class, remove all the `import` statements and replace them with these statements:

```
import swing2swt.components.*;
import swing2swt.layout.*;
import java.awt.event.*;
import javax.swing.event.*;
```

Now, save the file and try to compile it; you'll get about 100 compilation errors, saying that the Swing classes (`JPanel`, `JList`, `JButton`, etc...) cannot be located.

Next, use the automatic search-and-replace functions of your editor to successively:

- Replace all occurrences of `JPanel` with `SWTPanel`
- Replace all occurrences of `JList` with `SWTList`
- Replace all occurrences of `JButton` with `SWTButton`
- Replace all occurrences of `JComponent` with `SWTComponent`
- Replace all occurrences of `JLabel` with `SWTLabel`
- Replace all occurrences of `JSeparator` with `SWTSeparator`

Save the file and try to compile it again; the number of compilation errors should now be reduced to about 50.

Most of the remaining errors complain that the constructor of the wrapper classes is not defined. As you learned earlier, all the wrapper classes require the reference to the parent component to be passed as the first argument in their constructor. This can be done in a semi-automatic way by using the search function of your text editor. Search from the beginning of the class for all the occurrences of the string `add(`. This will show you all the places in the code where a component is added to its parent.

The lines of code found by the search function show you the parent container in which each component is contained, and should have one of the following forms:

```
aContainer.add(aComponent);
aContainer.add(aConstraint, aComponent);
```

where *`aContainer`* is the parent container where *`aComponent`* is added. For each occurrence found by the search function, notice the name of the component (*`aComponent`*) and the name of its container (*`aContainer`*). Search in the code where the component (*`aComponent`*) is created and add the container (*`aContainer`*) as first argument in the constructor.

For example, the first occurrence of **`add(`** found in the source code is at line 17, `add(content)` (which is equivalent to `this.add(content)`). The component is `content` and the container is `this`. The component `content` is created at line 16: `SWTPanel content = new SWTPanel(new BorderLayout(5, 5));`. The name of the container (`this`) should be added as the first argument in the constructor.

Thus, the original code

```
SWTPanel content = new SWTPanel(new BorderLayout(5, 5));
add(content);
```

should be transformed into:

```
SWTPanel content = new SWTPanel(this, new BorderLayout(5, 5));
add(content);
```

Here's another example. Consider the block of code at lines 26-32:

```
SWTPanel leftPanel = new SWTPanel(new BorderLayout(5, 5));
leftPanel.add(BorderLayout.NORTH, new SWTLabel("Available items:"));
leftList = new SWTList(leftListModel);
leftList.setPreferredSize(new Dimension(100, 150));
leftList.getSelectionModel().addListSelectionListener(this);
leftPanel.add(new JScrollPane(leftList));
mainPanel.add(BorderLayout.WEST, leftPanel);
```

This code should be modified as follows:

```
SWTPanel leftPanel = new SWTPanel(mainPanel, new BorderLayout(5, 5));
leftPanel.add(BorderLayout.NORTH, new SWTLabel(leftPanel, "Available items:"));
leftList = new SWTList(leftPanel, leftListModel);
leftList.setPreferredSize(new Dimension(100, 150));
leftList.getSelectionModel().addListSelectionListener(this);
leftPanel.add(leftList);
mainPanel.add(BorderLayout.WEST, leftPanel);
```

Note that line 31 -- `leftPanel.add(new JScrollPane(leftList));` -- was modified into `leftPanel.add(`**`leftList`**`);` because an SWT list is by nature scrollable and doesn't have to be added into a scrollpane like in Swing.

For the same reason -- because an SWT component needs a reference of its parent container at construction time -- we have to slightly modify the signature of the methods `createSoshPanel()` and `createButtonsPanel()` to pass the reference of the parent container as a parameter. First, we'll modify `createSoshPanel()`. Here's the code before modification:

```
protected SWTComponent createSoshPanel() {
```

```
  SWTPanel mainPanel = new SWTPanel(new BorderLayout(5, 5));
  (...)
```

And here's the modified code:

```
protected SWTComponent createSoshPanel(SWTContainer parent) {
  SWTPanel mainPanel = new SWTPanel(parent, new BorderLayout(5, 5));
  (...)
```

Next, let's modify `createButtonsPanel()`. Here's the code before modification:

```
  protected SWTComponent createButtonsPanel() {
    SWTPanel buttonsPanel = new SWTPanel(new BorderLayout());
    (...)
```

And here's the modified code:

```
  protected SWTComponent createButtonsPanel(SWTContainer parent) {
    SWTPanel buttonsPanel = new SWTPanel(parent, new BorderLayout());
    (...)
```

Finally, we need to modify the constructor `SWTSamplePanel`, which invokes these methods. Here's the code before modification:

```
  public SWTSamplePanel() {
    SWTPanel content = new SWTPanel(this, new BorderLayout(5, 5));
    add(content);
    content.add(BorderLayout.SOUTH, createButtonsPanel());
    content.add(BorderLayout.CENTER, createSoshPanel());
    initContent();
  }
```

And here's the code after modification:

```
public SWTSamplePanel(SWTContainer parent) {
  super(parent);
  SWTPanel content = new SWTPanel(this, new BorderLayout(5, 5));
  add(content);
  content.add(BorderLayout.SOUTH, createButtonsPanel(content));
  content.add(BorderLayout.CENTER, createSoshPanel(content));
  initContent();
}
```

Now, save the file and try to compile it. The number of compilation errors should have been reduced to about 25. Most of these errors are due to some missing import statements. Add the following import statements at the beginning of the class:

```
import javax.swing.DefaultListModel;
import java.awt.Dimension;
```

The number of compilation errors should have been reduced to four, all of them contained in the `main()` method. Let's fix these now. Replace the `main()` method used to test the code with this one:

```
public static void main(String[] args) {
  org.eclipse.swt.widgets.Display display =
        new org.eclipse.swt.widgets.Display();
  org.eclipse.swt.widgets.Shell shell =
        new org.eclipse.swt.widgets.Shell(display);
  shell.setLayout(new BorderLayout());
  new SWTSamplePanel(new SWTContainer(shell));
  shell.pack();
  shell.open();
  while (!shell.isDisposed ()) {
    if (!display.readAndDispatch ()) display.sleep ();
  }
  display.dispose ();
}
```

Source code for the migrated panel

Here is the complete source code of the panel migrated to SWT. (See the SWTSamplePanel.java file in the j-swing2swtsrc.zip download available in Resources on page 95 .)

```
import swing2swt.components.*;
import swing2swt.layout.*;
import java.awt.event.*;
import javax.swing.event.*;
import javax.swing.DefaultListModel;
import java.awt.Dimension;

public class SWTSamplePanel extends SWTPanel implements ListSelectionListener {

  private SWTList leftList, rightList;
  private SWTButton selectButton, selectAllButton;
  private SWTButton deselectButton, deselectAllButton;

  private DefaultListModel leftListModel = new DefaultListModel();
  private DefaultListModel rightListModel = new DefaultListModel();

  public SWTSamplePanel(SWTContainer parent) {
    super(parent);
    SWTPanel content = new SWTPanel(this, new BorderLayout(5, 5));
    add(content);
    content.add(BorderLayout.SOUTH, createButtonsPanel(content));
    content.add(BorderLayout.CENTER, createSoshPanel(content));
    initContent();
  }

  protected SWTComponent createSoshPanel(SWTContainer parent) {
    SWTPanel mainPanel = new SWTPanel(parent, new BorderLayout(5, 5));

    SWTPanel leftPanel = new SWTPanel(mainPanel, new BorderLayout(5, 5));
    leftPanel.add(
      BorderLayout.NORTH,
      new SWTLabel(leftPanel, "Available items:"));
    leftList = new SWTList(leftPanel, leftListModel);
    leftList.setPreferredSize(new Dimension(100, 150));
    leftList.getSelectionModel().addListSelectionListener(this);
    leftPanel.add(leftList);
    mainPanel.add(BorderLayout.WEST, leftPanel);
```

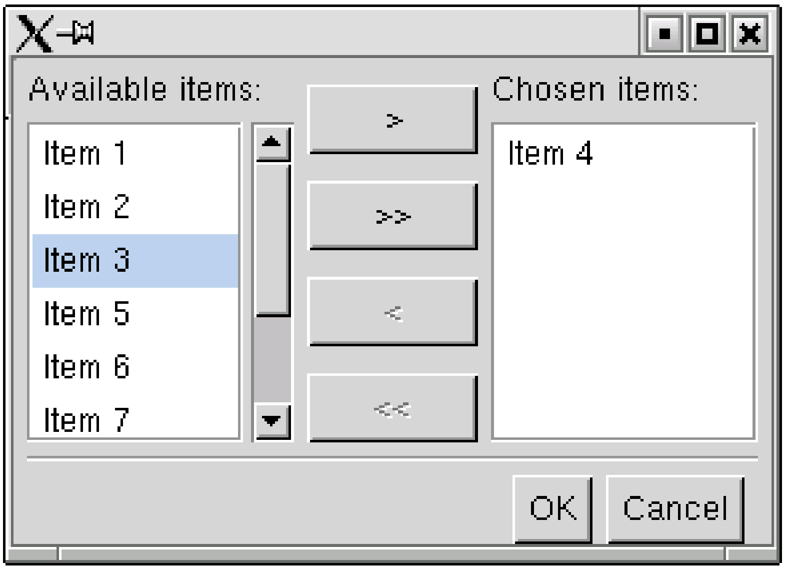
```
SWTPanel centerPanel = new SWTPanel(mainPanel, new BorderLayout());
mainPanel.add(centerPanel);

SWTPanel p1 = new SWTPanel(centerPanel);
centerPanel.add(BorderLayout.SOUTH, p1);
SWTPanel p2 = new SWTPanel(p1, new BorderLayout());
p1.add(p2);
SWTPanel buttonsPanel = new SWTPanel(p2, new GridLayout(0, 1, 10, 10));
p2.add(BorderLayout.NORTH, buttonsPanel);

selectButton = new SWTButton(buttonsPanel, ">");
selectButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    Object[] selectedItems = leftList.getSelectedValues();
    for (int i = 0; i < selectedItems.length; i++) {
      rightListModel.addElement(selectedItems[i]);
      leftListModel.removeElement(selectedItems[i]);
      updateButtonsState();
    }
  }
});
buttonsPanel.add(selectButton);

selectAllButton = new SWTButton(buttonsPanel, ">>");
selectAllButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    Object[] items = leftListModel.toArray();
    for (int i = 0; i < items.length; i++) {
      rightListModel.addElement(items[i]);
      leftListModel.removeElement(items[i]);
      updateButtonsState();
    }
  }
});
buttonsPanel.add(selectAllButton);

deselectButton = new SWTButton(buttonsPanel, "<");
deselectButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    Object[] selectedItems = rightList.getSelectedValues();
    for (int i = 0; i < selectedItems.length; i++) {
      leftListModel.addElement(selectedItems[i]);
      rightListModel.removeElement(selectedItems[i]);
      updateButtonsState();
    }
  }
});
buttonsPanel.add(deselectButton);

deselectAllButton = new SWTButton(buttonsPanel, "<<");
deselectAllButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    Object[] items = rightListModel.toArray();
    for (int i = 0; i < items.length; i++) {
      leftListModel.addElement(items[i]);
      rightListModel.removeElement(items[i]);
      updateButtonsState();
    }
  }
});
buttonsPanel.add(deselectAllButton);

SWTPanel rightPanel = new SWTPanel(mainPanel, new BorderLayout(5, 5));
rightPanel.add(
```

```
      BorderLayout.NORTH,
      new SWTLabel(rightPanel, "Chosen items:"));
    rightList = new SWTList(rightPanel, rightListModel);
    rightList.setPreferredSize(new Dimension(100, 150));
    rightList.getSelectionModel().addListSelectionListener(this);
    rightPanel.add(rightList);
    mainPanel.add(BorderLayout.EAST, rightPanel);

    updateButtonsState();
    return mainPanel;
  }

  protected SWTComponent createButtonsPanel(SWTContainer parent) {
    SWTPanel buttonsPanel = new SWTPanel(parent, new BorderLayout());
    buttonsPanel.add(BorderLayout.NORTH, new SWTSeparator(buttonsPanel));
    SWTPanel subPanel =
      new SWTPanel(buttonsPanel, new FlowLayout(FlowLayout.RIGHT));
    buttonsPanel.add(BorderLayout.CENTER, subPanel);

    SWTButton okButton = new SWTButton(subPanel, "OK");
    okButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        performOK();
      }
    });
    subPanel.add(okButton);

    SWTButton cancelButton = new SWTButton(subPanel, "Cancel");
    cancelButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        performCancel();
      }
    });
    subPanel.add(cancelButton);

    return buttonsPanel;
  }

  protected void initContent() {
    for (int i = 0; i < 10; i++) {
      leftListModel.addElement("Item " + (i + 1));
    }
    rightListModel.addElement("Item 11");
  }

  protected void performOK() {
    System.out.println("OK performed");
  }

  protected void performCancel() {
    System.out.println("Cancel performed");
  }

  private void updateButtonsState() {
    selectButton.setEnabled(!leftList.getSelectionModel().isSelectionEmpty());
    selectAllButton.setEnabled(!leftListModel.isEmpty());
    deselectButton.setEnabled(
      !rightList.getSelectionModel().isSelectionEmpty());
    deselectAllButton.setEnabled(!rightListModel.isEmpty());
  }

  // Implementation of ListSelectionListener
  public void valueChanged(ListSelectionEvent e) {
    if (e.getSource() == leftList.getSelectionModel()
```

```
        || e.getSource() == rightList.getSelectionModel()) {
      updateButtonsState();
    }
  }

  public static void main(String[] args) {
    org.eclipse.swt.widgets.Display display =
      new org.eclipse.swt.widgets.Display();
    org.eclipse.swt.widgets.Shell shell =
      new org.eclipse.swt.widgets.Shell(display);
    shell.setLayout(new BorderLayout());
    new SWTSamplePanel(new SWTContainer(shell));
    shell.pack();
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
}
```

Migrated panel

You can now launch the migrated code:



The migrated panel has the same layout and same behavior as the original Swing panel. The modifications that we have made in the code were purely syntactic. The original layout managers, event listeners and data models have remained unchanged. We achieved this migration without reengineering or even deeply understanding the original Swing code.

Section 7. Wrap-up and resources

Summary

In this tutorial, you have learned the main differences between AWT/Swing and SWT/JFace. You also how to simplify the migration of an existing Swing application by first porting the AWT layout managers to SWT, then creating wrapper classes around SWT controls that emulate the API of Swing, and finally converting the SWT events into AWT events sent to AWT listeners. You have also seen that Swing data models can be easily reused in SWT/JFace.

You studied in detail the equivalent SWT component for each Swing component, and saw the differences that exist and the issues you have to expect during the migration of your application.

The sample code used in this tutorial provided a guide for applying the migration techniques described in the first part of the tutorial to most of the standard Swing components. By using this sample code in your project, you should be able to migrate a Swing UI using standard components and layout managers. I've even offered a simplified the migration of code to a series of search-and-replace operations.

Finally, you saw a concrete example, where a Swing panel was ported to SWT by using this method. Hopefully all this will help you port your legacy Swing and AWT code to the higher-performing SWT toolkit.

Resources

Source code

- Download the sample code used in this tutorial -- the AWT layout managers converted to SWT and the wrapper classes.

APIs

- Consult the *Eclipse and SWT API* at Eclipse.org.

- Consult the Swing API in the *API documentation of the J2SE platform*.

General Eclipse and SWT articles

- Visit *Eclipse.org* for downloads, documentation, mail archives, and articles.

- Check out the IBM *WebSphere Studio Application Developer* 5.1.

- For Eclipse project development plans, a FAQ, and a list of handy SWT code snippets, check out the *component development resources*.

- See *collection of code snippets* for the code illustrating the use of `TreeEditor`.

- Read more about the *GEF project*.
- In "*SWT: The Standard Widget Toolkit, Part 1*" (*Eclipse Corner*, March 2001), Steve Northover gives an introduction to the design strategies used in SWT.
- In "*Plug a Swing-based development tool into Eclipse* " (*developerWorks*, October 2002), Terry Chan describes how to integrate a Swing application into the Eclipse platform.
- Read "*Understanding Layouts in SWT*" by Carolyn MacLeod and Shantha Ramachandran (*Eclipse Corner*, May 2002) to get an introduction to SWT's layouts.
- In their articles "*Getting your feet wet with the SWT StyledText widget*" and "*Into the deep end of the SWT StyledText widget*" (*Eclipse Corner*, September 2002), Lynne Kues and Knut Radloff explain how to use the `StyledText` widget to display and edit formatted text in SWT.

Articles on resource management and garbage collection in SWT

- In "*SWT: The Standard Widget Toolkit, Part 2*," (*Eclipse Corner*, November 2001), Steve Northover and Carolyn MacLeod provide a list of rules to follow to manage graphical resources when programming in SWT.
- "*SWT color model*," James Moody and Carolyn MacLeod (*Eclipse Corner,* April 2001) gives some tip about the management of color resources in SWT.

Articles on the JFace viewers API

- In his article "*Using the Eclipse GUI outside the Eclipse Workbench*" (*developerWorks*, January 2003), Adrian Van Emmenis demonstrates the use of JFace viewers, content providers, and label providers with SWT tables and trees.
- In his article "*Building and delivering a table editor with SWT/JFace*" (*Eclipse Corner*, July 2003), Laurent Gauthier explains how to build an editable and sortable table, using the `TableViewer` API of JFace.
- In "*How to use the JFace Tree Viewer*" (*Eclipse Corner*, May 2002), Chris Grindstaff explains how to use the JFace `TreeViewer` API.

Additional resources

- Download the latest *Eclipse technologies from IBM alphaWorks*.
- Get the latest news on the Websphere Studio tools at the *WebSphere Studio Zone*.
- See the *Java technology zone tutorials page* for a complete listing of free Java-related tutorials from *developerWorks*.

- Stay on top of the Eclipse platform at the *developerWorks Open source projects zone*.
- Find hundreds of articles about every aspect of Java programming in the *developerWorks Java technology zone*.

Feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.