CHAPTER 1

Widget Fundamentals

This chapter provides an overview of the classes contained in the packages *org.eclipse.swt.widgets* and *org.eclipse.swt.events*. We begin by defining what a widget is, then cover the fundamental relationships between widgets, and finally cover how widgets interrelate with each other and the user.

1.1 What Is a Widget?

A widget is a graphical user interface element responsible for interacting with the user. Widgets maintain and draw their state using some combination of graphical drawing operations. Using the mouse or the keyboard, the user can change the state of a widget. When a state change occurs, whether initiated by the user or the application code, widgets redraw to show the new state. This is an important distinguishing feature that all widgets share. It means that when you set a property on a widget, you are not responsible for telling the widget to redraw to reflect the change.

1.1.1 Life Cycle

Widgets have a life cycle. They are created by the programmer and disposed of when no longer needed. Because the widget life cycle is so fundamental to the understanding of the Standard Widget Toolkit (SWT), we are going to cover it in detail here.

1.1.2 Creating a Widget

Widgets are created using their constructor, just like any other Java object. Some widget toolkits employ the *factory* pattern to instantiate their widgets. For simplicity, SWT does not.

When a widget is instantiated, operating system resources are acquired by the widget. This simplifies the implementation of SWT, allowing most of the widget state to reside in the operating system, thus improving performance and reducing memory footprint.[1] There is another important benefit of acquiring operating system resources in the constructor. It gives a clear indication when resources have been allocated. We will see that this is critical in the discussion of widget destruction (see Disposing of a Widget).

Finally, constructors take arguments that generally cannot be changed after the widget has been created. Note that these arguments are create-only **from the point of view of the operating system** and must be present when the widget is created.

Standard Constructors

Widget is an abstract class, so you will never create a Widget instance. In the discussion that follows, note that references to the class Widget actually apply to the subclasses of Widget. This is because subclasses of Widget share the same constructor signatures, giving widget creation a strong measure of consistency, despite the different kinds of widgets and their implementation.

There are four general forms of widget constructor implemented by the subclasses of the class Widget.

1. Widget ()
2. Widget (Widget parent)
3. Widget (Widget parent, int style)
4. Widget (Widget parent, int style, int index)

1. If this were not the case, fields would be needed to keep track of values not yet stored in the operating system. When the operating system widget was created, state would need to be copied back and forth between SWT and the operating system, slowing things down and introducing potential inconsistencies. The same application code operating on a widget that has been created in the operating system must *not* behave differently when that widget has yet to be created. Although different behavior can be characterized as a bug, the potential for these kinds of problems is large, especially when multiplied by the number of platforms.

The concept of *hierarchy* (see Widget Hierarchy) is very important in SWT, so much so that the parent widget is the first parameter in most widget constructors.[2] The following sections describe each of the parameters in detail.

A Word About Parameters, Exceptions, and Error Checking

In SWT, methods that take parameters that are Objects check for null and throw IllegalArgumentException ("Argument cannot be null") when the argument can't be null. Besides being more informative, checking for null helps to ensure consistent behavior between different SWT implementations. Barring unforeseen circumstances, such as catastrophic virtual machine failure, an SWT method will throw only three possible exceptions and errors: IllegalArgumentException, SWTException, and SWTError. Anything else is considered to be a bug in the SWT implementation.

The Parent Parameter

Widgets cannot exist without a parent, and the parent cannot be changed after a widget is created.[3] This is why the parent is present in almost every constructor. The type of parent depends on the particular widget. For example, the parent of a menu item must be a menu and cannot be a text editor. Strong typing in the constructor enforces this rule. Code that attempts to create a menu item with a text editor parent does not compile, making this kind of programming error impossible.

It is also possible to query the parent of a widget using getParent() but this method is not found in the class Widget.

Why Is getParent() Not Implemented in Widget?

We could have implemented getParent() in class Widget but the method would need to return a Widget. This would require the programmer to cast the result

2. Note that the no-parameter version of the constructor is very rare and occurs only as a convenience constructor in the class Shell, so it will not be discussed here.
3. For an exception to this rule, see Control.setParent(), which allows you to change the parent on *some* platforms.

to the appropriate type, despite the fact that the correct type was provided in the constructor. By implementing getParent() in each subclass, the type information that was specified when the widget was created is preserved. One of the design goals of SWT is to preserve as much type information as possible in the API, reducing the need for application programs to cast.

The Style Parameter

Styles are integer bit values used to configure the behavior and appearance of widgets. They specify create-only attributes, such as choosing between multiple- and single-line editing capability in a text widget. Because these attributes cannot be changed after creation, the style of a widget cannot be changed after it has been created. Style bits provide a compact and efficient method to describe these attributes.

All styles are defined as constants in the class org.eclipse.swt.SWT.

Class SWT

SWT uses a single class named (appropriately) *SWT* to share the constants that define the common names and concepts found in the toolkit. This minimizes the number of classes, names, and constants that application programmers need to remember. The constants are all found in one place.

As expected, you can combine styles by using a bitwise OR operation. For example, the following code fragment creates a multiline text widget that has a border and horizontal and vertical scroll bars.

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

The list of the style constants that are applicable to each widget is described in the Javadoc for the widget. Styles that are defined in a given superclass are valid for the subclasses unless otherwise noted. The constant SWT.NONE is used when there are no applicable style bits.

The widget style can be queried after it has been created using getStyle().

getStyle() Returns the actual style of the widget represented using a bitwise OR of the constants from class SWT. Note that this can be different from the value that was passed to the constructor because it can

include defaults provided by the widget implementation. In addition, if a style requested in the constructor cannot be honored, the value returned by getStyle() will not contain the bits. This can happen when a platform does not support a particular style.

The following code fragment uses a bitwise AND to test to see whether a text widget displays and can edit only a single line of text.

```
if ((text.getStyle () & SWT.SINGLE) != 0) {
    System.out.println ("Single Line Text");
}
```

The Position Parameter

The position parameter allows you to create a widget at a specific index in the list of children or by the parent.[4] The other children in the list are shifted to make room for the new widget. For example, the position parameter could be used to create a menu item and make it the third item in a menu. By default, if the position parameter is not provided, the child is placed at the end of the list.

Why is there no widget "add()" method to add a child to the children list of its parent? For an add() method to do something reasonable, it would require that you be able to remove a widget from the children list without destroying it. Given that a widget cannot exist without a parent, this would leave the child in a state where it knows about its parent but the parent does not know about the child.

Convenience Constructors—Just Say No

Some programmers demand convenience constructors using arguments such as, *"Every time a button is created, I always set the text so there should be a button constructor that takes a string."* Although it is tempting to add convenience constructors, there is just no end to them. Buttons can have images. They can be checked, disabled, and hidden. It is tempting to provide convenience constructors for these properties, as well. When a new API is defined, even more convenience constructors are needed. To minimize the size of the widget library and provide consistency, SWT does not normally provide convenience constructors.

4. The position parameter is the natural expression of the rule that widgets cannot exist without a parent. Because the parent-child relationship is specified in the constructor, the child will be placed in the list of children when it is created.

1.1.3 Disposing of a Widget

When a widget is no longer needed, its dispose() method must be explicitly called.

> **dispose**() Hides the widget and its children, and releases all associated operating system resources. In addition, it removes the widget from the children list of its parent. All references to other objects in the widget are set to null, facilitating garbage collection.[5]

SWT does *not* have a widget remove() method for the same reason that there is no add() method: It would leave the child in a state where it knows about its parent but the parent does not know about the child. Because widgets are alive for exactly the duration that they are referenced by their parents, implicit finalization (as provided by the garbage collector) does not make sense for widgets. Widgets are not finalized.[6]

Accessing a widget after it has been disposed of is an error and causes an SWTException ("Widget is disposed") to be thrown. The only method that is valid on a widget that has been disposed of is:

> **isDisposed**() Returns *true* when the widget has been disposed of. Otherwise, returns *false*.

If you never dispose of a widget, eventually the operating system will run out of resources. In practice, it is hard to write code that does this. Programmers generally do not lose track of their widgets because they require them to present information to the user. Users generally control the life cycle of top-level windows—and the widgets they contain—by starting applications and clicking on "close boxes."

When a widget is disposed of, a dispose event is sent, and registered listeners are invoked in response. For more on this, see the section Events and Listeners.

---

5. Of course, your application has to have no references to the objects for them to be garbage collected.
6. This is covered in great detail in the article *SWT: The Standard Widget Toolkit—PART 2: Managing Operating System Resources* at Eclipse.org (see http://www.eclipse.org/articles/swt-design-2/swt-design-2.html).

1.1.4 Rules for Disposing of Widgets

There are only two rules that you need to know to determine when to dispose of a particular widget. Please excuse the references to specific classes and methods that have yet to be discussed. They will be described in detail later in the book. It is more important at this time that the "rules" are complete.

Rule 1:

If you created it, you dispose of it. SWT ensures that all operating system resources are acquired when the widget is created. As we have already seen, this happens in the constructor for the widget. What this means is that you are responsible for calling dispose() on SWT objects that you created using new. SWT will never create an object that needs to be disposed of by the programmer outside of a constructor.

Rule 2:

Disposing a parent disposes the children. Disposing of a top-level shell will dispose of its children. Disposing of a menu will dispose of its menu items. Disposing of a tree widget will dispose of the items in the tree. This is universal.

There are two extensions to Rule 2. These are places where a relationship exists that is not a parent-child relationship but where it also makes sense to dispose of a widget.

Rule 2a:

Disposing a MenuItem disposes the cascade Menu.

> **MenuItem.setMenu**() Disposing of a MenuItem that has a submenu set with the setMenu() method disposes of the submenu. This is a natural extension of Rule 2. It would be a burden to the programmer to dispose of each individual submenu. It is also common behavior in most operating systems to do this automatically.[7]

Rule 2b:

Disposing a control disposes the pop-up Menu.

7. Windows and X/Motif dispose submenus when a cascading menu item is disposed. GTK and the Macintosh employ a reference-counting scheme that normally disposes of the menu, provided that there are no more references.

Control.setMenu() Disposing of a control that has a pop-up menu assigned using the setMenu() method disposes the pop-up menu. Many application programmers expected this behavior, even though the operating systems do not do it automatically. We added this rule because too many application programs temporarily leaked pop-up menus.[8]

Another way to remember the extensions to Rule 2 is to notice that both extensions concern instances of the class Menu when used with the setMenu() method. For more information about menus, see Classes Menu and MenuItem in the ToolBars and Menus chapter.

1.2 Widget Hierarchy

The class Widget is the root of an *inheritance* hierarchy of diverse user interface elements such as buttons, lists, trees, and menus. Subclasses, both within the widgets package and outside it, extend the basic behavior of widgets by implementing events and adding API that is specific to the subclass. The inheritance hierarchy of the class Widget is static. For example, the class Button will always inherit from its superclass, Control.[9] Instances of Button will always respond to messages that are implemented in the class Control.

The *containment* hierarchy of a widget is defined by the parent/child relationship between widgets. It is built dynamically at runtime. For example, the parent of an instance of Button might be a group box or tab folder but instances of the class Button do not inherit from the classes that represent these widgets. As we have already seen, the containment hierarchy of a widget is defined when the widget is created, using the parent argument of the constructor.

1.2.1 Subclassing in SWT

Generally speaking, subclassing is not the safest way to extend a class in an object-oriented language, due to the *fragile superclass* problem.

The term *fragile superclass* comes from the C++ programming world. It normally refers to the fact that when a new method or field is added to a superclass, subclasses need to be recompiled or they might corrupt memory. Java solves the *static* or binary compatibility portion of the problem using a

8. The leak was only temporary because the pop-up menu was eventually disposed of when the top-level Shell was closed.

9. Controls are discussed in the section Controls, Composites, Shells, and the Display. Although we tend to use the terms widget and control somewhat interchangeably, there are differences. See the section Widgets That Are Not Controls for more on this.

name-lookup mechanism that is transparent to the programmer. However, there is also a *dynamic* portion of the problem where subclasses can inadvertently depend on the implementation of a superclass. For example, a subclass may depend on the fact that the internal implementation of the superclass calls a certain public method that is reimplemented in the subclass. Should the superclass be changed to no longer call this method, the subclass will behave differently and might be subtly broken. In SWT, where the implementation of most classes differs between platforms, the chances of this happening are increased. For this reason, subclassing in arbitrary places in the Widget class hierarchy is discouraged *by the implementation*.

In order to allow subclassing where it is normally disallowed, the Widget method checkSubclass() must be redefined.

checkSubclass() Throws an SWTException("Subclassing not allowed") when the instance of the class is not an allowed subclass.

The protected method checkSubclass() is called internally by SWT when an instance of a widget is created. Subclasses can override this method to avoid the check and allow the instance to be created. The following code fragment defines an inner class that is a subclass of the class Label and reimplements the setText() method.

```
Label label = new Label(shell, SWT.NONE) {
    protected void checkSubclass() {
    }
    public void setText(String string) {
        System.out.println("Setting the string");
        super.setText(string);
    }
};
```

Why Aren't the Widget Classes "final"?

Java allows the programmer to tag a class as *final*, disallowing subclasses. In fact, in very early versions of SWT, classes that should not be subclassed were clearly marked as such using the *final* keyword. Unfortunately, this proved to be too inflexible. In particular, it meant that if a problem was found in an SWT class, only the SWT team could fix it. There was no way to temporarily "patch" the class by creating a subclass to override the problem method(s). Customers who needed to ship their product before a fix could be integrated into the next SWT release were willing to risk the dynamic fragile superclass problem in order to

have the freedom to make this kind of patch. The checkSubclass() method is a compromise that allows them to do this without removing all constraints on subclassing.

It is important to note that this really is the only reason why the checkSubclass() method was added. Well-written SWT programs should *never* override checkSubclass().

In SWT, user interfaces are constructed by composition of widget instances. Event listeners (see Events and Listeners, below) are added to widgets to run application code when an event occurs, rather than overriding a method. Application programmers use listeners instead of subclassing to implement the code that reacts to changes in the user interface.

Subclassing *is* allowed in SWT but only at very controlled points, most notably, the classes that are used when implementing a custom widget: Composite or its subclass Canvas. To indicate this, the checkSubclass() method in Composite does not constrain the allowable subclasses. To create a new kind of widget in SWT, you would typically subclass Canvas, then implement and use event listeners to give it the required appearance and behavior. Note that you should still not reference internal details of the superclasses, because they may vary significantly between platforms and between subsequent versions of SWT. For an example of creating a custom widget, see MineSweeper in the Applications part of the book.[10]

1.2.2 Controls, Composites, Shells, and the Display

A *control* is a user interface element that is contained somewhere within a top-level window, called a *Shell*. Controls are common in all user interfaces. Buttons, labels, progress bars, and tables are all controls. Users are familiar with the standard set of controls that come with the operating system. In SWT, operating system controls are, by definition, instances of subclasses of the abstract class Control.

Taking a bottom-up view of the world, every control has a parent that is an instance of the class Composite or one of its subclasses. The class Shell, which represents the top-level windows of your application, is a subclass of Composite. Shells are created on a Display, which represents the "screen."

10. A good article that describes how to create custom widgets, entitled *Creating Your Own Widgets Using SWT*, can be found in the "articles" area at www.eclipse.org.

Stated another way, this time from the top down, a display contains a list of top-level shells, where each shell is the root of a tree composed of composites and controls. Composites can contain other composites, allowing the tree to have arbitrary depth. If the child of a shell is another shell, the child is commonly called a *dialog* shell. A dialog shell always stays in front of the parent shell.

1.2.3 Widgets That Are Not Controls

Unfortunately, the picture is not quite that simple. Some of the user interface elements that make up an SWT application are not represented by controls. In general, these are objects that are interesting enough to warrant being represented by instances of some class but do not have the operating system resource requirements of controls. For example, operating systems do not represent each of the items in a tree using other controls. Instead, to improve performance, the tree is responsible for drawing the items. Because the items no longer behave like controls, it would be a mistake to model them as such. The set of widgets that are not controls are exactly those that are not modeled as controls on all operating systems.[11]

1.3 Events and Listeners

An *event* is simply an indication that something interesting has happened. Events, such as "mouse down" and "key press," are issued when the user interacts with a widget through the mouse or keyboard. Event classes, used to represent the event, contain detailed information about what has happened. For example, when the user selects an item in a list, an event is created, capturing the fact that a "selection" occurred. The event is delivered to the application via a listener.

A *listener* is an instance of a class that implements one or more agreed-upon methods whose signatures are captured by an interface. Listener methods always take an instance of an event class as an argument. When something interesting occurs in a widget, the listener method is invoked with an appropriate event.

Most widgets track sequences of events and redraw based on them, sometimes issuing a higher-level event to indicate a state change. For example, a

11. Restrictions in Windows, for example, prevented us from making menus and scroll bars into controls.

button may track "mouse press," "move," and "release" in order to issue a "selection" event when the mouse is released inside the button.

Some widget methods generate events. For example, when you call setFocus() on a control, "focus" events are issued. The rules governing which methods cause events and which do not are largely historical, based on the behavior of the Windows and X/Motif operating systems. In order to be able to write portable applications, these rules have been standardized across platforms.

SWT has two kinds of listeners: untyped and typed.

1.3.1 Untyped Listeners

Untyped listeners provide a simple, generic, low-level mechanism to handle any type of event. There are only two Java types involved: a single generic interface called *Listener* and a single event class called *Event*. Untyped listeners are added using the method addListener().

> **addListener(int event, Listener listener)** Adds the listener to the collection of listeners that will be notified when an event of the given type occurs. When the event occurs in the widget, the listener is notified by calling its handleEvent() method.

The type argument specifies the event you are interested in receiving. To help distinguish them from all the other SWT constants, type arguments are mixed upper- and lowercase by convention. All other constants in SWT are uppercase. The following code fragment uses addListener() to add a listener for SWT.Dispose.

```
widget.addListener(SWT.Dispose, new Listener() {
    public void handleEvent(Event event) {
        // widget was disposed
    }
});
```

When multiple listeners are added, they are called in the order they were added. This gives the first listener the opportunity to process the event and possibly filter the data before the remaining listeners are notified (see Event Filters in the chapter Display). Adding the same instance of a listener multiple times is supported, causing it to be invoked once for each time it was added.[12]

12. We can't think of any reason you would want to add the same listener multiple times but there is no restriction to stop you from doing this.

It is also possible to remove listeners using removeListener().

removeListener(int type, Listener listener) Removes the listener from the collection of listeners that will be notified when an event of the given type occurs.

In order for a listener to be removed, you must supply the *exact* instance of the listener that was added. If the same listener instance is added multiple times, it must be removed the same number of times it was added to remove it from the listener collection completely.

Generally speaking, removing listeners is unnecessary. Listeners are garbage collected when a control is disposed of, provided that there are no other references to the listener in the application program.

Application code can send events using notifyListeners().

notifyListeners(int type, Event event) Sets the type of the event to the given type and calls Listener.handleEvent() for each listener in the collection of listeners.

An important point to note is that notifyListeners() does not cause the corresponding operating system event to occur. For example, calling notifyListeners() with SWT.MouseDown on a button will not cause the button to appear to be pressed. Also, notifyListeners() does not ensure that the appropriate fields for the event have been correctly initialized for the given type of event. You can use notifyListeners() to invoke listeners that you define but it is probably easier simply to put the code in a helper method.

Class Event has a number of fields that are applicable only for a subset of the event types. These fields are discussed as each type of event is described. Table 1.1 shows the fields that are valid for all event types.

Table 1.1 Public Fields in Class Event That Are Applicable to All Untyped Events

Field	*Description*
display	the Display on which the event occurred
widget	the Widget that issued the event
type	the event type

Table 1.2 shows the type constants that describe all of the untyped events that SWT implements. More details are available in the descriptions of the individual widgets.

Table 1.2 Untyped Events

Event Type Constant	*Description*
SWT.KeyDown	A key was pressed
SWT.KeyUp	A key was released
SWT.MouseDown	A mouse button was pressed
SWT.MouseUp	A mouse button was released
SWT.MouseMove	The mouse was moved
SWT.MouseEnter	The mouse entered the client area of the control
SWT.MouseHover	The mouse lingered over a control
SWT.MouseExit	The mouse exited the client area of the control
SWT.MouseDoubleClick	A mouse button was pressed twice
SWT.Paint	A control was asked to draw
SWT.Move	The position of the control was changed
SWT.Resize	The size of the client area of the control changed
SWT.Dispose	The widget was disposed
SWT.Selection	A selection occurred in the widget
SWT.DefaultSelection	The default selection occurred in the widget
SWT.FocusIn	Keyboard focus was given to the control
SWT.FocusOut	The control lost keyboard focus
SWT.Expand	A tree item was expanded
SWT.Collapse	A tree item was collapsed
SWT.Iconify	The shell was minimized
SWT.Deiconify	The shell is no longer minimized
SWT.Close	The shell is being closed
SWT.Show	The widget is becoming visible
SWT.Hide	The widget is being hidden
SWT.Modify	Text has changed in the control
SWT.Verify	Text is to be validated in the control

Table 1.2 Untyped Events (continued)

Event Type Constant	*Description*
SWT.Activate	The control is being activated
SWT.Deactivate	The control is being deactivated
SWT.Help	The user requested help for the widget
SWT.DragDetect	A drag-and-drop user action occurred
SWT.MenuDetect	The user requested a context menu
SWT.Arm	The menu item is drawn in the armed state
SWT.Traverse	A keyboard navigation event occurred
SWT.HardKeyDown	A hardware button was pressed (handhelds)
SWT.HardKeyUp	A hardware button was released (handhelds)

If you are writing your own widget, you may want to use notifyListeners() to support the built-in untyped events in SWT since this allows your widget to behave like the native widgets with respect to the mapping between untyped and typed listeners. However, for events that are particular to your widget, you will also typically implement typed listeners.

1.3.2 Typed Listeners

A *typed listener* follows the standard JavaBeans listener pattern. Typed listeners and their corresponding event classes are found in the package *org.eclipse.swt.events*. For example, to listen for a dispose event on a widget, application code would use addDisposeListener().

> **addDisposeListener(DisposeListener listener)** Adds the listener to the collection of listeners that will be notified when a widget is disposed. When the widget is disposed, the listener is notified by calling its widgetDisposed() method.

The following code fragment listens for a dispose event on a widget.

```
widget.addDisposeListener(new DisposeListener() {
    public void widgetDisposed(DisposeEvent event) {
        // widget was disposed
    }
});
```

DisposeListener is an interface. If there is more than one method defined by the listener, SWT provides an adapter class that contains no-op implementations of the methods.[13] For example, the interface SelectionListener has two methods, widgetSelected() and widgetDefaultSelected(), that take SelectionEvents as arguments. As a result, the class SelectionAdapter is provided that provides no-op implementations for each method.

Typed listeners are removed using the corresponding remove method for the listener. For example, a listener for a dispose event is removed using removeDisposeListener().

removeDisposeListener(DisposeListener listener) Removes the listener from the collection of listeners that will be notified when the widget is disposed.

Table 1.3 shows all of the typed events that SWT implements. These are described in more detail in the descriptions of the individual widgets.

Table 1.3 Typed Events

Event	*Listener*	*Methods*	*Untyped Event*
ArmEvent	ArmListener	widgetArmed(ArmEvent)	SWT.Arm
ControlEvent	ControlListener (and ControlAdapter)	controlMoved(ControlEvent) controlResized(ControlEvent)	SWT.Move SWT.Resize
DisposeEvent	DisposeListener	widgetDisposed(DisposeEvent)	SWT.Dispose
FocusEvent	FocusListener (and FocusAdapter)	focusGained(FocusEvent) focusLost(FocusEvent)	SWT.FocusIn SWT.FocusOut
HelpEvent	HelpListener	helpRequested(HelpEvent)	SWT.Help
KeyEvent	KeyListener (and KeyAdapter)	keyPressed(KeyEvent) keyReleased(KeyEvent)	SWT.KeyPressed SWT.KeyReleased
MenuEvent	MenuListener (and MenuAdapter)	menuHidden(MenuEvent) menuShown(MenuEvent)	SWT.Hide SWT.Show
ModifyEvent	ModifyListener	modifyText(ModifyEvent)	SWT.Modify
MouseEvent	MouseListener (and MouseAdapter)	mouseDoubleClick(MouseEvent) mouseDown(MouseEvent) mouseUp(MouseEvent)	SWT.MouseDoubleClick SWT.MouseDown SWT.MouseUp
MouseEvent	MouseMoveListener	mouseMove(MouseEvent)	SWT.MouseMove

13. Adapter classes are provided for convenience and to adhere to JavaBeans listener conventions.

Table 1.3 Typed Events (continued)

Event	*Listener*	*Methods*	*Untyped Event*
MouseEvent	MouseTrackListener (and MouseTrackAdapter)	mouseEnter(MouseEvent) mouseExit(MouseEvent) mouseHover(MouseEvent)	SWT.MouseEnter SWT.MouseExit SWT.MouseHover
PaintEvent	PaintListener	paintControl(PaintEvent)	SWT.Paint
SelectionEvent	SelectionListener (and SelectionAdapter)	widgetDefaultSelected(SelectionEvent) widgetSelected(SelectionEvent)	SWT.DefaultSelection SWT.Selection
ShellEvent	ShellListener (and ShellAdapter)	shellActivated(ShellEvent) shellClosed(ShellEvent) shellDeactivated(ShellEvent) shellDeiconified(ShellEvent) shellIconified(ShellEvent)	SWT.Activate SWT.Close SWT.Deactivate SWT.Deiconify SWT.Iconify
TraverseEvent	TraverseListener	keyTraversed(TraverseEvent)	SWT.Traverse
TreeEvent	TreeListener (and TreeAdapter)	treeCollapsed(TreeEvent) treeExpanded(TreeEvent)	SWT.Collapse SWT.Expand
VerifyEvent	VerifyListener	verifyText(VerifyEvent)	SWT.Verify

1.3.3 Why Are There Two Listener Mechanisms?

In early versions of SWT, there were only untyped listeners. After considerable discussion between the Eclipse implementers, the SWT user community, and the developers, it was decided to include a more "JavaBeans-like" listener mechanism. It was felt that this would ease the transition to SWT for developers who were already familiar with AWT/Swing. The untyped listeners remain as the implementation mechanism for event handling in SWT. The typed listeners are defined in terms of them.

We recommend that SWT applications always be implemented in terms of the typed listener mechanism, although this is largely based on the more familiar pattern they represent. Because of the simplicity of the untyped mechanism and our closeness to the implementation, we tend to use it in many of the small examples we write. To see a clear example of the typed mechanism in action, take a look at the FileExplorer example in the Applications part of the book.

Effectively, the trade-off between the two listener models is one of space versus ease of use and adherence to a standard pattern. Using untyped listeners, it is possible to minimize the number of classes and methods used to listen for events. The same listener can be used to listen for many different event types. For example, the following code fragment listens for dispose as well as mouse events.

```
Listener listener = new Listener() {
    public void handleEvent(Event event) {
        switch (event.type) {
            case SWT.Dispose: break;
            case SWT.MouseDown: break;
            case SWT.MouseUp: break;
            case SWT.MouseMove: break;
        }
        System.out.println("Something happened.");
    }
};
shell.addListener(SWT.Dispose, listener);
shell.addListener(SWT.MouseDown, listener);
shell.addListener(SWT.MouseUp, listener);
shell.addListener(SWT.MouseMove, listener);
```

In practice, unless space usage is the overwhelming constraint, we expect that most programmers will use the typed listener mechanism.[14] Note that typed events have very specific listener APIs, whereas the untyped events have only handleEvent(). Using untyped events can lead to switch logic that is more complicated. Refer to Table 1.3 to see the mapping between typed and untyped events.

1.3.4 Widget Events

SWT.Dispose (DisposeEvent)

Table 1.4 shows the dispose events that are provided by SWT.

Table 1.4 Dispose Events

Untyped Event	*Description*	
SWT.Dispose	The widget was disposed	
Typed Event	***Listener***	***Methods***
DisposeEvent	DisposeListener	widgetDisposed(DisposeEvent)

14. All of the typed listeners provided by the native widgets in SWT are found in the package *org.eclipse.swt.events*. When space is extremely limited, this package and all of the corresponding addXXXListener() and removeXXXListener() methods in *org.eclipse.swt.widgets* can be safely removed.

The SWT.Dispose event (typed event DisposeEvent) is sent when a widget is disposed of. Dispose events contain meaningful values in only the *display*, *widget,* and *type* fields.

1.4 Application Data

Using application data, you can associate arbitrary *named* and *unnamed* data with a widget. In some situations, this can be a useful alternative to subclassing. Given that subclassing at arbitrary places in the Widget hierarchy is strongly discouraged, if you are subclassing in order to add fields to a widget, application data can be used instead.

The setData() method is used to set application data. To allow any object to be associated with a widget, setData() and getData() accept and return objects.

setData(Object data) Sets the unnamed application data. The *data* parameter is associated with the widget and stored in a single unnamed field. This field is for use by the application.

getData() Answers the unnamed application data or null if none has been set.

setData(String key, Object data) Sets the named application data. The *data* is stored as a "key/value" pair in the widget. These key/value pairs are for use by the application. The "value" can be any object. If the "key" is null, an IllegalArgumentException ("Argument cannot be null") is thrown.

getData(String key) Answers the named application data. The *key* is used to find the key/value pair. If no such pair exists, null is returned.

The following code fragment associates an unnamed object (the string "Picard") with a widget. The string is then retrieved and the message "Found the captain!" is printed.

```
widget.setData("Picard");
if ("Picard".equals(widget.getData())) {
    System.out.println("Found the captain!");
}
```

The following code fragment associates the key "Android" with the object "Data" and the key "Captain" with the object "Picard". The key "Captain"

is used to find the object “Picard”, and the message “Found the captain again!” is printed.

```
widget.setData("Android", "Data");
widget.setData("Captain", "Picard");
if ("Picard".equals(widget.getData("Captain"))) {
    System.out.println("Found the captain again!");
}
```

Application data is optimized for space over speed. This means that storing keys and values is memory-efficient, with the result that adding, removing, and retrieving values is somewhat slower. In general, this is a good design trade-off, because most applications tend to store little or no data with their widgets. When data is stored with a widget, it is often a single named or unnamed value. When storing thousands of named values, if you experience performance problems, store a hash table or another data structure as data with the widget and look the values up in the table instead of in the widget.

1.5 Querying the Display

Widgets provide the method getDisplay() that is used to query the Display.

getDisplay() Answers the display where the widget was created.

Display is such an important class in SWT that every widget knows on which Display it was created. Display is described in full detail in the chapter Display.

1.6 Summary

In this chapter, you have learned the fundamental behavior of widgets. Widgets are created using standard constructor patterns and explicitly disposed of when no longer needed. User interfaces are built in SWT by the composing instances of widgets to form containment hierarchies. Rather than subclassing and overriding methods, Listeners are used to notify an application when something interesting has happened. Methods of the class Widget were described as they relate to events, application data, and the Display.