The Definitive Guide to SWT and JFace

ROB WARNER
WITH
ROBERT HARRIS

Apress®

The Definitive Guide to SWT and JFace

ISBN (pbk): 1-59059-325-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin

Technical Reviewer: Gábor Lipták

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, John Franklin, Jason Gilmore, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Project Manager: Tracy Brown Collins

Copy Edit Manager: Nicole LeClerc

Copy Editors: Susannah Pfalzer, Kim Wimpsett

Production Manager: Kari Brooks

Production Editor: Ellie Fountain

Compositor: Linda Weidemann, Wolf Creek Press

Proofreader: Patrick Vincent

Indexer: Kevin Broccoli

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 3

Your First SWT Application

Burgeoning programmers yearn to greet the world in code; this chapter guides you through creating your first application in SWT—the inescapable "Hello, World." It explains how SWT works, and leads you through the major objects you'll deal with when using SWT. It discusses the lifecycle of SWT widgets as well.

"Hello, World" in SWT

You must apply a few minor changes to your BlankWindow program from the previous chapter to convert it into the canonical "Hello, World" application. More specifically, you must create an instance of an `org.eclipse.swt.widgets.Label` object, set its text to the preferred message, and add the label to your form. The following code reflects these changes.

```
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.SWT;

public class HelloWorld
{
 public static void main(String[] args)
 {
  Display display = new Display();
  Shell shell = new Shell(display);
  Label label = new Label(shell, SWT.CENTER);
  label.setText("Hello, World");
  label.setBounds(shell.getClientArea());
  shell.open();
  while (!shell.isDisposed())
  {
   if (!display.readAndDispatch())
   {
    display.sleep();
   }
  }
```

```
  display.dispose();
 }
}
```

Compiling and Running the Program

Compiling `HelloWorld.java` should work similarly to the compile command from the previous chapter. From this point forward, we won't explicitly give instructions on the compilation or run steps, unless they vary from those examples presented in the previous chapter.

Compiling and running your programs from the command line soon becomes tedious and error prone. To address this issue, we provide an Ant build configuration file that you can use for the programs you develop in this book. To compile and run your programs, copy `build.xml` to the same directory as your source code and run Ant, specifying your main class name as the value for the property `main.class`. For example, to compile and run your HelloWorld program, type this:

```
ant -Dmain.class=HelloWorld
```

To just compile your program, you may omit the `main.class` property, and you must specify the compile target, like this:

```
ant compile
```

Listing 3-1 contains the Ant build file you'll use throughout the SWT portion of this book.

Listing 3-1. `build.xml`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="GenericSwtApplication" default="run" basedir=".">
 <description>
  Generic SWT Application build and execution file
 </description>

 <property name="main.class" value=""/>
 <property name="src"      location="."/>
 <property name="build"    location="."/>

 <!-- Update location to match your eclipse home directory -->
 <property name="ecl.home"  location="c:\eclipse"/>

 <!-- Update value to match your windowing system (win32, gtk, motif, etc.) -->
 <property name="win.sys"   value="win32"/>
```

```
 <!-- Update value to match your os (win32, linux, etc.) -->
 <property name="os.sys"   value="win32"/>

 <!-- Update value to match your architecture -->
 <property name="arch"    value="x86"/>

 <!-- Update value to match your SWT version -->
 <property name="swt.ver"   value="3.0.0"/>

 <!-- Do not edit below this line -->
 <property name="swt.subdir"
  location="${ecl.home}/plugins/org.eclipse.swt.${win.sys}_${swt.ver}"/>
 <property name="swt.jar.lib" location="${swt.subdir}/ws/${win.sys}"/>
 <property name="swt.jni.lib" location="${swt.subdir}/os/${os.sys}/${arch}"/>

 <path id="project.class.path">
  <pathelement path="${build}"/>
  <fileset dir="${swt.jar.lib}">
   <include name="**/*.jar"/>
  </fileset>
 </path>

 <target name="compile">
  <javac srcdir="${src}" destdir="${build}">
   <classpath refid="project.class.path"/>
  </javac>
 </target>

 <target name="run" depends="compile">
  <java classname="${main.class}" fork="true" failonerror="true">
   <jvmarg value="-Djava.library.path=${swt.jni.lib}"/>
   <classpath refid="project.class.path"/>
  </java>
 </target>
</project>
```

You must update your copy of the `build.xml` file as indicated in the file, updating the Eclipse home directory, the windowing system, the operating system, the architecture, and the SWT version.

What is Ant?

Ant, part of the Apache Jakarta project (`http://jakarta.apache.org/`), is a Java-specific "make" utility. Winner of the *Java Pro* 2003 Readers' Choice Award for Most Valuable Java Deployment Technology, it simplifies the build process for Java applications, and has become the Java industry standard build utility.

> Rather than using traditional "make" files, Ant uses XML configuration files for building applications. To build a Java application, then, you create an XML file that specifies your files, dependencies, and build rules, and then run Ant against that XML file. By default, Ant searches for a file called `build.xml`, but you can tell Ant to use other file names. You can specify targets and properties for Ant as well.
>
> For more information, and to download Ant, see the Ant Web site at `http://ant.apache.org/`.

Running this program displays a window that greets the world, as seen in Figure 3-1.

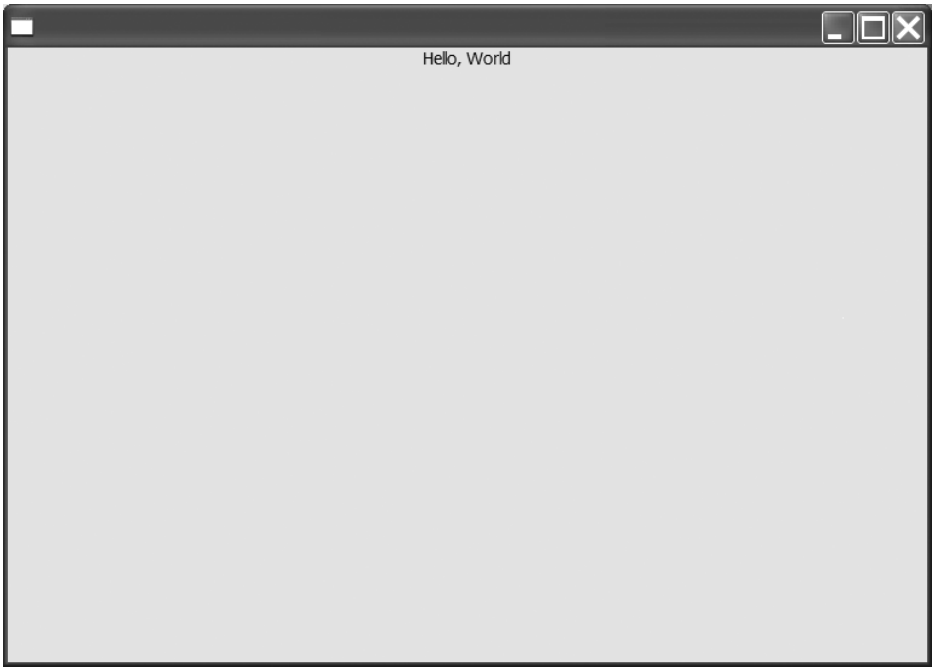

Figure 3-1. "Hello, World" in SWT

Understanding the Program

These lines give you the proper imports for the class:

```
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.SWT;
```

Most classes that use SWT import the `SWT` object and pieces of the `swt.widgets` package.

These lines create the `Display` object and the `Shell` object:

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

At a high level, the `Display` object represents the underlying windowing system. The `Shell` object is an abstraction that represents a top-level window when created with a `Display` object, as this one is. A more detailed introduction to the `Display` and `Shell` classes is presented later in this chapter.

Next, you create your label widget with this code:

```
Label label = new Label(shell, SWT.CENTER);
label.setText("Hello, World");
label.setBounds(shell.getClientArea());
```

The `Label` object is capable of displaying either simple text, as you use it here, or an image. The widget is constructed with a reference to a `Shell` object, which is an indirect descendant of the `Composite` class. `Composite` classes are capable of containing other controls. When SWT encounters this line, it knows to create the underlying windowing system's implementation of the label widget on the associated `Composite` object.

To make your window display, you call this:

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

This indicates to the underlying system to set the current shell visible, set the focus to the default button (if one exists), and make the window associated with the shell active. This displays the window and allows it to begin receiving events from the underlying windowing system.

The main loop of your application is this:

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

You'll have a loop similar to this in each of your SWT applications. In this loop, you first check to make sure that the user hasn't closed your main window. Because the window is still open, you next check your event queue for any messages that the windowing system or other parts of your application might have generated for you. If no events are in the queue, you sleep, waiting for the next event to arrive. When the next event arrives, you repeat the loop, ensuring first that the event didn't dispose your main window.

Finally, you call:

```
display.dispose();
```

Because your window has been disposed (by the user closing the window), you no longer need the resources of the windowing system to display the graphical components. Being good computing citizens, you now return these resources back to the system.

Understanding the Design Behind SWT

As you learned in Chapter 1, SWT uses the native widget library provided by the underlying OS, providing a Java veneer for your application to talk to. The lifecycle of the widget's Java object mirrors the lifecycle of the native widget it represents; when you create the Java widget, the native widget is created, and when the Java widget is destroyed the native widget is also destroyed. This design avoids issues with calling methods on a code object when the underlying widget hasn't yet been created, which can occur in other toolkits that don't match the lifecycles of the code widget and the native widget.

For example, compare the two-step creation process of the Microsoft Foundation Classes (MFC). If you want to create a button, you write code such as this:

```
CButton button; // Construct the C++ object on the stack
button.Create(<parameters>); // Create the Windows widget
```

Say you were to insert code between the construction of the C++ object and the native Windows widget that relied on the existence of the Windows widget; for example, code such as this:

```
CButton button; // Construct the C++ object on the stack
CString str = _T("Hi"); // Create a CString to hold the button text
button.SetWindowText(str); // Set the button text--PROBLEM!
button.Create(<parameters>); // Creates the Windows widget
```

The code compiles without complaint, but doesn't run as expected. The debug version of the code causes an assertion, and the behavior of the release version is undefined.

Parenting Widgets

Most GUIs require you to specify a parent for a widget before creating that widget, and the widget "belongs" to its parent throughout its lifecycle. The lifetime of the parent component constrains the lifetime of the child component. In addition, many native widgets have particular characteristics, or "styles," that you must set on their creation. For example, a button might be a push button or a checkbox. Because an SWT widget creates its corresponding native widget when it's constructed, it must have this information passed to its constructor. SWT widgets in general take two parameters: a parent and a style. The parent is typically of type `org.eclipse.swt.widgets.Widget` or one of

its subclasses. The styles available are integer constants defined in the `SWT` class; you can pass a single style, or use bitwise ORs to string several styles together. We'll introduce the styles available to a particular widget throughout this book as we discuss that widget.

Disposing Widgets

Swing developers will scoff at the information in this section, taking it as proof of SWT's inferiority. Java developers in general will likely feel a certain amount of distaste or discomfort here, for the message of this section is: you have to clean up after yourself. This notion, anathema to Java developers, flouts Java's garbage collection and returns a responsibility to developers that they'd long ago left behind.

Why do you have to dispose objects? Java's garbage collection manages memory admirably, but GUI resource management operates under heavier constraints. The number of available GUI resources is much more limited and, on many platforms, is a system-wide limitation. Because SWT works directly with the native underlying graphic resources, each SWT resource consumes a GUI resource, and timely release of that resource is essential not only for your SWT application's well-being, but also for the well-being of all other GUI programs currently running. Java's garbage collection carries no timeliness guarantees, and would make a poor manager of graphic resources for SWT. So, instead, you as programmer must assume the responsibility.

How onerous is the task? Actually, it's not much work at all. In their series of articles on SWT, Carolyn MacLeod and Steve Northover describe two simple rules to guide your disposal efforts:[1]

- If you created it, you dispose it.
- Disposing the parent disposes the children.

Rule 1: If You Created It, You Dispose It

In the section "Understanding the Design Behind SWT" earlier in this chapter, you learned that native resources are created when an SWT object is created. In other words, when you call the SWT object's constructor, the underlying native resource is created. So, if you code this, you've constructed an SWT `Color` object, and thus have allocated a color resource from the underlying GUI platform:

```
Color color = new Color(display, 255, 0, 0); // Create a red Color
```

Rule 1 says you created it, so you must dispose it when you are done using it, like this:

```
color.dispose(); // I created it, so I dispose it
```

1. Carolyn MacLeod and Steve Northover, *SWT: The Standard Widget Toolkit—Part 2: Managing Operating System Resources*, `www.eclipse.org/articles/swt-design-2/swt-design-2.html`.

However, if you don't call a constructor to get a resource, you must not dispose the resource. For example, consider the following code:

```
Color color = display.getSystemColor(SWT.COLOR_RED); // Get a red Color
```

Once again, you have a `Color` object that contains a red Color resource from the underlying platform, but you didn't allocate it. Rule 1 says you must not dispose it. Why not? It doesn't belong to you—you've just borrowed it, and other objects might still be using it or will use it. Disposing such a resource could be disastrous.

Rule 2: Disposing the Parent Disposes the Children

Calling `dispose()` on every SWT object created with `new` would quickly become tedious, and would doom SWT to a marginalized existence. However, SWT's designers realized that, and created a logical cascade of automatic disposal. Whenever a parent is disposed, all its children are disposed. This means that when a `Shell` is disposed, all the widgets belonging to it are automatically disposed as well. In fact, when any `Composite` is disposed, all its children are automatically disposed. You'll notice that you never call `label.dispose()` in your "Hello, World" program, even though you create a new `Label` object using a constructor. When the user closes the `Shell`, the `Label` object is automatically disposed for you.

You might be thinking that you'll never need to call `dispose()`, and that this entire section was a waste of space. Indeed, you'll likely write many applications in which all resources have a parent, and they'll all automatically be disposed for you. However, consider the case in which you want to change the font used in a `Text` control. You'd code something like this:

```
Text text = new Text(shell, SWT.BORDER); // Create the text field
Font font = new Font(display, "Arial", 14, SWT.BOLD); // Create the new font
text.setFont(font); // Set the font into the text field
```

The `Font` object you've created has no parent, and thus won't be automatically disposed, even when the `Shell` is closed and the `Text` object using it is disposed. You might chafe at the added burden of having to dispose of `font` yourself, but realize that `text` has no business disposing it—it doesn't own it. In fact, you might be using the same `Font` object for various other controls; automatic disposal would cause you serious problems.

Ignoring Disposed Objects

Astute readers will have noticed a hole in the mirrored lifecycle discussed in this chapter: what happens in the case where the Java object wrapping a native widget is still in scope, but the `Shell` object to which it belongs has been disposed? Or what about a widget that has had its `dispose` method invoked manually? Won't the native widget have been disposed? Can't you then call a method on the Java object when the underlying native widget doesn't exist?

The answer is indeed yes, and you can get yourself into a bit of trouble if you call methods on a widget whose native widget has been disposed. Once a widget has been disposed, even if it is still in scope, you shouldn't try to do anything with it. Yes, the Java object is still available, but the underlying peer has been destroyed. If you do try to do anything with a disposed widget, you'll get an `SWTException` with the text "Widget has been disposed." Consider the code in Listing 3-2.

Listing 3-2. `Broken.java`

```
import org.eclipse.swt.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;
public class Broken
{
 public static void main(String[] args)
 {
   Display display = new Display();
  Shell shell = new Shell(display);
  shell.setLayout(new RowLayout());
  Text text = new Text(shell, SWT.BORDER);
  shell.open();
  while (!shell.isDisposed())
  {
   if (!display.readAndDispatch())
   {
    display.sleep();
   }
  }
  System.out.println(text.getText()); // PROBLEM!
  display.dispose();
 }
}
```

The code compiles and runs, but after the main window is closed the console prints a stack trace that looks like this:

```
org.eclipse.swt.SWTException: Widget is disposed
 at org.eclipse.swt.SWT.error(SWT.java:2332)
 at org.eclipse.swt.SWT.error(SWT.java:2262)
 at org.eclipse.swt.widgets.Widget.error(Widget.java:385)
 at org.eclipse.swt.widgets.Control.getDisplay(Control.java:735)
 at org.eclipse.swt.widgets.Widget.isValidThread(Widget.java:593)
 at org.eclipse.swt.widgets.Widget.checkWidget(Widget.java:315)
 at org.eclipse.swt.widgets.Text.getText(Text.java:705)
 at Broken.main(Version.java:24)
```

What's more, when you run this on Windows XP, you get a dialog telling you that `javaw.exe` has encountered a problem, needs to close, and would you like to send Microsoft an error report?

The lesson is simple: once an object is disposed, whether its `dispose()` method has been explicitly invoked or its parent has been disposed, leave it alone.

Understanding the Display Object

The `Display` object represents the connection between the application-level SWT classes and the underlying windowing system implementation. The `Display` class is windowing-system dependent and might have some additional methods in its API on some platforms. Here we'll discuss only the part of the API that's universally available.

In general, each of your applications will have one, and only one, `Display` object (this is a limitation of some lower-level windowing systems). The thread that creates the `Display` object is, by default, the thread that executes the event loop and is known as the user-interface thread. You can call many of the member functions of widgets only from the user-interface thread. Other threads accessing these members will result in an `SWT.ERROR_THREAD_INVALID_ACCESS` type of exception.

One of the most important tasks of this class is its event-handling mechanism. The `Display` class maintains a collection of registered event listeners, reads events from the lower-level operating-system event queue, and delivers these events to the appropriate implementations of registered listener logic.

There are two levels to the event-handling mechanism in SWT. At the lowest level, Listeners are registered via the `Display` object with an identifier specifying the type of associated event. When the associated event occurs, the Listener's `handleEvent()` method is called. This system isn't as elegant as the alternative event handling mechanism; however, it's more efficient.

At a higher level, "typed" implementations of `EventListener`s are notified of the occurrence of the event. The classes that are registered to listen for these events implement subinterfaces of `EventListener`. This system is more elegant, granular, and object-oriented, at the expense of being more demanding on the system.

You typically construct a `Display` object with no arguments; you can construct one from a `DeviceData` object, which might be useful for debugging. See Table 3-1 for descriptions of the `Display` constructors.

Table 3-1. `Display` *Constructors*

Constructor	Description
`public Display()`	Creates a new `Display` object and sets the current thread to be the user-interface thread. You'll almost always use either this constructor or `Display.getDefault()` in your application.
`public Display(DeviceData data)`	Creates a new `Display` object, setting the `DeviceData` member of the `Display`. You use `DeviceData` for some lower-level debugging and error configuration.

Display also has several methods, some of which can be profitably ignored (beep(), anyone?). Table 3-2 lists Display's methods.

Table 3-2. Display *Methods*

Method	Description
void addFilter(int eventType, Listener listener)	Adds a listener that's notified when an event of the type specified by eventType occurs.
void addListener(int eventType, Listener listener)	Adds a listener that's notified when an event of the type specified by eventType occurs.
void asyncExec(Runnable runnable)	Gives non-user-interface threads the ability to invoke the protected functions of the SWT widget classes. The user-interface thread performs the code (invokes the run() method) of the runnable at its next "reasonable opportunity." This function returns immediately. See syncExec().
void beep()	Sounds a beep.
void close()	Closes this display.
void disposeExec(Runnable runnable)	Registers a Runnable object whose run() method is invoked when the display is disposed.
static Display findDisplay (Thread thread)	Given a user-interface thread, this function returns the associated Display object. If the given thread isn't a user-interface thread, this method returns null.
Widget findWidget(int handle)	Returns the widget for the specified handle, or null if no such widget exists.
Shell getActiveShell()	Returns the currently active Shell, or null if no shell belonging to the currently running application is active.
Rectangle getBounds()	Returns this display's size and location.
Rectangle getClientArea()	Returns the portion of this display that's capable of displaying data.
static Display getCurrent()	If the currently running thread is a user-interface thread, this thread returns the Display object associated with the thread. If the thread isn't a privileged user-interface thread, this method returns null.
Control getCursorControl()	If the mouse or other pointing device is over a control that's part of the current application, this function returns a reference to the control; otherwise, it returns null.
Point getCursorLocation()	Returns the location of the on-screen pointer relative to the top left corner of the screen.
Point[] getCursorSizes()	Returns the recommended cursor sizes.
Object getData()	Returns the application-specific data set into this display.
Object getData(String key)	Returns the application-specific data for the specified key set into this display.

Table 3-2. `Display` *Methods (continued)*

Method	Description
`static Display getDefault()`	Returns the default display of this application. If one hasn't yet been created, this method creates one and marks the current thread as the user-interface thread. The side effect of becoming the user-interface thread obligates the use of the current thread as the event loop thread for the application.
`int getDismissalAlignment()`	Returns the alignment for the default button in a dialog, either `SWT.LEFT` or `SWT.RIGHT`.
`int getDoubleClickTime()`	Sets the maximum amount of time that can elapse between two mouse clicks for a double-click event to occur.
`Control getFocusControl()`	Returns the control that currently has the focus of the application. If no application control has the focus, returns `null`.
`int getIconDepth()`	Returns the depth of the icons on this display.
`Point[] getIconSizes()`	Returns the recommended icon sizes.
`Monitor[] getMonitors()`	Returns the monitors attached to this display.
`Monitor getPrimaryMonitor()`	Returns the primary monitor for this display.
`Shell[] getShells()`	Returns an array of the active shells (windows) that are associated with this `Display`.
`Thread getSyncThread()`	If the user-interface thread is executing code associated with a `Runnable` object that was registered via the `syncExec()` method, this function will return a reference to the thread that invoked `syncExec()` (the waiting thread). Otherwise, this function returns `null`.
`Color getSystemColor(int id)`	Returns the matching system color as defined in the SWT class. If no color is associated with `id`, this method returns the color black. Remember this is a system color—you shouldn't dispose it when you're finished with it.
`Font getSystemFont()`	Returns a reference to a system font (it shouldn't be disposed), which is appropriate to be used in the current environment. In general, widgets are created with the correct font for the type of component that they represent and you should rarely need to change this value to maintain the correct system appearance.
`Thread getThread()`	Returns the user-interface thread of this `Display`. The thread that created this `Display` is the user-interface thread.
`Point map(Control from, Control to, int x, int y)`	Maps the point specified by `x`, `y` from the `from` control's coordinate system to the `to` control's coordinate system.
`Rectangle map(Control from, Control to, int x, int y, int width, int height)`	Maps the rectangle specified by `x`, `y`, `width`, `height` from the `from` control's coordinate system to the `to` control's coordinate system.

Table 3-2. `Display` *Methods (continued)*

Method	Description
`Point map(Control from, Control to, Point point)`	Maps the specified point from the `from` control's coordinate system to the `to` control's coordinate system.
`Point map(Control from, Control to, Rectangle rectangle)`	Maps the specified rectangle from the `from` control's coordinate system to the `to` control's coordinate system.
`boolean readAndDispatch()`	This is the main event function of the SWT system. It reads events, one at a time, off the windowing system's event queue. After receiving the event, it invokes the appropriate methods on the listener objects that have registered interest in this event. If no events are on the event queue, `readAndDispatch()` executes any requests that might have been registered with this display via `syncExec()` or `asyncExec()`, notifying any `syncExec`ed threads on completion of the request. This method returns `true` if there are more events to be processed, `false` otherwise. Returning `false` allows the calling thread to release CPU resources until there are more events for the system to process via the `sleep()` method.
`void removeFilter(int eventType, Listener listener)`	Removes the specified listener from the notification list for the specified event type.
`void removeListener(int eventType, Listener listener)`	Removes the specified listener from the notification list for the specified event type.
`static void setAppName (String name)`	Sets the application name.
`void setCursorLocation (int x, int y)`	Moves the on-screen pointer to the specified location relative to the top left corner of the screen.
`void setCursorLocation (Point point)`	Moves the on-screen pointer to the specified location relative to the top left corner of the screen.
`void setData(Object data)`	Sets the application-specific data.
`void setData(String key, Object data)`	Sets the application-specific data for the specified key.
`void setSynchronizer (Synchronizer synchronizer)`	Sets the synchronizer for this display.
`boolean sleep()`	Allows the user-interface thread to relinquish its CPU time until it has more events to process or is awakened via another means; for example, `wake()`. This allows the system to process events much more efficiently, as the user-interface thread only consumes CPU resources when it has events to process.
`void syncExec(Runnable runnable)`	Like `asyncExec()`, this method gives non-user-interface threads the ability to invoke the protected functions of the SWT widget classes. The user-interface thread performs this code (invokes the `run` method) of `runnable` at its next "reasonable opportunity." This function returns after the `run` method of the `Runnable` object returns.

Table 3-2. `Display` *Methods (continued)*

Method	Description
`void timerExec(int milliseconds, Runnable runnable)`	Registers a `Runnable` object that the user-interface thread runs after the specified time has elapsed.
`void update()`	Causes all pending paint requests to be processed.
`void wake()`	Wakes up the user-interface thread if it's in `sleep()`. Can be called by any thread.

Although a `Display` object forms the foundation for your GUI, it doesn't present any graphical components to the screen. In fact, the `Display` by itself displays nothing at all. You must create a window, represented by a `Shell` object. This leads us to our next section, which discusses `Shell`s.

Understanding the Shell Object

The `Shell` object represents a window—either a top-level window or a dialog window. It contains the various controls that make up the application: buttons, text boxes, tables, and so on. It has six constructors; two of them aren't recommended for use, and future releases might not support them. Construction follows the SWT pattern of passing a parent and a style (or multiple styles bitwise-ORed together), though some constructors allow default values for either or both parameters. Table 3-3 lists the constructors.

Table 3-3. `Shell` *Constructors*

Constructor	Description
`public Shell()`	Empty constructor, which is equivalent to calling `Shell((Display) null)`. Currently, passing `null` for the `Display` causes the `Shell` to be created on the active display, or, if no display is active, on a "default" display. This constructor is discouraged, and might be removed from a future SWT release.
`public Shell(int style)`	This constructor, too, isn't recommended for use, as it calls `Shell((Display) null, style)`, so also might be removed from SWT.
`public Shell(Display display)`	Constructs a shell using `display` as the display, `null` for the parent, and `SHELL_TRIM` for the style, except on Windows CE, where it uses `NONE` (see Table 3-4).
`public Shell(Display display, int style)`	Constructs a shell using `display` as the display, `null` for the parent, and `style` for the style. See Table 3-4 for appropriate `Shell` styles.
`public Shell(Shell parent)`	Constructs a shell using the parent's `Display` as the display, `parent` for the parent, and `DIALOG_TRIM` for the style, except on Windows CE, where it uses `NONE` (see Table 3-4).
`public Shell(Shell parent, int style)`	Constructs a shell using the parent's `Display` as the display, `parent` for the parent, and `style` for the style. See Table 3-4 for appropriate `Shell` styles.

Internally, all the constructors call a package-visible constructor that sets the display, sets the style bits, sets the parent, and then creates the window. If the `Shell` has a parent, it's a dialog; otherwise, it's a top-level window. Table 3-4 lists the appropriate styles for a `Shell` object; note that all style constants, as you'll see in the next section, are static members of the `SWT` class. Also, realize that the style you set is treated as a hint; if the platform your application is running on doesn't support the style, it's ignored.

Table 3-4. `Shell` *Styles*

Style	Description
`BORDER`	Adds a border.
`CLOSE`	Adds a close button.
`MIN`	Adds a minimize button.
`MAX`	Adds a maximize button.
`NO_TRIM`	Creates a `Shell` that has no border and can't be moved, closed, resized, minimized, or maximized. Not very useful, except perhaps for splash screens.
`RESIZE`	Adds a resizable border.
`TITLE`	Adds a title bar.
`DIALOG_TRIM`	Convenience style, equivalent to `TITLE | CLOSE | BORDER`.
`SHELL_TRIM`	Convenience style, equivalent to `CLOSE | TITLE | MIN | MAX | RESIZE`.
`APPLICATION_MODAL`	Creates a `Shell` that's modal to the application. Note that you should specify only one of `APPLICATION_MODAL`, `PRIMARY_MODAL`, `SYSTEM_MODAL`, or `MODELESS`; you can specify more, but only one is applied. The order of preference is `SYSTEM_MODAL`, `APPLICATION_MODAL`, `PRIMARY_MODAL`, then `MODELESS`.
`PRIMARY_MODAL`	Creates a primary modal `Shell`.
`SYSTEM_MODAL`	Creates a `Shell` that's modal system-wide.
`MODELESS`	Creates a modeless `Shell`.

Most of the time, you won't specify a style when you create a `Shell`, as the default settings usually produce what you want. Feel free to experiment with the styles, though, so you understand what each of them does.

`Shell` inherits a number of methods from its extensive inheritance tree, and adds a few methods of its own (see Table 3-5 for a full listing of `Shell`-specific methods). However, the two methods you'll use most are `open()`, which opens (displays) the `Shell`, and, to a lesser degree, `close()`, which closes the `Shell`. Note that the default operating platforms' methods for closing a `Shell` (for example, clicking the close button on the title bar) are already implemented for you, so you might never need to call `close()`.

Table 3-5. `Shell` *Methods*

Method Name	Description
`void addShellListener (ShellListener listener)`	Adds a listener that's notified when operations are performed on the `Shell`.
`void close()`	Closes the `Shell`.
`void dispose()`	Disposes the `Shell`, and recursively disposes all its children.
`void forceActive()`	Moves the `Shell` to the top of the z-order on its `Display` and forces the window manager to make it active.
`Rectangle getBounds()`	Returns the `Shell`'s size and location relative to its parent (or its `Display` in the case of a top-level `Shell`).
`Display getDisplay()`	Returns the `Display` this `Shell` was created on.
`boolean getEnabled()`	Returns `true` if this `Shell` is enabled, and `false` if not.
`int getImeInputMode()`	Returns this `Shell`'s input-method editor mode, which is the result of bitwise ORing one or more of `SWT.NONE`, `SWT.ROMAN`, `SWT.DBCS`, `SWT.PHONETIC`, `SWT.NATIVE`, and `SWT.ALPHA`.
`Point getLocation()`	Returns the location of this `Shell` relative to its parent (or its `Display` in the case of a top-level `Shell`).
`Region getRegion()`	Returns this `Shell`'s region if it's nonrectangular. Otherwise, returns `null`.
`Shell getShell()`	Returns a reference to itself.
`Shell[] getShells()`	Returns all the `Shell`s that are descendants of this `Shell`.
`Point getSize()`	Returns this `Shell`'s size.
`boolean isEnabled()`	See `getEnabled()`.
`void open()`	Opens (displays) this `Shell`.
`void removeShellListener (ShellListener listener)`	Removes the specified listener from the notification list.
`void setActive()`	Moves the `Shell` to the top of the z-order on its `Display` and asks the window manager to make it active.
`void setEnabled(boolean enabled)`	Passing `true` enables this `Shell`; passing `false` disables it.
`void setImeInputMode (int mode)`	Sets this `Shell`'s input-method editor mode, which should be the result of bitwise ORing one or more of `SWT.NONE`, `SWT.ROMAN`, `SWT.DBCS`, `SWT.PHONETIC`, `SWT.NATIVE`, and `SWT.ALPHA`.
`void setRegion(Region region)`	Sets the region for this `Shell`. Use for nonrectangular windows.
`void setVisible(boolean visible)`	Passing `true` sets this `Shell` visible; passing `false` sets it invisible.

The SWT Class—Constants and Methods

The SWT class contains a repository of class-level constants and methods to simplify SWT programming.

Curiously, nothing prevents you from creating an SWT object, though no harm is done by creating one. The SWT class derives from java.lang.Object and has no constructors defined so that the default constructor can be invoked. However, an SWT object has no state beyond what it inherits from java.lang.Object, and is essentially useless.

The SWT class provides a few convenience methods, all of which, as mentioned earlier, are static. Most applications will have no need to use these; they're listed in Table 3-6.

Table 3-6. SWT *Methods*

Method Name	Description
static void error(int code)	Throws an exception based on code. It's the same as calling static void error(int code, (Throwable) null).
static void error(int code, Throwable throwable)	Throws an exception based on code. throwable should either be null or the Throwable that caused SWT to throw an exception. code is one of the error constants defined in SWT.
static String getMessage (String key)	Gets the appropriate National Language Support (NLS) message as a String for key. See java.util.ResourceBundle for more information on NLS. The resource bundle is found in org.eclipse.swt.internal.SWTMessages .properties; see Table 3-7 for the supported keys and corresponding messages.
static String getPlatform()	Gets the SWT platform name (for example, "win32," "gtk," "carbon").
static int getVersion()	Gets the SWT library version number.

Enter this code on Windows XP and Eclipse 2.1.1:

```
System.out.println("Platform: " + SWT.getPlatform());
System.out.println("Version: " + SWT.getVersion());
```

The code prints:

```
Platform: win32
Version: 2135
```

Table 3-7. SWT Message Keys and Values

Key	Value
SWT_Yes	Yes
SWT_No	No
SWT_OK	OK
SWT_Cancel	Cancel
SWT_Abort	Abort
SWT_Retry	Retry
SWT_Ignore	Ignore
SWT_Sample	Sample
SWT_A_Sample_Text	A Sample Text
SWT_Selection	Selection
SWT_Current_Selection	Current Selection
SWT_Font	Font
SWT_Color	Color
SWT_Extended_style	Extended style
SWT_Size	Size
SWT_Style	Style
SWT_Save	Save
SWT_Character_set	Character set
SWT_ColorDialog_Title	Colors
SWT_FontDialog_Title	Fonts
SWT_Charset_Western	Western
SWT_Charset_EastEuropean	East European
SWT_Charset_SouthEuropean	South European
SWT_Charset_NorthEuropean	North European
SWT_Charset_Cyrillic	Cyrillic
SWT_Charset_Arabic	Arabic
SWT_Charset_Greek	Greek
SWT_Charset_Hebrew	Hebrew
SWT_Charset_Turkish	Turkish
SWT_Charset_Nordic	Nordic
SWT_Charset_Thai	Thai
SWT_Charset_BalticRim	Baltic Rim
SWT_Charset_Celtic	Celtic
SWT_Charset_Euro	Euro
SWT_Charset_Romanian	Romanian
SWT_Charset_SimplifiedChinese	Simplified Chinese
SWT_Charset_TraditionalChinese	Traditional Chinese
SWT_Charset_Japanese	Japanese

Table 3-7. SWT Message Keys and Values (continued)

Key	Value
SWT_Charset_Korean	Korean
SWT_Charset_Unicode	Unicode
SWT_Charset_ASCII	ASCII
SWT_InputMethods	Input Methods

Summary

An SWT-based program connects to the underlying windowing system through its `Display` object. Windows, widgets, and events are built upon and travel through this crucial object. The windows you create in your applications are all `Shell`s. This chapter built your obligatory "Hello, World" SWT program and explained the design behind SWT. You now know how to create widgets with parents, to clean up after yourselves, and not to touch things that don't belong to you.

In the next chapter, you learn how to place your widgets on windows where you want them.