In this chapter, we look at the fundamental classes used for implementing a graphical user interface. In Java, these classes are provided primarily in a user interface library known as Swing. Swing is an enhancement of a library called the Abstract Window Toolkit, or AWT. The AWT has been available as long as the language itself and has undergone substantial evolution since its original design. Swing is an extension of the AWT, intended to provide the functionality and flexibility necessary to develop substantial commercial applications.

The Java user interface libraries are large and complex, containing several hundred classes and interfaces. This complexity is probably unavoidable in any set of tools capable of supporting the degree of customization required in building sophisticated, portable, extensible graphical user interfaces. Furthermore, the Swing components are built as an extension of previous AWT library elements. The designers likely had in mind maintaining some degree of compatibility between Swing and existing applications built with the AWT and providing a relatively straightforward path for upgrading applications to Swing. Maintaining compatibility between versions of a software system almost inevitably leads to increased complexity in the later versions.

A comprehensive discussion of the Swing components and their use would require thousands of pages and is clearly beyond the scope of this text. Our intention is not to develop proficiency in designing user interfaces with Swing. Rather, we want to understand some of the more fundamental concepts and take a brief look at a few of the most central classes. This will serve our purpose, which is to see how an event-driven, window-based application is structured. Furthermore, as our goal in this chapter is to understand the user interface, we will keep the model component of our examples as simple as possible.

Swing will also serve as a case study of how a large library is organized. However, many Swing design decisions were influenced both by the existing AWT libraries and by the need to provide a degree of flexibility well beyond the scope of this discussion. As a result, we cannot always offer a convincing justification for the underlying structures of the user interface libraries. Fortunately, there are several good reference texts available to the reader who needs detailed information regarding Swing and its use.
Chapter 17 Building a graphical user interface

Objectives

After studying this chapter you should understand the following:

• the idea of an event driven interface;
• Java’s Swing component and container structure;
• the use of layout managers to manage component positioning and sizing;
• handling events occurring in a component.

Also, you should be able to:

• build a simple display from graphical components;
• write event handlers that listen for high-level events;
• build a simple graphical user interface for an application.

17.1 Event-driven interfaces

We have seen that the interface and the model are two of the principal components of a software system. The model is the basic problem representation and problem-solving component; the interface handles interaction with the external world. When the external world is a person, we refer to the interface as a user interface. We have concentrated previously on design and implementation of the model. Now we take a closer look at the user interface.

There are two fundamental patterns describing how an application interacts with its environment, that we’ll call algorithm-driven and event-driven. An application’s structure substantially depends on the approach adopted. In an algorithm-driven approach, the application determines exactly what information it needs from the environment, and when to get it. The text-based interfaces that we designed in Chapters 7 and 8 are algorithm driven.

While the application is active in an algorithm-driven interface, it is passive in an event-driven system. In an event-driven system, the application waits for something to happen in the environment: that is, it waits for an event. When an event occurs, the application responds to the event, and then waits for the next event. Applications with a graphical, window-based user interface (usually called simply a graphical user interface, or GUI) are almost always event driven.

**event-driven**: an input-output model in which the application waits for an event to occur, responds to the event, and waits for the next event.

An application with a window-based interface provides the user with a graphical “control panel” containing a range of options. There may be menus, buttons, text fields, etc. After displaying the interface, the application waits for the user to do something. The
user can press a button, choose a menu item, enter text in a text field, or exercise any other option provided. When the user performs an action, the application responds to this event, and then waits for the next user action.

In a window-based system, there is a native windowing system (such as Windows XP or X Windows) that manages the display and detects events like mouse clicks, mouse movement, key strokes, etc. A Java application interacts with the native windowing system through AWT components. The application communicates with the native windowing system to create a display. Then events occurring in the display are detected by the native windowing system and delivered to the application. The application performs some action in response to the event (possibly changing the display), and then waits for notification of the next event.

17.2 An introduction to Swing

17.2.1 Components

A graphical user interface is made up of components (sometimes called widgets). Components are things like windows, buttons, checkboxes, menus, scroll bars, text fields, labels,
and so on. A component occupies display screen real estate: it has location, size, shape, color, etc.

There are two fundamental types of components.

- **Basic** components, sometimes called “atomic” components, are self-contained entities that present information to or get information from the user. A basic component can be a very simple thing like a button or label, or a complex structure like an editor pane (a text area for entering, displaying, and editing various kinds of textual content) or a combo box (a component that combines a button or editable field with a drop-down list).

- **Containers** are components whose function is to hold and position other components. A top-level container contains all of the visual components of an application’s graphical user interface, and provides the screen real estate used by the application. The window in which an application is run, such as the window labeled **Components** in Figure 17.3, is a top-level container. Other containers, called intermediate containers, are used to simplify the organization and positioning of basic graphical components.

### Some elementary component attributes

The Java Swing components are defined in the package `javax.swing` and its subpackages. An application uses these classes to build and manage a graphical user interface. One of the fundamental classes in `javax.swing` is the abstract class `JComponent`. Most of the components created by an application are instances of some `JComponent` subclass

`JComponent` is a subclass of the older AWT class `java.awt.Component`, and inherits many of its most obvious properties from that class. Component properties include foreground and background colors, location, size, and font. We’ll see more as we go along. The values of these properties can be obtained with queries:

```java
public Color getForeground ();
public Color getBackground ();
public Point getLocation ();
public Dimension getSize ();
public Font getFont ();
```
and can be set with methods:

```java
public void setForeground (Color fg);
public void setBackground (Color bg);
public void setLocation (Point p);
public void setSize (Dimension d);
public void setFont (Font f);
```

Many of the methods are overloaded. For instance, there are versions of `setLocation` and `setSize` that take two `int` arguments rather than a `Point` or `Dimension`.

`Color`, `Point`, `Dimension`, and `Font` are AWT classes (defined in the package `java.awt`) that model the obvious notions. Instances of the class `Color` are immutable: once the instance is created, it cannot be changed. The class `Color` defines a number of constant references, such as `Color.red`, `Color.blue`, `Color.green`. 

---

**Figure 17.3** Some Swing components.
An instance of the class `Point` represents a (relative) position in an $x$-$y$ coordinate space. Units are pixels, and the origin (0,0) is in the upper left. Attributes of a `Point` can be accessed through `public` instance variables $x$ and $y$:

```java
public int x;
public int y;
```

(The methods `getX` and `getY` return the coordinates as `doubles`. There are no `setX` and `setY` methods.)

An instance of the class `Dimension` encapsulates width and height, again in pixels. Attributes are accessed directly through public instance variables as was the case with `Point`:

```java
public int height;
public int width;
```

(As with `Point`, methods `getHeight` and `getWidth` return the values of the attributes as `doubles`.)

Instances of the class `Font` represent text fonts. A font provides information needed to render text in a visible way. Working with fonts is a rather complex business. We touch briefly on fonts later in the chapter.

**Basic components**

As we mentioned, basic or “atomic” components present information to or get information from the user. We briefly describe some of common components below. Some are illustrated in Figures 17.3 and 17.5. These components are defined in `javax.swing` and are subclasses of `JComponent`.

The following are used primarily to get input from the user:

1. A pixel is the smallest dot that can be independently colored on a display screen.
17.2 An introduction to Swing

**JButton** a simple push button that can be “pushed” with a mouse click.

**JCheckBox** a button that can be toggled on or off, and displays its state to the user.

**JRadioButton** a button that can be toggled on or off, and displays its state to the user. Only one radio button in a group can be on.

**JComboBox** a drop-down list with an optional editable text field. The user can either key in a value or select a value from the drop-down list.

**JList** a component that allows a user to select one or more items from a list.

**JMenu** a popup list of items from which the user can select. Menus either appear on a “menu bar” or as “pop-ups” associated with a user action such as pressing the right mouse button.

**JSlider** a component that lets the user graphically select a value by sliding a knob within a bounded interval.

**JTextField** an area for entering a single line of input.

These components are used to provide information to the user.

**JLabel** a component containing a short text string, an image, or both.

**JProgressBar** a component that communicates the progress of some work by displaying its percentage of completion and possibly a textual display of this percentage.

**JToolTip** a small window that describes another component.

The remaining atomic components provide formatted information and a way of editing it.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Last Name</th>
<th>Position</th>
<th>Average</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnny</td>
<td>Damon</td>
<td>Jr</td>
<td>3.85</td>
<td>19</td>
</tr>
<tr>
<td>Toby</td>
<td>Walker</td>
<td>Sr</td>
<td>3.92</td>
<td>12</td>
</tr>
<tr>
<td>Norman</td>
<td>Cardopina</td>
<td>As</td>
<td>3.92</td>
<td>5</td>
</tr>
<tr>
<td>Manny</td>
<td>Ramirez</td>
<td>Jr</td>
<td>3.12</td>
<td>24</td>
</tr>
<tr>
<td>David</td>
<td>Gritt</td>
<td>Gh</td>
<td>2.91</td>
<td>64</td>
</tr>
</tbody>
</table>

**Figure 17.5** Some complex components: Table and Tree.
A JComponent is-a Container is-a Component.

JTree
- a component that displays hierarchical data in outline form.

JTable
- a component user to edit and display data in a two-dimensional grid.

Classes JTextArea, JTextPane, and JEditorPane define multi-line areas for displaying, entering, and editing text.

17.2.2 Containers

An object that contains components is a container. Containers are modeled by the java.awt abstract class Container. An important simplifying aspect of the library structure is that a Container is just another kind of component. That is, the class Container is a subclass of the class Component. This means that a Container can contain another Container. Furthermore, the Swing class JComponent is a subclass of Container. This relationship is illustrated in Figure 17.6. In this and subsequent figures, shaded rectangles denote AWT classes, and un-shaded rectangles Swing classes.

Since JComponent is a subclass of Container, any JComponent, even a basic component, can contain other components. However, basic components are generally not used to hold other components.

component: a distinct element of a graphical user interface, such as a button, text field, and so on.

container: a graphical user interface component that can contain other components.

Since a Container can hold another Container, a user interface has a hierarchical containment structure.
Do not confuse this hierarchy with the subclass hierarchy. The containment hierarchy is defined by a relation between objects: one object contains another.

We often use “genealogical” terminology when referring to objects in the hierarchy: if \( A \) contains \( B \), we say “\( A \) is the parent of \( B \)” and “\( B \) is the child of \( A \).” Terms like ancestor, descendent, etc. have the obvious meanings. In the above illustration, \( A \) is parent of \( B \) and an ancestor of \( D \). \( F \) is a child of \( C \) and a descendent of \( A \).

The class \( \text{Container} \) defines an extensive set of methods for adding components and manipulating its contents, including five overloaded version of the method \( \text{add} \). But containers in a user interface tend to be stable. Once they are built and populated with components, there is rarely much need to add to, remove from, or rearrange their contents.

**Intermediate containers**

Intermediate containers are used to organize and position other components. The simplest, garden-variety, no-frills intermediate container is a \( \text{JPanel} \). A \( \text{JPanel} \) is generally used just as a place for collecting other components. For instance, the following creates a \( \text{JPanel} \) and adds two buttons, labeled “on” and “off.”

```java
    JPanel p = new JPanel();
    p.add(new JButton("on"));
    p.add(new JButton("off"));
```

Some other intermediate containers are:

- \( \text{JScrollPane} \) provides a “scrollable” view of another component; that is, a view with scroll bars (see Figure 17.3);
- \( \text{JSplitPane} \) divides two components graphically; the components can be interactively resized by the user;
Chapter 17  Building a graphical user interface

`JTabbedPane` lets the user switch between a group of components by clicking on a labeled tab;

`JToolBar` is useful for displaying a set of commonly used controls.

**Top-level containers**

A *top-level container* is one that is not contained in any other container. Top-level containers provide a screen area where other components can display themselves. Instances of Swing classes `JApplet`, `JDialog`, `JFrame`, and `JWindow` are commonly used as top-level containers. The standard top-level container for an application’s user interface is a `JFrame`. We concentrate our attention on `JFrame`

A `JFrame` is a window with title and border. It can be moved, resized, iconified, etc. like any other native system window. A `JFrame` can also have a menu bar. There are two things to note about the class `JFrame`. First, though it is a subclass of `java.awt.Container`, it is not a subclass of `JComponent`. Second, it delegates the responsibility of managing its components to another object.

The detailed structure of a `JFrame` need not concern us at the moment. We can consider a `JFrame` to have a component called a *content pane*\(^1\). This is the working area of the `JFrame`, excluding the title, border, and menu. The content pane is generally a `JPanel`. If we want to add a component to a `JFrame`, we actually add it to the `JFrame`'s content pane. The content pane can be obtained with the method `getContentPane`, which returns a `Container`. We add a button to a `JFrame` like this:

```
JFrame f = new JFrame("A Frame");
JButton b = new JButton("Press");
Container cp = f.getContentPane();
cp.add(b);
```

![A Frame](image)

**Figure 17.7** A `JFrame` is a top-level application window.

---

1. A `JFrame` has a `JRootPane`, which has a `JLayeredPane`, which actually contains the content pane and an optional `JMenuBar`.
17.3 Creating a display

17.3.1 The top-level frame

The implementation of a Java application with a graphical user interface involves the use of a JFrame to display application output, gather user input, and provide access to the application’s functionality. Creating and displaying a frame is relatively simple. Let’s start with a trivial program that creates an empty display. For simplicity, we do everything at first in the method main. In an actual system, of course, building and managing the display is the job of the user interface.

First, we need to create a JFrame instance:
import javax.swing.*

public class DisplayFrame {

    public static void main (String[] args) {
        JFrame f = new JFrame("A Frame");
        ...
    }

    The String argument to the JFrame constructor gives the frame a title, and can be omitted.
    The JFrame instance has an initial width and height of 0 when created. We’ll set its size a little bigger, to make it easier to deal with:

1. How a 0 × 0 window actually looks is system dependent.
17.3 Creating a display

```java
f.setSize(300,200);
```

This sets the width of the `JFrame` to 300 pixels and the height to 200. The exact physical size of the `JFrame` and how much display real estate it will occupy depend on the size and resolution of the display screen.

If we look up the class `JFrame` in the Swing documentation, we won't find the method `setSize`. `JFrame` inherits `setSize` from `Component`.

The `JFrame` has been created and sized, but is not yet visible on the display screen. In effect, the native windowing system which actually manages the display knows nothing about the `JFrame`. To show the `JFrame`, we must set its `visible` property to `true`:

```java
f.setVisible(true);
```

This command does considerably more than change the state of the object. A peer is created, and the `JFrame` is displayed on the screen, as shown in Figure 17.7.

17.3.2 The event dispatching thread

Suppose we complete the program from the previous section,

```java
import javax.swing.*;

public class DisplayFrame {

    public static void main (String[] args) {
        JFrame f = new JFrame("A Frame");
        f.setSize(300,200);
        f.setVisible(true);
    }
}
```

and run it. A window similar to that pictured in Figure 17.7 is displayed on the screen. Simple enough. But something odd is happening. Look at the method `main`. It consists of three statements. When the program is run, the method `main` is executed. The three statements comprising `main` are executed, and the method terminates. The variable `f`, local to `main`, is deallocated. But the application has not terminated, even though the method `main` has completed execution. The `JFrame` remains on the screen. You can resize, move, and iconify the `JFrame`, just as with any other native GUI window.

Choosing “Close” from the `JFrame` window menu will likely close the window, but not terminate the application. As things are now, the only way that you can terminate the application is by an explicit signal from the operating system. (Control-C works on many systems, but if you don’t know how to get the operating system to kill a process, don’t run this application!)

The reason the application does not terminate when `main` is finished is that a second thread, called the event dispatching thread, is created when the `JFrame` is set visible. A thread is essentially a sequence of instructions being executed by the processor.
So far, all of our programs have consisted of a single thread, called the \textit{main} thread. This is the sequence of actions initiated by the \textit{main} method. But a program can contain several threads. That is, there can be several independent sequences of actions being carried out simultaneously in the program. If there is a single processor executing the program, these actions are not in fact happening at the same time. The processor will execute a few instructions from one thread and then execute a few instructions from another. But in general we cannot determine whether a particular statement in one thread will be executed before or after a particular statement in another thread.

Managing threads is a complex and demanding business, as it is easy to introduce subtle errors into the code. The details of coordinating and synchronizing interacting threads is well beyond the scope of this text. However, you should understand that all the code that involves repainting components and handling events executes in the event dispatching thread. After the \textit{JFrame} has been made visible, the main thread should not in general perform actions that affect or depend on the state of the user interface.\footnote{In fact, invoking \texttt{setVisible} from the main thread is not entirely thread safe and is not recommended practice in production systems. The event dispatching thread can be active before \texttt{setVisible} completes. A detailed discussion of the problem and possible solutions is well beyond the scope of this text. See \url{http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html} for more information.}

\subsection{Adding components: layout}

Containers are used to contain other components. These components are added to the container using one of the \texttt{add} methods mentioned earlier. But as you might guess, it is not enough simply to add a component. We want to control how a component will be positioned in the container when the container is painted on the screen.
To handle this problem, each container is equipped with a layout manager. A layout manager is an object with the responsibility of positioning and sizing components in a container. The container delegates these responsibilities to its layout manager. We control the positioning of components by setting the container’s layout manager and by telling the layout manager how to position components as they are added to the container.

A layout manager must implement the interface `java.awt.LayoutManager`. This interface specifies the methods all layout managers must provide. Some layout managers additionally implement the interface `java.awt.LayoutManager2`, which extends `LayoutManager`. A `Container` has query `getLayout` and command `setLayout` for accessing and setting its layout manager:

```java
public LayoutManager getLayout();
public void setLayout (LayoutManager manager);
```

Java provides several classes that implement `LayoutManager`, including `FlowLayout`, `BorderLayout`, `GridLayout`, `CardLayout`, `GridBagLayout`, `BoxLayout`, and `OverlayLayout`. `BoxLayout` and `OverlayLayout` are defined in the package `javax.swing`. The others are defined in `java.awt`. You can, of course, implement your own layout manager. Some layout manager classes are designed so that a single layout manager instance can manage the layout of several containers. In other cases, each container requires its own layout manager instance.

Discussing layout in detail is well beyond the scope of this chapter. We briefly summarize the standard layout managers.

- **FlowLayout** lays out components left to right, top to bottom. `FlowLayout` is the default layout of a `JPanel`.
- **BorderLayout** lays out up to five components, positioned “north,” “south,” “east,” “west,” and “center.” `BorderLayout` is the default layout of a `JFrame`’s content pane.
Chapter 17  Building a graphical user interface

*GridLayout* lays out components in a two-dimensional grid.

*CardLayout* displays components one at a time from a preset deck of components.

*GridLayout* lays out components vertically and horizontally according to a specified set of constraints; the most complex and flexible of the Java-provided layout managers.

*BoxLayout* lays out components in either a single horizontal row or single vertical column. *BoxLayout* is the default layout for the Swing lightweight container *Box*.

*OverlayLayout* lays out components so that specified component alignment points are all in the same place. Thus components are laid out on top of each other.

The default layout manager of a *JPanel* is a *FlowLayout*. However, a *JFrame*’s content pane is a *JPanel* with a *BorderLayout*. A *FlowLayout* simply lays out components in the order in which they are added to the container. With a *BorderLayout*, a component’s position is specified by a second argument to the add method when the component is added to the container. For instance, if *pane* is a container with a *BorderLayout*,

```java
pane.add(new JButton("Push"), BorderLayout.NORTH);
```

adds a *JButton* to the container to be positioned “north”: that is, at the top of the container. Figures 17.7 and 17.7 illustrate containers with *FlowLayout* and *BorderLayout* respectively.

**Container validity**

We mention one further *Container* property before moving on to events. A *Container* is valid if it does not need to be laid out. That is, if its size is known to the system, and its

*Figure 17.12*  Two instances of a *JFrame* laid out with *FlowLayout*. 
A Container is invalid if its state is inconsistent with its appearance. Adding a component to a Container does not automatically cause the Container to be laid out again. The layout manager determines when to layout the Container. A Container to which a component has been added after it was last laid out is invalid.

Any number of things can cause a layout manager to lay out a Container. Layout managers typically notice a change in Container size, for example, and lay out the Container again if the user resizes the Container. The command validate explicitly sets the Container’s valid property to true and instructs the layout manager to lay out the container. The query isValid returns the value of this property. The features are specified (for the class Component) as follows:

```java
public boolean isValid ();
public void validate ();
```

DrJava: layout managers

The purpose of this exercise is to see how layout managers position and size components. Open the file BorderExample1.java, located in directory layout, in ch17, in nhText. This file contains a program that generates a JFrame containing a single button. The button is positioned north by a BorderLayout.

1. In the Interactions pane, run the program BorderExample1.
   ```
   > java ch17.layout.BorderExample1
   ```
   Resize the window and note how the layout manager resizes the button. The height of the button remains fixed, but the width expands and shrinks as the window is resized. To exit the program, close the window.

2. Modify the program so that the button is positioned west rather than north. Save and recompile.

3. Run the program again and notice how this button is resized as you resize the window.
Open the file BorderExample2.java. This program generates a JFrame containing five buttons positioned by a BorderLayout.

4. Run the program BorderExample2:

   > java ch17.layout.BorderExample2

   Resize the window and note how the layout manager resizes the buttons.

Open the file FlowExample.java. This program generates a JFrame containing ten buttons positioned by a FlowLayout.

5. Run the program FlowExample:

   > java ch17.layout.FlowExample

   Resize the window and note how the layout manager positions the buttons.

Open the file GridExample.java. This program generates a JFrame containing six buttons positioned by a GridLayout.

6. Run the program GridExample:

   > java ch17.layout.GridExample

   Resize the window and note how the layout manager positions and resizes the buttons.

7. Edit the program so that ten buttons are produced rather than six. Do not change the specification of the grid. Save, compile, and run the program. Note how the ten buttons are positioned.

17.4 Events: programming the user interface

An event-driven system waits for and responds to external events. In a system with window-based graphical user interface, external events are user actions such as moving the mouse, pressing a key, etc. Programming the user interface reduces to capturing and handling these events.

Some events are low-level: pressing or releasing a key on the keyboard, moving the mouse, pressing or releasing a mouse button. Other events are high-level: selecting an item from a menu, pressing a button, entering text in a field. Note that a high-level event usually involves one or more low-level events: to enter text in a text field, the user moves the mouse cursor, clicks a mouse button, and presses and releases several keyboard keys.

We summarize some events categories below. We limit our attention in the examples, however, to high-level events. (See Table 17.1 on page 690 for a more comprehensive list of components and the events they can generate.)

- key event: keyboard key has been pressed or released.
- mouse event: mouse button has been pressed or released; mouse has been moved or dragged; mouse cursor has entered or left component.
17.4 Events: programming the user interface

**component event:** component has been hidden, shown, resized, or moved.

**container event:** component has been added to or removed from a container.

**window event:** window has been opened, closed, iconified, de-iconified, activated, deactivated.

**focus event:** component has gained or lost focus.

The following are high-level events.

**action event:** component-defined action has occurred (e.g., a button has been pressed, a checkbox selected, Return/Enter pressed in a JTextField).

**adjustment event:** scrollbar has been moved.

**item event:** user selects has selected a checkbox or list item.

**document event:** content has changed in a TextComponent.

---

**Figure 17.14** Some of the Java event classes.
An important aspect of Components is that they are the source of events: events occur within Components. The Component in which an event occurs is said to generate the event, or to be the source of the event.

Events are, surprise, represented by objects. Many events are modeled by subclasses of the abstract class `java.awt.AWTEvent`, which is itself a subclass of `java.util.EventObject`. (See Figure 17.14.)

An event object knows the event source and other relevant information about the event. We can determine the source of an event with the query `getSource`, defined on the class `EventObject`:

```java
public Object getSource();
```

Note that the method returns an `Object`. If we know that the source is, say, a `JButton`, and want to treat it as such, we must cast the returned value to `JButton`. We’ll see examples below.

An object that is interested in knowing when an event occurs is called a listener, or event handler. To be notified of an event, a listener must register with the event’s source:

```java
public void addNotify()
```
that is with the component that will generate the event. A component can have any number of listeners, and a listener can register with any number of components. When an event occurs, each listener registered with the event’s source is notified by having a specific listener method invoked. The relation between a listener and an event source is the same as that between an InteractiveController and InteractivePlayer, defined in Section 9.5.3.

Each kind of event is associated with an interface, called the event listener interface. This interface specifies the methods that must be implemented by a handler for that kind of event. Programming a handler for an event consists of implementing the interface associated with the event type (See Figure 17.17, and Table 17.2 on page 696 for a list of events, their listener interfaces, and the methods specified by the interfaces).

**Figure 17.15** An event listener is notified when an event occurs.

**17.4.1 An example**

Let’s develop a simple example to get an idea of how this works. We program an application that displays a button. When the button is pressed, its foreground and background colors are swapped.

First, we create a JFrame containing a single large button with black foreground and white background. Since it’s the only component, we add it to the center of the content pane. The BorderLayout manager allocates any extra space to the button. Thus the button occupies the entire content pane, as shown in Figure 17.16.

```java
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class OnOffSwitch extends JFrame {

    public OnOffSwitch () {
        super("On/Off Switch"); // frame title
```
// create button and set its colors
JButton button = new JButton("On/Off");
button.setForeground(Color.black);
button.setBackground(Color.white);

// add button to JFrame’s content pane:
this.getContentPane().add(
    button, BorderLayout.CENTER);
}
}

public class OnOffTest {
    public static void main (String[] args) {
        OnOffSwitch frame = new OnOffSwitch();
        frame.setSize(300,200);
        frame.setVisible(true);
    }
}

We’ve organized things a little differently this time. Specifically, we’ve extended the class JFrame with OnOffSwitch, and given its constructor the responsibility for building the frame.

If we execute the program, pressing the button will have no effect at all. This is not surprising. When the button is pressed, it generates an ActionEvent. (Actually, pressing the button generates a number of events, such as mouse pressed, mouse released, etc. We are concerned only with the high-level ActionEvent.) But we have not yet written any listeners that will do anything about the event. We have not “programmed” the user interface.

**Programming the gui: adding an ActionListener for the JButton**

If the user presses the button, the JButton object generates an ActionEvent. Let’s implement a listener that will handle this event.
First, the listener – we’ll call it a Switcher – must implement the interface `java.awt.event.ActionListener`. As we’ve said, Java defines an interface for each category of event, and an object interested in being notified of a particular kind of event must implement the appropriate interface. This insures that the listener object defines the methods that will be called when an event occurs.

There is only one method specified in the interface `ActionListener`:

```java
public void actionPerformed (ActionEvent e);
```

This method is invoked to inform the listener that an event has occurred. A minimal implementation of the `Switcher` class looks like this:
class Switcher implements ActionListener {

    public void actionPerformed (ActionEvent e) {
    }
}

Of course with this implementation, a Switcher does nothing about the event after being informed of it. We’ll take care of that in a bit. First, though, let’s add code to create a listener and register it with the JButton component:

    public OnOffSwitch () {
        super("On/Off Switch"); // frame title

        // create button and set its colors
        JButton button = new JButton("On/Off");
        button.setForeground(Color.black);
        button.setBackground(Color.white);

        // create and register button’s listener:
        button.addActionListener(new Switcher());

        // add button to JFrame’s content pane:
        this.getContentPane().add(button, BorderLayout.CENTER);
    }

We create a new Switcher instance and register it with the JButton component by calling the JButton’s addActionListener method. This tells button that the listener wants to be informed of any ActionEvents it generates.

Finally, let’s make the Switcher do something in response to the event. As we said above, we’ll have it invert the foreground and background colors of the button.

    class Switcher implements ActionListener {

        public void actionPerformed (ActionEvent e) {
            Component source = (Component)e.getSource();
            Color oldForeground = source.getForeground();
            source.setForeground(source.getBackground());
            source.setBackground(oldForeground);
        }
    }

Note that we query the event to determine the source. Since the query getSource returns an Object, we must cast the result to a Component in order to invoke Component methods getForeground, setBackground and setForeground.

Let’s review this program before moving on. There are three objects involved: the top-level OnOffSwitch (a JFrame); a component JButton; and an event handler, the ActionListener Switcher. The main method creates the top-level OnOffSwitch, sizes it, makes it vis-
17.4 Events: programming the user interface 695

The `OnOffSwitch` constructor creates a `JButton` and then registers a `Switcher` with the `JButton` so that any `ActionEvents` generated by the `JButton` will be delivered to the `Switcher`. Finally, the `OnOffSwitch` adds the `JButton` to itself as a component (by adding the `JButton` to its content pane).

When the user presses the button, an `ActionEvent` is generated by the button. This event is delivered to all of the button’s `ActionListeners`, by means of an invocation of their `actionPerformed` method. In this case, there is only one listener and its `actionPerformed` method inverts the button’s foreground and background colors.

**Programming the gui: adding a WindowListener for the JFrame**

Next, let’s look at how we can terminate the application cleanly, without explicitly requesting the operating system to kill it.

We would like to terminate the application when the user selects the “Close” option from the top-level window menu. Selecting “Close” generates a `WindowEvent` in the `JFrame`, specifically a window-closing event. So we must first create a listener for `WindowEvents`. The `WindowListener` interface is a bit more complicated than the `ActionListener` interface. `WindowListener` specifies seven methods, while `ActionListener` specifies only one. Each of these methods handles a different kind of `WindowEvent`. Specifically, `WindowListener` specifies the methods:

- `void windowActivated (WindowEvent e)`
  Invoked when the window is set to be the user’s active window, which means the window (or one of its subcomponents) will receive keyboard events.

- `void windowClosed (WindowEvent e)`
  Invoked when a window has been closed as the result of calling `dispose` on the window.

- `void windowClosing (WindowEvent e)`
  Invoked when the user attempts to close the window from the window’s system menu.

- `void windowDeactivated (WindowEvent e)`
  Invoked when a window is no longer the user's active window, which means that keyboard events will no longer be delivered to the window or its subcomponents.
Table 17.2  Generated events and corresponding listener interfaces

<table>
<thead>
<tr>
<th>Event Generated</th>
<th>Listener Interface</th>
<th>Listener Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionEvent</td>
<td>ActionListener</td>
<td>actionPerformed()</td>
</tr>
<tr>
<td>AdjustmentEvent</td>
<td>AdjustmentListener</td>
<td>adjustmentValueChanged()</td>
</tr>
<tr>
<td>ComponentEvent</td>
<td>ComponentListener</td>
<td>componentHidden()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>componentMoved()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>componentResized()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>componentShown()</td>
</tr>
<tr>
<td>ContainerEvent</td>
<td>ContainerListener</td>
<td>componentAdded()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>componentRemoved()</td>
</tr>
<tr>
<td>FocusEvent</td>
<td>FocusListener</td>
<td>focusGained()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>focusLost()</td>
</tr>
<tr>
<td>ItemEvent</td>
<td>ItemListener</td>
<td>ItemStateChanged()</td>
</tr>
<tr>
<td>KeyEvent</td>
<td>KeyListener</td>
<td>keyPressed()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>keyReleased()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>keyTyped()</td>
</tr>
<tr>
<td>MouseEvent</td>
<td>MouseListener</td>
<td>mouseClicked()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mouseEntered()</td>
</tr>
<tr>
<td></td>
<td>MouseMotionListener</td>
<td>mouseDragged()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mouseMoved()</td>
</tr>
<tr>
<td>TextEvent</td>
<td>TextListener</td>
<td>textValueChanged()</td>
</tr>
<tr>
<td>WindowEvent</td>
<td>WindowListener</td>
<td>windowActivated()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>windowClosed()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>windowClosing()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>windowDeactivated()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>windowDeiconified()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>windowIconified()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>windowOpened()</td>
</tr>
</tbody>
</table>
17.4 Events: programming the user interface

void windowDeiconified (WindowEvent e)
Invoked when a window is changed from a minimized to a normal state.

void windowIconified (WindowEvent e)
Invoked when a window is changed from a normal to a minimized state.

void windowOpened (WindowEvent e)
Invoked the first time a window is made visible.

We are not interested in most of these WindowEvents. We’re only interested in events associated with the user closing the window. To simplify implementation of listeners, Java provides a collection of abstract event adapter classes. These adapter classes (see Figure 17.19) implement listener interfaces with empty, do-nothing methods. To implement a listener class, we can extend one of the adapter classes and override only the methods we are interested in.

We’ll call our WindowListener class Terminator. It extends the adapter class WindowAdapter.

class Terminator extends WindowAdapter {
    ...
}

The class WindowAdapter implements WindowListener, and since Terminator is a subclass of WindowAdapter, it also implements WindowListener. Terminator inherits the seven WindowListener methods from WindowAdapter. These are all null “do-nothing” methods. We must override the methods relating to the events we’re interested in: namely, the events that are generated when the user tries to close the window.

A window-closing event occurs when the user attempts to close the window. This event results in an invocation of the listener’s windowClosing method. Thus we must override windowClosing:

class Terminator extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        ...
    }
}

What should we do when a window-closing event is delivered? We could simply terminate the application. But there is a slightly better approach, since in general we do not know if there are other listeners also interested in this event. What we’ll do is release the window’s resources, close it, and remove its peer. This is accomplished by the Window method dispose:

class Terminator extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        Window w = e.getWindow();
        w.dispose();
    }
}
The `WindowEvent` method `getWindow` is essentially the same as the `EventObject` method `getSource`, except that it returns a `Window` rather than an `Object`. Thus we are saved the trouble of casting the result.

Disposing of the window causes it to generate one more event, a `window-closed` event. We handle this event with the method `windowClosed`, and here is where we finally terminate the application:

```java
class Terminator extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        Window w = e.getWindow();
        w.dispose();
    }
    public void windowClosed(WindowEvent e) {
        System.exit(0);
    }
}
```

The call to `System.exit` causes the application to terminate. By convention, an argument of 0 indicates normal termination, and a nonzero argument indicates abnormal termination – typically because of some error condition.

Of course, we still must create a `Terminator` instance and register it with the top-level `JFrame`. This is straightforward:

```java
public OnOffSwitch () {
    super("On/Off Switch");
    ...
    this.addWindowListener(new Terminator());
    ...
}
```

Since we only need one instance of the `WindowListener`, rather than explicitly defining a class like `Terminator`, we typically use an anonymous class (see Section 14.2.4):

```java
this.addWindowListener(
    new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            Window w = e.getWindow();
            w.dispose();
        }
        public void windowClosed(WindowEvent e) {
            System.exit(0);
        }
    });
```
The application is now complete, and consists of the classes OnOffSwitch, Switcher, and Terminator (or anonymous WindowListener).

This introductory example illustrates only graphical user interface design and implementation. We have no model component at all. In a complete application, the user interface event handlers interact with the model, and changes in the model generally require the interface to be updated.

**Sizing components**

In the example just completed, we set the size of the JFrame but not the size of the JButton. In general, the size of a component is controlled by the layout manager of the container holding the component. The JFrame is a top-level container, and so is not contained in any other container. Therefore its size is not controlled by a layout manager. The size of the JButton, on the other hand, is determined by the BorderLayout manager of the JFrame’s content pane.

For components controlled by a layout manager, the best we can do is to provide suggestions, or “hints,” about their size to the layout manager. This is done by overriding a
 Exactly how the layout manager uses this information is up to the layout manager. Some layout managers, such as FlowLayout and GridBagLayout, invoke these methods to determine how to size a component. BorderLayout, on the other hand, gives the north and south components their preferred height, and the east and west components their preferred width. North and south components are stretched or squashed horizontally, and east and west components are stretched or squashed vertically, depending on the container size. The center component gets whatever space is left over. In particular, if you resize the window containing the OnOffSwitch’s button, the button will be resized as well.

Let’s sketch a simple approach for having the button remain the same size, regardless of how the user resizes the window. First, we extend JButton, overriding the methods getPreferredSize and getMinimumSize so that they return the size we want:

```java
class OnOffButton extends JButton {

    public OnOffButton (String label) {
        super(label);
    }

    public Dimension getPreferredSize () {
        // make the width 50 pixels, height 100
        return new Dimension(100,50);
    }

    public Dimension getMinimumSize () {
        return this.getPreferredSize();
    }
}
```

Rather than creating a JButton in the OnOffSwitch constructor, we create an OnOffButton:

```java
public OnOffSwitch () {
    super("On/Off Switch"); // frame title

    // create button and set its colors
    OnOffButton button = new OnOffButton("On/Off");
    ...
}
```

Finally, we change the content pane’s layout manager to a FlowLayout:

```java
public OnOffSwitch () {
    ...

    // use FlowLayout and add button to JFrame’s
    // content pane:
```

1. Most standard layout managers ignore maximum size.
17.4 Events: programming the user interface 701

```java
this.getContentPane().setLayout(new FlowLayout());
this.getContentPane().add(button);
}

As shown in Figure 17.20, FlowLayout will use the JButton's preferred size and keep it centered horizontally, regardless of how the window is resized. The button, however, will be at the top of the pane. If we want the button to remain centered vertically as well as horizontally, we would use a more flexible (but more complicated) layout manager like GridBagLayout. (We'll see an example using GridBagLayout in the next chapter.)

**Event handling happens in the event dispatching thread**

Remember that all component repainting and event handling is done in the event dispatching thread. Suppose we put a delay in the action event handling method:

```java
public void actionPerformed (ActionEvent e) {
    Component source = (Component)e.getSource();
    Color oldForeground = source.getForeground();
    source.setForeground(source.getBackground());
    source.setBackground(oldForeground);
    // wait 10 seconds:
    nhUtilities.utilities.Control.sleep(10);
}
```

(Recall from Section 8.3.5 that the nhUtilities.utilities.Control method sleep causes the program – actually the thread – to pause for a specified number of seconds.)

If we run the program now, we will notice that there is about a 10 second delay between the time we press the button and the time it changes color. Furthermore, the application will seem “frozen” during this period. The reason is that the button is scheduled to be repainted, but is not repainted until after the action event handling is complete. In fact, no repainting or other event handling will be done until after the action event handling is complete. This implies that event handling methods should do their work quickly. If the response to an event requires an lengthy, time-consuming action, a new thread should be created to

![Figure 17.20 OnOffSwitch using a FlowLayout.](image)
perform the action. Creating and managing threads, however, is beyond the scope of our discussion.

DrJava: threads

Open the file DelayExample.java, located in threads, in ch17, in nhText. This application generates a JFrame containing two buttons and two text fields. The text fields display how many times each button has been pressed.

Note that the ButtonPanel constructor has an int parameter, delay. If the value of the argument is positive, the button’s action listener will sleep for the specified number of seconds before returning.

1. In the Interactions pane, run the application DelayExample and note its behavior.

   > java ch17.threads.DelayExample

   Exit the program by closing the window.

2. Modify the program so that the west ButtonPanel is created with a five second delay in its event handler:

   pane.add(new ButtonPanel(5),BorderLayout.WEST);

3. Run the program again. Note that the text field is not repainted until after the delay. Also notice that the application appears to “freeze” during the delay. This is because repainting and event handling is all done in the same thread. Repainting is not done until handling the button press event is complete.

17.5 A more complex example

We next develop a little more complex user interface. In this section, we just build the display and show how to use a few more components. We won’t bother completely programming the event handlers.

The interface, pictured in Figure 17.21, is for an application that gathers some miscellaneous student data. It is consists of five areas: the top, occupied by a text field used to enter age; the left, containing a combo box used to enter college; the center, consisting of two buttons for sex; the right, containing check boxes for miscellaneous information.; and the bottom, consisting of a “finished” button. With this structure, it seems natural to use a BorderLayout.

Rather than adding components directly to the top-level frame’s content pane, we’ll build a JPanel and add the JPanel to the content pane. This is a standard technique. It gives us a bit more flexibility in that the JPanel can be easily incorporated into other structures if we need to expand or modify the application. Thus the main method will look like this

```java
public static void main(String[] args) {
    JFrame top = new JFrame("Student survey");
    top.getContentPane().add(
```
new StudentSurveyPanel(), BorderLayout.CENTER);
...

where

class StudentSurveyPanel extends JPanel {

    public StudentSurveyPanel () {
        setLayout(new BorderLayout());
        ...
    }

is the JPanel that contains the graphical structure. Note that we must explicitly set the JPanel's layout manager to BorderLayout. The default JPanel manager is FlowLayout.

We construct each section in its own JPanel, and add them to the StudentSurveyPanel.

For each section, we write a method that creates and returns the JPanel:

public StudentSurveyPanel () {
    setLayout(new BorderLayout());
    add(agePanel(), BorderLayout.NORTH);
    add(collegePanel(), BorderLayout.WEST);
    add(sexPanel(), BorderLayout.CENTER);
    add(miscPanel(), BorderLayout.EAST);
    add(finishedPanel(), BorderLayout.SOUTH);
    ...
}

Now all that remains is to write the five methods that construct the panels.

The button panel

Let’s start with the “finished” panel. It contains only a single button, and we’ve already seen how to use buttons. You’ll notice from the figure that this button has a raised, three-dimensional look to it. That’s because we’ve put a border around it. The JComponent method
public void setBorder (Border border)

is used to put a decorative border around a component. Standard borders are obtained from
the class javax.swing.BorderFactory. The particular border used here is a “raised bevel
border,” generated with the factory method

public static Border createRaisedBevelBorder ()

We also add an ActionListener for the button, but as we mentioned above, we won’t
program the ActionListener.

private JPanel finishedPanel () {
    JButton button = new JButton("Finished!");
    button.setBorder(
        BorderFactory.createRaisedBevelBorder());
    button.addActionListener(new DoneListener());
    JPanel finishedPanel = new JPanel();
    finishedPanel.add(button);
    return finishedPanel;
}

The JPanel’s default FlowLayout respects the preferred size of the button. If the win-
dow is resized by the user, the button will remain the same size and the JPanel that con-
tains it will be stretched or compressed as necessary.

The text field panel

The methods that create the other panels will have similar structures to finished-
Panel. But constructing them is a bit more complex.

In the age panel, there is a label and a text field contained in a panel with a border. We
can build it like this:

JLabel agePrompt = new JLabel("Age: ");
JTextField ageField = new JTextField(10);
ageField.addActionListener(new AgeListener());
JPanel innerPanel = new JPanel();
innerPanel.add(agePrompt);
innerPanel.add(ageField);
innerPanel.setBorder(
    BorderFactory.createRaisedBevelBorder());

The integer argument passed to the JTextField constructor, 10 in the example, indi-
cates the number of “columns” in the field. This number is used along with size of the
field’s font to determine the field’s preferred width. It does not limit the number of charac-
ters the user can enter.

An ActionEvent is generated in a JTextField when the user presses the Return/Enter
key in the field. An ActionListener can get the text in the field as a String using the JTextField’s getText method.
17.5 A more complex example

We need to be able to access the text field after it is constructed, other than from the
listener. Specifically, we want to clear the text field as part of a “reset” operation. Therefore,
rather than making the variable ageField local as shown above, we make it an
instance variable of the StudentSurveyPanel:

```java
private JTextField ageField;
```

We can add the innerPanel to a JPanel, just as we did with the button, so the
innerPanel won’t expand or contract if the window is resized.

```java
JPanel agePanel = new JPanel();
agePanel.add(innerPanel);
```

If the window is resized, the innerPanel remains the same size while the agePanel
expands or contracts as necessary.

Remember that a FlowLayout manager will keep the contents of its container cen-
tered right to left, but at the top of the container. To separate the innerPanel from the
top of the window and from the sections below it, we add an empty 10-pixel border to the
agePanel. The complete method can now be written. (Yes, yes, the literals should really
be named constants.)

```java
private JPanel agePanel () {
    JLabel agePrompt = new JLabel("Age: ");
ageField = new JTextField(10);
ageField.addActionListener(new AgeListener());
JPanel innerPanel = new JPanel();
innerPanel.add(agePrompt);
innerPanel.add(ageField);
innerPanel.setBorder(BorderFactory.createRaisedBevelBorder());
JPanel agePanel = new JPanel();
agePanel.add(innerPanel);
agePanel.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));
return agePanel;
}
```

The combo box panel

The college panel contains a simple combo box: a pull-down list of options from which
the user chooses one. To make the combo box, we simply add items to the pull-down list.
Like the age text field, we want the combo box to be available to a “reset” operation. So
we define it as a StudentSurveyPanel instance variable:

```java
private JComboBox collegeChooser;
```

The panel is decorated with a named border, using BorderFactory’s method

```java
public static TitledBorder createTitledBorder ( Border border, String title)
```
The collegePanel method can be written as follows:

```java
private JPanel collegePanel () {
    collegeChooser = new JComboBox();
    collegeChooser.addItem("Sciences");
    collegeChooser.addItem("Liberal Arts");
    collegeChooser.addItem("Engineering");
    collegeChooser.addItem("Fine Arts");
    collegeChooser.addActionListener(
        new CollegeListener());
    JPanel collegePanel = new JPanel();
    collegePanel.add(collegeChooser);
    collegePanel.setBorder(
        BorderFactory.createTitledBorder(
            BorderFactory.createLineBorder(Color.red),
            "College");
    return collegePanel;
}
```

The JComboBox generates an ActionEvent when the user selects an item from the list. The ActionListener can determine which item was selected by invoking the JComboBox method getSelectedItem or getSelectedIndex. For example, if the user chooses the option “Sciences” from the above list, getSelectedItem returns the String "Sciences", and getSelectedIndex returns the integer 0. (The method getSelectedItem is defined as returning an Object, so the result must be cast before it can be treated as a String.)

**The check box panel**

The “miscellaneous” panel contains a set of check boxes in a bordered panel. The check boxes themselves are independent of each other, and are instances of JCheckBox. Again, we make the check boxes instance variables so they can be reset.

```java
private JCheckBox worksBox;
private JCheckBox hasKidsBox;
private JCheckBox ownsPCBox;
```

A check box is a form of button (both JButton and JCheckBox are subclasses of JAbstractButton), and we can handle them much as we do a button. But let’s use an ItemListener instead of an ActionListener, and rather than creating a separate listener for each check box, let’s create one listener for the group.

First, we define a class MiscellaneousListener to implement ItemListener. We introduced inner classes in Section 10.5. We can make the listener classes private inner classes of StudentSurveyPanel. In particular, we define

```java
private class MiscellaneousListener
    implements ItemListener ...
```
in the class `StudentSurveyPanel`, and create an instance of this class:

```java
MiscellaneousListener miscListener =
    new MiscellaneousListener();
```

We can create a simple labeled check box and add the listener as follows:

```java
worksBox = new JCheckBox("Work");
worksBox.addItemListener(miscListener());
```

When the state of the check box changes, that is, when it is selected or deselected, the `ItemListener`'s `itemStateChanged` method is invoked:

```java
private class MiscellaneousListener
    implements ItemListener {

    public void itemStateChanged (ItemEvent e) {
        ...
    }
```

The `ItemEvent` method `getItemSelectable` tells which check box originated the event. The method `getStateChange` returns either the `int` value `ItemEvent.SELECTED` or the `int` value `ItemEvent.DESELECTED`, so the listener can determine whether the check box is being selected or deselected. The listener code will look like this:

```java
private class miscellaneousListener
    implements ItemListener {

    public void itemStateChanged (ItemEvent e) {
        JCheckBox box =
            (JCheckBox) e.getItemSelectable();
        if (box == worksBox)
            if (e.getStateChange() == ItemEvent.SELECTED)
                ...
```

Notice that the body of the method will end up involving six cases: three boxes, two possible state changes for each box. As we noted in Section 11.2, this is not the kind of structure we are happy to see in object-oriented code. But we’ll leave the code unimproved for now, having noted a slight smell.

We want to add the combo box boxes in a column to a `JPanel`. An easy way to do this is to equip the `JPanel` with a `GridLayout`. A `GridLayout` manager lays out the components in a rectangular grid of equal sized rectangles, as shown in Figure 17.22. The number of rows and columns is specified when the `GridLayout` is created.

```java
JPanel miscPanel = new JPanel();
miscPanel.setLayout(new GridLayout(3,1));
miscPanel.add(worksBox);
miscPanel.add(hasKidsBox);
miscPanel.add(ownsPCBox);
```
Finally, it takes a bit more effort to create the border, since the default position of the label is top left. In this case we want it to be top right. The completed method is given below.

```java
private JPanel miscPanel () {
    MiscellaneousListener miscListener =
        new MiscellaneousListener();

    worksBox = new JCheckBox("Work");
    worksBox.addItemListener(miscListener);
    hasKidsBox = new JCheckBox("Work");
    hasKidsBox.addItemListener(miscListener);
    ownsPCBox = new JCheckBox("Work");
    ownsPCBox.addItemListener(miscListener);

    JPanel miscPanel = new JPanel();
    miscPanel.setLayout(new GridLayout(3,1));
    miscPanel.add(worksBox);
    miscPanel.add(hasKidsBox);
    miscPanel.add(ownsPCBox);
    miscPanel.setBorder(
        BorderFactory.createTitledBorder(
            BorderFactory.createLineBorder(Color.red),
            "Misc",
            TitledBorder.RIGHT,
            TitledBorder.TOP));

    return miscPanel;
}
```

*Figure 17.22 A JFrame laid out with GridLayout.*

---

The image contains a JFrame with a grid layout, numbered rows and columns, and placeholders for labels and buttons. The text is discussing the creation of a JPanel with checkboxes and a border. The code snippet demonstrates how to create a JPanel with a GridLayout, add checkboxes, and set a titled border with a red line.

---

**Chapter 17 Building a graphical user interface**
The radio button panel

The final panel consists of two radio buttons for selecting sex. Radio buttons are different from check boxes in that exactly one of a group will always be selected.

As with the check boxes, we make the radio buttons `StudentSurveyPanel` instance variables.

```java
private JRadioButton maleButton;
private JRadioButton femaleButton;
```

After the buttons are created, they are added to a group. One of the buttons is initially selected.

```java
maleButton = new JRadioButton("Male");
femaleButton = new JRadioButton("Female");
femaleButton.setSelected(true);
ButtonGroup sexGroup = new ButtonGroup();
sexGroup.add(maleButton);
sexGroup.add(femaleButton);
```

Each time the user presses a button, one or two `ItemEvents` (one for the deselected button, one for the selected button) and one `ActionEvent` (for the selected button) are generated. We generally handle radio buttons with an `ActionEventListener`.

Again, let’s have a single handler for both buttons, which we’ll call a `SexListener`:

```java
private class SexListener implements ActionListener {

  public void actionPerformed(ActionEvent e) {
    ...
  }

  We’ll use a slightly different technique than we used with the check boxes to determine which button has been selected. `ActionEvents` contain a `String` called an “action command.” This `String` is set by the component generating the event, and can be accessed by the `ActionEvent` `getActionCommand` method. We’ll have the buttons set specific action commands in the `ActionEvent` they generate:

  ```java
  maleButton = new JRadioButton("Male");
  maleButton.setActionCommand("male");
  femaleButton = new JRadioButton("Female");
  femaleButton.setActionCommand("female");
  ```

  and the listener determine which button was selected by examining the action command:

  ```java
  public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("male") {
      ...
  }

  (Once again, we admit that all these `String` literals should really be named constants.)

  We add the buttons to a `JPanel` with a `GridLayout`, and to keep them centered, add this to a `JPanel` with `FlowLayout`. (Yes, there are other ways of doing this!) The complete method:

```
private JPanel sexPanel () {
    SexListener sexListener = new SexListener();
    maleButton = new JRadioButton("Male");
    maleButton.setActionCommand("male");
    maleButton.addActionListener(sexListener);
    femaleButton = new JRadioButton("Female");
    femaleButton.setActionCommand("female");
    femaleButton.addActionListener(sexListener);
    femaleButton.setSelected(true);
    ButtonGroup sexGroup = new ButtonGroup();
    sexGroup.add(maleButton);
    sexGroup.add(femaleButton);

    JPanel innerPanel = new JPanel();
    innerPanel.setLayout(new GridLayout(2,1));
    innerPanel.add(maleButton);
    innerPanel.add(femaleButton);
    JPanel sexPanel = new JPanel();
    sexPanel.add(innerPanel);
    return sexPanel;
}

The reset method

We’ve mentioned a “reset” method several times. This method puts the display in a standard, initial configuration. It sets the data collection widgets to their initial states, and requests focus for the text field.

private void resetView () {
    ageField.setText(" ");
    ageField.requestFocus();
    collegeChooser.setSelectedIndex(0);
    worksBox.setSelected(false);
    hasKidsBox.setSelected(false);
    ownsPCBox.setSelected(false);
    femaleButton.setSelected(true);
}

The complete class definition is rather lengthy, so we won’t repeat it in the text. It is available in the directory studentSurvey, located in ch17, in nhText.
In this section, we briefly introduce a few additional graphical user interface topics. As usual, details can be obtained from the standard Java API specifications.

### 17.6.1 Menus and menu bars

A *menu* offers a number of options from which the user may choose. Menus are not generally placed with other components in the user interface. Instead, a menu usually appears either in a *menu bar* or as a *popup menu*. A menu bar contains one or more menus and is usually located at the top of the application window. A popup menu is invisible until the user performs a specific action, such as pressing the right mouse button, over a popup-enabled component. The popup menu then appears under the cursor.

A *JFrame* often has a menu bar. A menu bar can be added to a *JFrame* with the method `setJMenuBar`:

```java
JFrame window = new JFrame("Some Application");
JMenuBar menuBar = new JMenuBar();
window.setJMenuBar(menuBar);
```

As shown in Figure 17.23, a menu bar can hold several menus, and each menu can offer a number of menu choices. Menus are created as `JMenu` instances and added to the menu bar:

```java
JMenu batter = new JMenu("Batter");
menuBar.add(batter);
```

Menu choices are modeled with the class `JMenuItem`, and are added to the menu:

```java
JMenuItem swing = new JMenuItem("Swing");
JMenuItem take = new JMenuItem("Take");
JMenuItem bunt = new JMenuItem("Bunt");
batter.add(swing);
batter.add(take);
batter.add(bunt);
```

![Figure 17.23](image.png) **JFrame containing a menubar.**
Chapter 17  Building a graphical user interface

When the user selects an item, the JMenuItem selected generates an ActionEvent. To program menu events, implement an ActionListener for each JMenuItem.

17.6.2 Basic dialogs

A dialog is a window generated by an application as necessary to present information to or gather input from the user. Since dialogs are explicitly created by the application when input or output is needed, their use manifests more of an application-driven style than an event-driven style. Be conservative in your use of dialogs. Excessive dialog use generally implies a poorly designed interface.

Swing provides a library of easy to use dialog windows, as well as classes for building complex dialogs. The classes JOptionPane, JFileChooser, and JColorChooser can be used to create simple, standard dialogues, while JDialog is used to tailor custom dialogs.

Every dialog is dependent on a frame, called its owner. A dialog is destroyed if its owner is destroyed, and disappears from the screen while its owner is iconified.

The dialogs we consider are called modal dialogues. User input to all other windows of a program is blocked when a modal dialog is visible. To create non-modal dialogs, you must use JDialog.

The JOptionPane method showMessageDialog

The class JOptionPane is used to create simple, standard dialogues. The simplest are created with the method showMessageDialog. (There are actually three overloaded methods with this name.) The method displays a simple, one-button, informational dialog window. The most general showMessageDialog specification is:

```
public static void showMessageDialog (Component parentComponent, Object message, String title, int messageType, Icon icon)
```

The parameters are as follows.

- parentComponent – determines the frame on which the dialog depends. The frame containing the parent component (or the parent component, if it is a frame) is the frame owner for the dialog. Its screen coordinates are used to determine the placement of the dialog window. If the argument is null, a default frame is used as owner, in which case the dialog window is usually centered in the screen.
- message – the message to be displayed in the dialog window, typically a String.
- title – the title of the dialog window.
- messageType – an int value indicating the style of message. Possible values are
  - JOptionPane.ERROR_MESSAGE
  - JOptionPane.INFORMATION_MESSAGE
  - JOptionPane.WARNING_MESSAGE
  - JOptionPane.QUESTION_MESSAGE
  - JOptionPane.PLAIN_MESSAGE
17.6 Menus, dialogs, fonts, and graphics

- **icon** – the icon to be displayed in the dialog.

Two variants of the method use default arguments:

```java
public static void showMessageDialog (Component parentComponent, Object message)
```

produces an information message titled “Message,” and

```java
public static void showMessageDialog (Component parentComponent, Object message, String title, int messageType)
```

uses a default icon determined by the message type.

Figure 17.24 shows sample message dialog windows, and the code that created them.

```java
JOptionPane.showMessageDialog(frame,
                           "He went galumphing back",
                           "Important", JOptionPane.INFORMATION_MESSAGE);

JOptionPane.showMessageDialog(frame,
                           "Beware the Jabberwock, my son!",
                           "Important", JOptionPane.WARNING_MESSAGE);

JOptionPane.showMessageDialog(frame,
                           "The vorpal blade went snicker-snack!",
                           "Important", JOptionPane.ERROR_MESSAGE);

JOptionPane.showMessageDialog(frame,
                           "And, has thou slain the Jabberwock?",
                           "Important", JOptionPane.QUESTION_MESSAGE);

JOptionPane.showMessageDialog(frame,
                           "Twas brillig, and the slithy toves",
                           "Important", JOptionPane.PLAIN_MESSAGE);
```

*Figure 17.24*  Message dialog windows.
The `JOptionPane` method `showInputDialog`

The method `showInputDialog` is used to get input from the user. (Again, the method is overloaded; there are six `showInputDialog` methods.) This method gets a `String` from the user, using either a text field or a combo box.

One of the simpler variants of the method is specified as

```java
public static String showInputDialog (Component parentComponent, Object message)
```

The parameters are the same as in `showMessageDialog`. Figure 17.25 shows a typical dialog window produced by the method.

Other variants allow window title, message type, icon, initial value, and combo box options to be specified.

When the user presses the “OK” button, the contents of the text field (or selected item if a combo box is used) is returned. A null value is returned if the user presses “Cancel” or closes the window.

Note the contents are returned as a `String`. If you request a number from the user, you must validate the format and convert the `String` to the appropriate type of value. For example,

```java
int count;
try {
    count = Integer.parseInt(JOptionPane.showInputDialog(frame,"Count:"));
} catch (NumberFormatException e) {
    // user did not key in an integer
    ...
}
```

The `JOptionPane` method `showConfirmDialog`

The `showConfirmDialog` (there are four variants) generates a two or three button window. The two button window provides “Yes” and “No” buttons or “OK” and “Cancel” buttons; the three button window, “Yes,” “No,” and “Cancel” buttons. Sample invocations are shown in Figure 17.26.
The method returns an `int` indicating the user's response. Possible return values include `JOptionPane.YES_OPTION`, `JOptionPane.OK_OPTION`, `JOptionPane.NO_OPTION`, `JOptionPane.CANCEL_OPTION`, and, if the user closes the window, `JOptionPane.CLOSED_OPTION`.

**The JOptionPane method showOptionDialog**

The `showOptionDialog` method generates a two or three button dialog window, similar to a confirm dialog, but with customized buttons. A sample window is shown in Figure 17.27.

The first three parameters of the `showOptionDialog` method are familiar: parent component, message, and window title.

The fourth parameter is option type, just as for the confirm window. This value determines whether a two or three button window will be displayed.

The fifth parameter, message type, determines the default icon for the dialog.

The sixth parameter indicates the icon to be used. Since it is null in this example, the default “question message” icon is used.

The next parameter is an array that determines the button labels. Finally, the last parameter indicates the initialized choice.

The method returns an `int` value just like a confirm window. It does not matter what the buttons have been labeled. The value returned will be either `JOptionPane.YES_OPTION`, `JOptionPane.NO_OPTION`, `JOptionPane.CANCEL_OPTION`, or `JOptionPane.CLOSED_OPTION`.

**JFileChooser and JColorChooser**

Two other standard dialogs are defined by the classes `JFileChooser` and `JColorChooser`. `JColorChooser` presents a pane of controls that allow a user to select and manipulate a
Object[] options =
    {"Of course!","Sorry, no.","Say what?"};

JOptionPane.showMessageDialog(frame,
    "Have you completed all the exercises?",
    "Done Check",
    JOptionPane.YES_NO_CANCEL_OPTION,
    JOptionPane.QUESTION_MESSAGE,
    null, options, options[0]);

Figure 17.27 Option dialog window.

color. JFileChooser is provides a simple mechanism for a user to select a file. (See Figure 17.28.) We briefly look at JFileChooser.

A directory can be specified, either as a String or as a File, when the a JFileChooser is constructed. Three of the constructors are:

public JFileChooser ()
public JFileChooser (String currentDirectoryPath)
public JFileChooser (File currentDirectory)

If no directory is specified, or if the argument is null, a system dependent default directory is used. This is typically the user’s home directory in a Unix environment, or the “My Documents” folder in Windows.

The simplest way to display a dialog is to use one of these methods:

public int showOpenDialog (Component parent)
public int showSaveDialog (Component parent)

The argument in each case determines the dialog’s owner. The first pops up a “save file” dialog; the second, an “open file” dialog. Each dialog has two buttons: “save” and “cancel” buttons in the first case, “open” and “cancel” buttons in the second. The methods return one of the following three integers:

• JFileChooser.APPROVE_OPTION if the “save” or “open” button is pressed;
• JFileChooser.CANCEL_OPTION if the “cancel” button is pressed;
• JFileChooser.ERROR_OPTION if an error occurs or the dialog is dismissed.

The file chosen can be determined by the method

public File getSelectedFile ()

The JFileChooser can be tailored in any number of ways. Details can be found in the standard documentation.
The **JDialog** class

The **JDialog** class is used to create custom dialog windows. A **JDialog** is a top-level window very much like a **JFrame**. However, a **JDialog** doesn’t have caption bar controls allowing the user to iconify or maximize the window. Like a **JOptionPane**, a **JDialog** has an owner, generally a frame, but possibly another dialog. The **JDialog** always appears in front of its owner. If you try to move the owner over the **JDialog**, the owner moves behind instead. A **JDialog** is iconified and deiconified with its owner.

**Figure 17.28**  **JFileChooser** and **JColorChooser** dialogs
A typical constructor (there are eleven) is specified as follows:

```java
public JDialog (Frame owner, String title, boolean modal)
```

The third argument specifies whether the `JDialog` is modal or non-modal. As we mentioned above, input to all other windows is blocked when a modal dialog is visible.

Like a `JFrame`, a `JDialog` delegates management of its components to a content pane. A `JDialog` is constructed in the same way that a `JFrame` is, by adding components to its content pane and adding component listeners to detect user input. A `JDialog` is displayed by invoking its `setVisible` method with an argument of `true`, and is hidden by invoking its `setVisible` method with an argument of `false`.

It is almost always the case that we want to catch or prevent a user’s attempt to close a `JDialog`. Generally, we add a `WindowListener` to handle window closing event. The `JDialog`’s behavior is further controlled by its `setDefaultCloseOperation` method:

```java
public void setDefaultCloseOperation (int operation)
```

The argument is one of three possible values:

- `JDialog.DO NOTHING ON CLOSE` – the close attempt is handled by the `windowClosing` method of a `WindowListener`.
- `JDialog.HIDE ON CLOSE` – the `JDialog` is hidden (`isVisible(false)` is invoked) after any listeners are notified.
- `JDialog.DISPOSE ON CLOSE` – the `JDialog` is hidden and disposed of (`dispose()` is invoked) after any listeners are notified.

### 17.6.3 Fonts

Every Swing component has an associated `font`. A font provides the information needed to display text in that component. A `character` is an abstract representation of an item such as a letter, digit, or punctuation mark. A `glyph` is a shape used to render a character or sequence of characters. A font encapsulates a collection of glyphs and other information needed to render a particular set of characters. Fonts and text rendering can be a very complicated business. We touch only a few points here.

Fonts are modeled with the class `java.awt.Font`. Every component has an associated `Font`, which can be set or retrieved with the methods

```java
public void setFont (Font font)
public Font getFont ()
```

Java distinguishes between `physical fonts` and `logical fonts`. Physical fonts are the actual libraries containing the information necessary to map sequences of characters to sequences of glyphs. The set of available physical fonts is system and implementation dependent.

There are five logical font families supported by Java: `Serif`, `SansSerif`, `Monospaced`, `Dialog`, and `DialogInput`. Logical font names are mapped to physical fonts by the Java run-time environment. The mapping is system and implementation dependent.
A Serif font
A SansSerif font
A Monospaced font

*Figure 17.29 Three font styles.*

The simplest `Font` constructor requires font name, style, and point size as arguments:

```java
public Font (String name, int style, int size)
```

The parameters are:

- **name** – either a logical font name, such as "SansSerif" or a font face name, such as "Helvetica";
- **style** – `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC`, or `Font.BOLD + Font.ITALIC`;
- **size** – the point size of the font.

For example, we can set the `Font` to be used to render a button’s label as follows:

```java
JButton button = new JButton("Help!");
button.setFont(new Font("SansSerif", Font.BOLD, 14);
```

### 17.6.4 Graphics

Every component has a **graphics context** that gives access to the display space occupied by the component. The graphics context is modeled by an instance of the abstract class `java.awt.Graphics`. Since this is an abstract class, it can’t be instantiated. The only way we can get an instance is from a component.

The `Graphics` object provides a large number of methods for drawing text, images, and geometrical figures on components. The methods have names like `drawLine`, `drawImage`, `drawRect`, `drawString`, `fillRect`, etc.

The state of the `Graphics` object includes the current painting color, current font, drawing area, and paint mode. (The paint mode determines how pixels painted in a graphics operation interact with those already drawn.)

To see how this works, let’s draw a triangle on a `JPanel`. (A `JPanel` is a convenient blank canvas on which to draw.) The `JPanel` defines an integer coordinate system with coordinates ranging from `(0,0)` in the upper left to `(width-1, height-1)` in the lower right. Units are pixels.
To draw on a JPanel, we override the method paintComponent:

```java
class TrianglePanel extends JPanel {
    ...
    public void paintComponent (Graphics g) {
    
    }
}
```

This method is invoked in the event dispatching thread to allow the component to update its display.

Note that a Graphics object is passed to the method. This Graphics object will allow us to paint on the JPanel. The Graphics object is initialized to the font and foreground color of the JPanel. The drawing area is the entire JPanel. The default paint mode simply overwrites with the current color. All of these properties can be set by invoking appropriate Graphics methods.

Let’s draw a right triangle with a 100 pixel base and 70 pixel height. We’ll leave a 10 pixels above and to the left of the triangle. We need to draw three lines connecting the points illustrated below.

The method can be written as follows.

```java
public void paintComponent (Graphics g) {
    super.paintComponent(g);
    g.drawLine(10,10,10,80);
    g.drawLine(10,80,110,80);
    g.drawLine(110,80,10,10);
}
```

The first statement invokes JPanel’s paintComponent implementation, which paints in the background. The next three statements draw the lines.

Finally, we should override getPreferredSize and getMinimumSize to make sure that JPanel is large enough to display the triangle.

```java
public Dimension getPreferredSize () {
    return new Dimension(120,90);
}
```

```java
public Dimension getMinimumSize () {
    return this.getPreferredSize();
}
```

1. paintComponent is inherited from JComponent.
17.7 Some class features

We conclude this chapter by summarizing a few additional features of the classes we have been studying. There is no attempt at completeness. Recall that all the Swing component classes are extensions of the java.awt classes Component and Container, and that JFrame is a subclass of the java.awt classes Window and Frame.

17.7.1 Component

**Parent**

The Container that contains the Component. The method getParent returns null if the Component is a top-level window:

```java
public Container getParent()
```

**Enabled**

If the Component is enabled, the user can interact with it and it can generate events. If it is disabled, the user cannot interact with it:

```java
public boolean isEnabled()
public void setEnabled(boolean enabled)
```

**Valid**

The Component is valid if the system knows its size, and in the case of a Container, if it’s properly laid out. If the size of the Component has changed since it was last displayed, the Component is invalid. A Container is also invalid if one of its Components is invalid, or if a Component has been added or removed since the Container was last laid out. A layout manager typically notices an event such as Container resizing, and lays out the Container again.

Invoking a Container’s validate method will cause the Container to lay out its subcomponents again. Invoking a Component’s invalidate method marks the Component and all its ancestors as needing to be laid out.

```java
public boolean isValid()
public void validate()
public void invalidate()
```

**Visible and Showing**

Visible and showing can be somewhat confusing. A Component is showing if the user can find it on the display screen. It does not matter if the Component or window containing the Component is iconified or hidden behind another window.
Chapter 17  Building a graphical user interface

If a Component is visible, then it should be showing when its parent is. A Component can be visible without appearing on the display screen. For a Component to be showing, it must be visible and be contained in a Container that is visible and showing. It is possible for visible to be true, and showing to be false, but not vice versa.

A top-level window is either showing and visible, or not showing and not visible. That is, the visible and showing properties for a top-level window always have the same value. Top-level windows are not visible when created. They must be explicitly made visible. Other Components are visible when they are created.

(There must certainly be a reason behind this distinction, but we admit to being at a loss.)

```java
public boolean isVisible ()
public boolean isShowing ()
public void setVisible (boolean visible)
```

### 17.7.2 Container

**Component Manipulation**

The number of Components in the Container:

```java
public int getComponentCount ()
```

The Component at the specified position. (Note: the first Component is at position 0.)

```java
public Component getComponent (int position)
require:
    0 <= position <= this.getComponentCount() - 1
```

The Component containing a given point. These methods return the Container itself if the point is in the Container, but not in any of the Container’s Components. They return null if the point is not in the Container:

```java
public Component getComponentAt (int x, int y)
public Component getComponentAt (Point p)
```

Remove a Component from a Container:

```java
public void remove (Component component)
public void remove (int position)
```

### 17.7.3 Window

The following are some general Window commands. Resize the Window to the preferred size of its Components, and validate it.

```java
public void pack ()
```
Bring the Window to the display foreground.

```java
public void toFront ()
```

Send the Window to the display background.

```java
public void toBack ()
```

Release a Window’s resources, and remove its peer.

```java
public void dispose ()
```

### 17.7.4 Frame

#### Title

- ```java
  public String getTitle ()
  public void setTitle (String title)
  ```

#### Resizable

The resizable attribute determines whether or not the user can resize the Frame. It must be set before the Frame’s peer is created and the Frame made visible.

- ```java
  public boolean isResizable ()
  public void setResizable (boolean resizable)
  ```

### 17.7.5 JComponent

`JComponent` overrides many `Component` methods, but with the same basic functionality. Some additional features provided by `JComponent` include the following.

#### Ancestors

Get the `JRootPane` ancestor for the component, or the top-level ancestor of the component:

- ```java
  public JRootPane getRootPane ()
  public Container getTopLevelAncestor ()
  ```

#### Transparency

A component is opaque if its background will be filled with the background color. Otherwise, it is transparent:

- ```java
  public void setOpaque (boolean isOpaque)
  public boolean isOpaque()
  ```
17.7.6 JFrame

Default close operation

When a user attempts to close a window representing a JFrame, after the WINDOW_CLOSING event has been delivered to any listeners, the JFrame may hide or dispose of itself on its own accord. The default action is that the JFrame will hide itself: that is, it will become not visible. In this case, the program could later make the frame visible again.

The default response taken by a JFrame to a close action can be determined or set with these methods:

```java
public int getDefaultCloseOperation ()
public void setDefaultCloseOperation (int operation)
```

The argument passed to setDefaultCloseOperation must be one of the following constants:

- JFrame.DO_NOTHING_ON_CLOSE
- JFrame.HIDE_ON_CLOSE
- JFrame.DISPOSE_ON_CLOSE
- JFrame.EXIT_ON_CLOSE

17.8 Summary

In this chapter, we took a first look at Java’s facilities for building an event-driven, graphical user interface. A graphical user interface is constructed from components, such as buttons, text fields, and so on. After displaying the graphical image, an event-driven application waits for an external “event” such as the user pressing a button or entering text in a text field. The application responds to the event, and then waits for the next event.

The graphical components that make up the user interface not only deliver information to the user, but are the sources of the user-generated events that provide input. When the user performs an action such as pressing a button, the component in which the action takes place generates an event. An event listener is an object that responds to or handles a particular kind of event. The event listener registers with the component, and is subsequently informed when the component generates an applicable event. The listener responds by performing some appropriate action.

There are two kinds of components, basic components and containers. Basic, or atomic, components are self-contained entities, that furnish information to or gather input from the user. Buttons and text fields are examples. Containers, on the other hand, hold and position other components. Since containers are themselves components, a container can be a component of another container. Each container has a layout manager to which the container delegates the responsibility for arranging and sizing its components.
The top-level container in a Java application is a JFrame. A JFrame is a window in which the application interacts with the user. A JFrame delegates responsibility for managing its components to its content pane. The graphical display is created by adding components to the JFrame's content pane.

Event handling and updating the appearance of the graphical interface take place in a separate thread called the event dispatching thread. This thread is created when the top-level JFrame is set visible, and is independent of the main thread, in which the method main is executed.

We have introduced several kinds of components and layout managers, and have examined a number of component features. We've seen how to associate event listeners with basic components to handle events generated by the components. It should be obvious that building a graphical user interface is a complex and tedious business. We have touched on only a small fraction of the functionality available in the standard libraries, and have concentrated on the user interface ignoring the model for the most part. In the next chapter, we address issues involved in putting the interface and the model together.

SELF-STUDY EXERCISES

17.1 Indicate whether each of the following statements is true or false:
   a. Every Java Container is a Java Component.
   b. Every Java JComponent is a Java Container.
   c. Saying that a JPanel is the parent of a JButton means that JButton is a subclass of JPanel.
   d. A JFrame is a JComponent.
   e. To say that a component is “lightweight” means that it does not contain other components.
   f. Event handling happens in the event dispatching thread, while component repainting happens in the main thread.

17.2 Write an expression to compute the area (in pixels) of a Component c.

17.3 Suppose that frame is a JFrame. The following code is intended to set JFrame’s layout manager to a FlowLayout, and add a JButton to the JFrame:

   frame.setLayout(new FlowLayout());
   frame.add(new JButton("Press me");

   Is this correct?

17.4 Write code to create a black JFrame, 200 pixels wide and 200 pixels high, titled “Black hole.”

17.5 Write code to define a class BlackButton implementing a black button, 50 pixels wide and 50 pixels high.
Chapter 17 Building a graphical user interface

17.6 If a `BlackButton` is added to a container, will it always be 50 pixels by 50 pixels? Why or why not?

17.7 Write code that will create a `JFrame` 200 pixels wide and 200 pixels high containing a 50 by 50 `BlackButton`.

17.8 Assume `button` is a `JButton`. Write code that creates an anonymous `ActionListener` and registers it with the button. The `ActionListener` should write out the message “The button has been pressed!” to standard output whenever the button is pressed.

17.9 Why do we extend `WindowAdapter` rather than implementing `WindowListener` directly? Can we extend `WindowListener` directly, or must we use `WindowAdapter`?

17.10 Suppose two threads are executing similar code and access the same variable \( x \):

\[
\begin{align*}
\text{thread A} & & \text{thread B} \\
\text{a1. } & \text{assert } x \geq 0; & \text{b1. } & \text{assert } x \geq 0; \\
\text{a2. } & \text{if } (x > 0) & \text{b2. } & \text{if } (x > 0) \\
\text{a3. } & x = x - 1; & \text{b3. } & x = x - 1; \\
\text{a4. } & \text{assert } x \geq 0; & \text{b4. } & \text{assert } x \geq 0;
\end{align*}
\]

Assume that the variable \( x \) is initially 1. One processor executes the threads, alternating between them. Give a possible execution sequence that will cause the assert of line \( a4 \) to succeed, but the assert of line \( b4 \) to fail.

17.11 Look at the code on page 714 for reading an integer from an input dialog. Complete the code so that it will continue to show the input dialog until the user keys an integer.

Exercises

17.1 Using a web browser, read the Java API documentation for the `JFrame` method `pack` (hint: it’s not defined in `JFrame`, it’s inherited), the `JComponent` method `setBorder`, and the `BorderFactory` method `createRaisedBevelBorder`. These classes are all defined in `javax.swing`. Modify the `OnOffSwitch` so that the button has a preferred size of 100 pixels by 100 pixels, and has a raised bevel border. Pack the `JFrame` rather than explicitly setting its size.

17.2 Create a `JFrame` with two buttons, labeled “One” and “Two.” Create a single `ActionListener` to listen to both buttons. When a button is pressed, the `ActionListener` should write “Button one pressed” or “Button two pressed” to standard output.

17.3 Create a `JFrame` containing a single text field. When the user enters text into the text field and presses `Return/Enter`, the text filed is cleared and the text is written to standard output.

17.4 Create a `JFrame` similar to that in the previous exercise, but containing a text field and a button. When the user pressed the button, whatever was in the text field is written to standard output, and the text field is cleared.
17.5 Create a `JFrame` containing a single black button that stays in the center as the `JFrame` is resized. To do this, use a `GridBagLayout`. Create a `GridBagConstraints` object, and set its `gridx`, `gridy`, `anchor`, and `insets` properties. Note that these are public instance variables. Set `gridx` and `gridy` to 0, and set `anchor` to `GridBagConstraints.CENTER`. (Which of these are default values?) The value of `insets` specifies the external padding of the component. Set 50 pixels of pad as follows, were `constraints` is assumed to be the `GridBagConstraints` object:

```java
constraints.insets = new Insets(50, 50, 50, 50);
```

When you add the button, provide the `GridBagConstraints` object as second argument to the `add` command.

17.6 Create a `JFrame` with a 100 pixel by 100 pixel black button in the center, as described in the previous exercise. Create an `ActionListener` for the button. Whenever the button is pressed, the `ActionListener` reduces the button’s preferred (and minimum) size by two pixels.

The trick is to get the button redrawn. One way to do this is to have the `ActionListener` invalidate the button (by invoking the button’s `invalidate` method). This marks the button and all its ancestors as needing to be laid out. Then the `ActionListener` invokes the button’s parent’s `validate` method. This will cause the layout manager to layout the container again.

The next sequence of exercises involves building a simple accumulator that lets users add numbers to a running total. The application interface will a simple window that looks something like the following:

![Accumulator Interface](image)

The user enters a summand by keying into the text field and pressing `Return/Enter`. Pressing the button labeled “+” adds the summand to a running sum, which is displayed in the text field after the button is pressed.

17.7 First, we will build the model for the application. The model is a very simple class specified as follows:

```java
class Accumulator
    A simple object that maintains a running integer sum.

public Accumulator ()
    Create a new `Accumulator`, with sum and summand set to 0.
```
Chapter 17  Building a graphical user interface

```java
public int sum ()
    The running sum.

public int summand ()
    The value to be added to the sum.

public void add ()
    Add summand to sum.

public void setSummand (int val)
    Set the value of the summand.

public void reset ()
    Set the value of the sum and summand to 0.
```

Implement and test this class.

17.8  The interface will be defined by a class `SimpleAccumulatorUI`. This class will extend `JFrame`. It contains a `JLabel`, `JTextField`, and `JButton`. A `SimpleAccumulatorUI` will be given a model instance as a constructor argument. The `main` method creates an `Accumulator` instance, creates a `SimpleAccumulatorUI` instance passing the `Accumulator` as argument, and sets the `SimpleAccumulatorUI` visible.

Implement the class `SimpleAccumulatorUI` and the top-level class containing the `main` method. Include a `WindowListener` to terminate the application, but no other listeners. Make the reference to text field an instance variable, so that it will be able to be accessed other than from its listener. The application should run and display the user interface, but of course nothing happens when you enter data or press the button.

17.9  Add `ActionListener` for the `JButton` and the `JTextField`.

The button listener should do the following when the button generates an `ActionEvent`.

a. Command the model to do an addition.
b. Query the model for the current sum.
c. Display the current sum in the text field.

The `Integer` method `toString` can be used to convert an `int` value to a `String`, and the `JTextField` method `setText` to set the text in the text field.

The text field listener should do the following when the text field generates an `ActionEvent`.

a. Get the number entered by the user in the text field.

   The `JTextField` method `getText` can be used to read the contents of the text field, and the `Integer` method `parseInt` to convert the `String` to an `int`.

b. Set the summand in the model to the value entered by the user.

17.10 Suppose that the text field listener in the above exercise responds to `TextEvents` rather than `ActionEvents`. How will the behavior of the application change?
17.11 The interface above is not particularly “user friendly.” For instance, the user must press Return/Enter before pressing the “+” button. (What happens if a user keys a number into the text field, doesn’t press Enter, and presses the “+” button?)

Without changing the model, design a better interface, with the following properties:

a. The interface has two text fields, one for input and one for output. The field used for output is not enabled.

b. The user need only press the “+” button after entering a number. The Return/Enter key need not be pressed.

c. A “Reset” button is provided to reset the accumulator.

d. An informational dialog is displayed if the user attempts to enter text that is not a decimal integer.

**Self-study exercise solutions**

17.1  
- a. True.
- b. True.
- c. False.
- d. False.
- e. False.
- f. False.

17.2  
c.height * c.width

17.3 No, a JFrame delegates responsibility for managing its components to its content pane. The proper code would be

```java
(frame.getContentPane()).setLayout(new FlowLayout());
(frame.getContentPane()).add(new JButton("Press me"));
```

17.4  
JFrame frame = new JFrame("Black hole");
(frame.getContentPane()).setBackground(Color.black);
frame.setSize(200,200);

17.5  
class BlackButton extends JButton {

    public BlackButton() {
        super();
        this.setBackground(Color.black);
    }

    public Dimension getPreferredSize () {
        return new Dimension(50,50);
    }
}
public Dimension getMinimumSize () {
    return this.preferredSize();
}

17.6 It will not always be 50 pixels by 50 pixels. Its size depends on the layout manager.

17.7 Either change the layout manager of the JFrame’s content pane as in self study exercise 17.3, or add the button to a JPanel which is then added to the content pane. For instance:

    JFrame frame = new JFrame("The button");
    JPanel panel = new JPanel();
    panel.add(new BlackButton());
    (frame.getContentPane()).add(
        panel, BorderLayout.CENTER);
    frame.setSize(200,200);

17.8 button.addActionListener(
    new ActionListener() {
        public void actionPerformed (ActionEvent e) {
            System.out.println("The button has been pressed!");
        }
    });

17.9 We use WindowAdapter because it avoids our having to explicitly implement all seven 
WindowListener methods. We could implement WindowListener directly, if we are willing to implement all seven methods.

17.10 1.  $a_1$ and $a_2$ of thread $A$ are executed; since $x$ is positive, the if condition is true.
2.  $b_1$ and $b_2$ of thread $B$ are executed; since $x$ is positive, the if condition is true.
3.  $a_3$ is executed, changing $x$ to 0.
4.  $a_4$ is executed; the assert condition holds.
5.  $b_3$ is executed, changing $x$ to -1.
6.  $b_4$ is executed; the assert condition fails.

17.11 int count;
    boolean failure = true;
    while (failure)
        try {
            count = Integer.parseInt(
                JOptionPane.showInputDialog(frame,"Count:"));
            failure = false;
        } catch (NumberFormatException e) {}
**Glossary**

*application-driven:* an input-output model in which the application explicitly determines when to get input and when to produce output.

*atomic component:* a *basic component*.

*basic component:* a self-contained component used to present information to or get information from the user.

*component:* a distinct element of a graphical user interface, such as a button, text field, and so on.

*container:* a graphical user interface component that can contain other components.

*dialog:* an informational or data gathering window displayed by an application as needed.

*event:* the occurrence of an action, typically external to the system, that the system is aware of and must respond to.

*event-driven:* an input-output model in which the application waits for an event to occur, responds to the event, and waits for the next event.

*event handler:* an *event listener*.

*event listener:* an object that will be notified when a particular event occurs in a particular component, and that will then take appropriate action in response to the event.

*font:* an entity that encapsulates a collection of glyphs and other information necessary to render a set of characters.

*glyph:* a shape used to render a character or sequence of characters.

*heavyweight component:* a graphical user interface component that has a corresponding (peer) component in the native windowing system of the machine.

*intermediate container:* a container whose purpose is to simplify the organization and positioning of basic graphical components.

*layout manager:* an object responsible for positioning and sizing components in a container.

*lightweight component:* graphical user interface component that is implemented entirely in Java. It has not corresponding (peer) component in the native windowing system of the machine.

*menu:* a pull-down or pop-up list of options from which the user may choose.

*menu bar:* an area, typically at the top of a window, containing one or more menus.

*modal dialog:* a dialog that blocks interaction with all other application windows while it is visible.

*peer:* a native windowing system component, associated with a (heavyweight) Java component.
Swing: a particular collection of library classes and interfaces used to build graphical user interfaces in Java.

thread: an independent sequence of instructions being executed by the processor.

top-level container: a graphical user interface container that is not an element of any other container.