[CHAPTER] 12

Graphical User Interfaces

12.1 Introduction

Probably no area of software development has undergone a more profound change in the last 10 to 20 years than the interface between user and program. Although graphical user interfaces first appeared in the 1970s (see "Missed Opportunities" later in this section), they required high-resolution screens and high-performance processors that were not yet available to the general public. Therefore, virtually all program interactions before the mid-1980s were text based, and users had to learn complex command sequences to manipulate their software. For example, a user would enter the following command to delete a file in UNIX:

```
rm -if /home/user/schneider/myFile
```

To get MS-DOS to list all executable programs on drive A, users would type the following cryptic sequence:

```
A:\DIR *.EXE /P
```

These arcane codes were accepted during the 1960s, 1970s, and 1980s because the overwhelming majority of computer users at that time were technical specialists—computer scientists, engineers, and physicists—who could master these complex control languages.

This all changed with the appearance of personal computers and the rapid growth of e-mail, word processing, and spreadsheets. By the mid-1980s, computing was becoming available to the general public, which wanted to take advantage of these exciting new applications. However, people often had difficulty communicating what they wanted to do. Many were unable to learn the necessary commands to get their programs to work, and they stopped using this new technology.

Apple Computer Inc. understood this, and in 1984 it released the Macintosh operating system, which popularized the use of graphical user interfaces. Suddenly, even novices could interact with computers and get them to do exactly what they wanted. Commands became visual and intuitive, rather than textual and complex. To delete a file, users dropped the file icon into a trash can. To list the files in a directory, they double-clicked a symbol on the desktop. To execute an application, they selected it from a drop-down menu. The impact of the Macintosh system was immediate and enormous. Within a few years, most programs incorporated graphical interfaces into their design.

Today, virtually all software includes an intuitive and easy-to-use visual interface. Users will not tolerate cumbersome, confusing textual interactions. Even if your program is well designed (as we learned in Section 1) and uses efficient data structures and algorithms (as we studied in Section 2), if it is not easy to use it will not be successful.

This chapter investigates the basic concepts of designing and building a **graphical user interface** (**GUI**), pronounced *goo-ee*. We want to emphasize three points before diving in:

1. GUI design is a *huge* topic with an enormous amount of detail, far too much for a single chapter. For example, the Java interface design packages contain dozens of classes and hundreds of methods. Our goal is not to provide encyclopedic coverage but to introduce the basic and fundamental concepts of GUI design. This prepares you to study other ideas that are not treated in this chapter.

2. It is a *volatile* topic. Software developers do not completely agree on which methods belong in a GUI toolkit or exactly how these methods should behave, and the design of a user interface library is still a point of intense debate. The original Java GUI package was the **Abstract Windowing Toolkit** (**AWT**), the standard for the first Java programmers. Later versions of Java included a newer and more powerful package called **Swing** that added functionality and flexibility to the original set of AWT components. In the coming years, even more powerful GUI design packages will be developed, and some existing features will become obsolete and disappear. This chapter focuses on essential concepts that are part of almost every graphical interface package in the marketplace—containers, components, layouts, and events. We present our examples using AWT and Swing, but our focus on fundamental ideas allows you to migrate to any new and improved GUI package that comes along.

3. It is a *conceptually deep* topic. We motivated our study of GUIs by saying that every modern software project must include a nicely designed visual front end. By itself, this is a sufficient reason to study interface design, but this topic also highlights two important concepts in modern software development:

   - *The power of inheritance*—Inheritance is widely used in GUI libraries. Studying how classes are organized and using the classes within the libraries helps you appreciate the contribution of inheritance to software development.
   - *Event-driven programming*—GUIs use a programming style that is quite different from the sequential model typically seen in a first computer science course. In sequential programs, things happen because we reach a statement and execute it. In a GUI, statements are executed not because of their location in the program but because an **event** has occurred, where an event is an action outside the normal sequential flow of control. For example, a user could cause an event by clicking a button, dragging the mouse, or closing a window. Graphical interfaces use a control flow model called **event-driven programming**, which is extremely important in modern software projects.

MISSED OPPORTUNITIES

Xerox PARC (*P*alo *A*lto *R*esearch *C*enter), headquartered in Palo Alto, California, was the main research division of the Xerox Corporation from 1970 until 2002, when it was spun off as an independent entity.

In the 1970s and 1980s, Xerox PARC was an incubator for many new ideas in computer science, and it carried out some of the most innovative and creative computing research in the world. It developed the first personal computer (four years before the Apple II and eight years before the IBM PC), the mouse, the first GUI, Ethernet, and laser printing. Many of these ideas were incorporated into the design of the Xerox Alto, a personal computer developed at PARC in 1973. The Alto, far ahead of its time, had a CRT screen, removable disk storage, three-button mouse, network connectivity, and the first GUI-based operating system interface, complete with icons, menus, and windows. Except for its bulk (it was the size of a small refrigerator), the Alto was similar to a modern desktop machine—but years before the appearance of the Apple II and IBM PC. Two PARC researchers, Butler Lampson and Alan Kay (profiled in Chapter 2), won Turing Awards for their fundamental contributions to computer science.

Sadly, though, Xerox never fully appreciated the business implications of what its scientists had developed, and it failed to commercially exploit their many innovations. Although neither the Alto nor its successor, the Xerox STAR, was financially successful, they did have an enormous impact on the computing industry. In the late 1970s, Steve Jobs, from a tiny startup company called Apple, toured PARC and was enormously impressed with their work on GUIs, user interfaces, and the moveable mouse. He negotiated a deal to exchange Apple stock for visits from Apple engineers and the right to develop a GUI-based personal computer. The result was the Apple Macintosh, which appeared in 1984 and was a huge commercial success. The Mac, as it is affectionately known, is still one of the most important and influential personal computers sold today.

In 1994, Xerox sued Apple for copyright infringement, claiming that Apple had stolen its ideas for the design of the Macintosh user interface. (Ironically, the suit was filed at the same time that Apple was suing Microsoft for violating its copyright on the Macintosh.) The lawsuit was dismissed because Xerox had waited too long to file and the statute of limitations had expired.

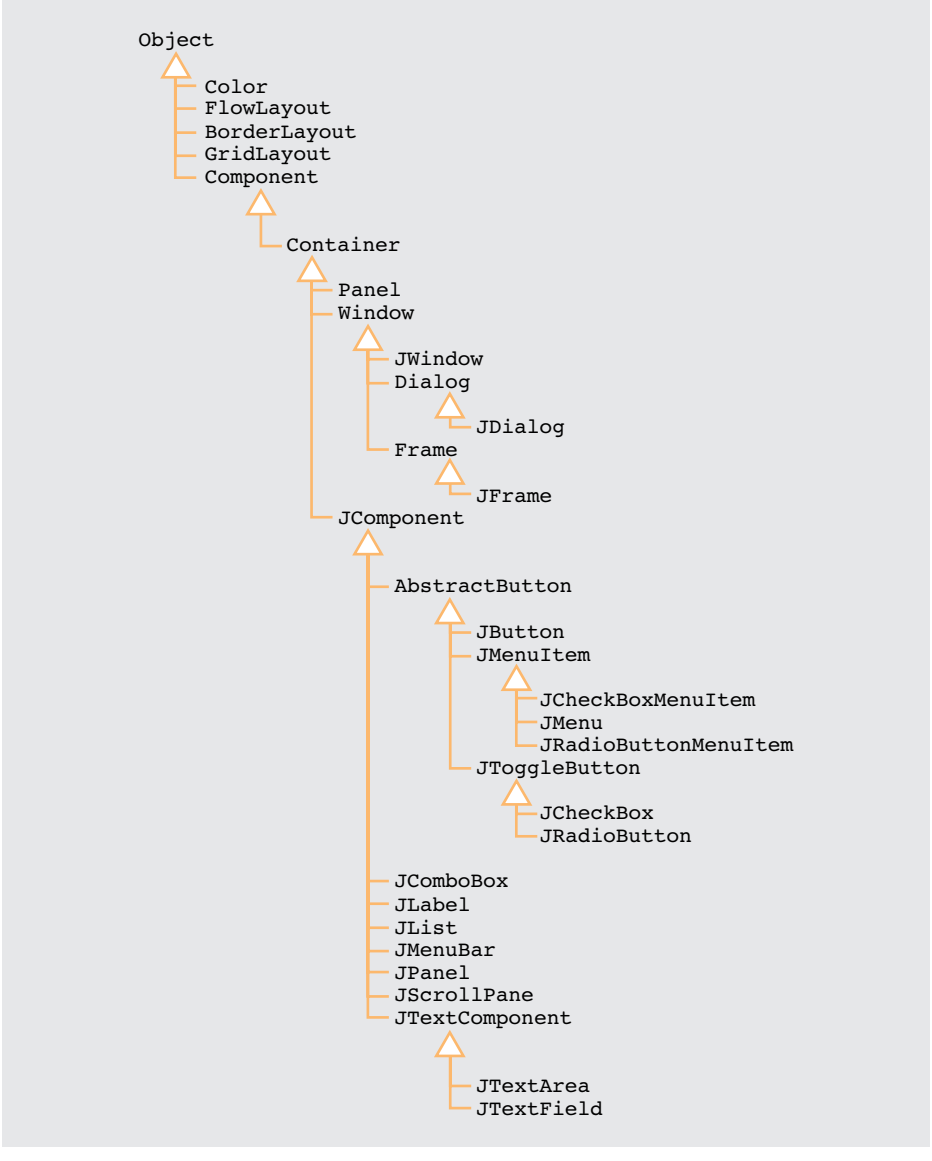
12.2 The GUI Class Hierarchy

12.2.1 Introduction

The package we use to build GUIs is `javax.swing`, usually referred to as **Swing**. It was developed at Sun in 1997 as part of a larger collection of resources called the **Java Foundation Classes (JFC)**, intended to help developers design and build sophisticated applications.

Swing is an extension of AWT, and AWT is located in the package `java.awt`. Swing enhances and extends the functionality of many AWT components. For example, AWT buttons can only have textual labels, but Swing buttons can be labeled using either text or icons. However, the most important difference between the two is that Swing components, such as menus and buttons, are implemented directly in Java without using the capabilities of the processor on which the program is executed. These so-called **lightweight components** allow all Swing objects to be rendered in the same way, producing a uniform look and feel regardless of the platform on which the program is run. AWT components, on the other hand, are mostly **heavyweight**, which means they rely on the services of the local machine to help render an image. Different systems may render in different ways, so the same GUI may have a slightly different appearance depending on the computer. Because AWT relies on the services of the local machine, developers had to take a lowest-common-denominator approach in their design, including only components they were absolutely sure would be correctly rendered by every machine that could run AWT. Because Swing uses only Java code, its designers were free to provide any services they wanted, without concern for what was available on the local machine. Although AWT is still supported in Java 1.5, the current recommendation is to use Swing extensions for interface design whenever possible. We take that approach in this chapter.

The dozens of classes in the `javax.swing` and `java.awt` packages are organized into a class hierarchy, a small part of which appears in Figure 12-1. Classes that begin with *J* are Swing classes. All others are members of AWT. When an identical component type is provided in both AWT and Swing—for example, `Panel` and `JPanel`, or `Window` and `JWindow`—we use the Swing alternative.



[FIGURE 12-1] Portion of the class hierarchy in Swing and AWT

Knowing the class hierarchy in Figure 12-1 is essential, because it determines which methods are available to an object in a user interface. For example, assume that an interface contains a `JFrame` object `f`, where a `JFrame` is a type of rectangular window. The hierarchy

of Figure 12-1 shows that `JFrame` is a subclass of `Frame`, `Window`, `Container`, `Component`, and `Object`. In addition to methods in the `JFrame` class itself, `f` inherits from all its parent classes, resulting in a huge number of accessible methods. For example, a `JFrame` object `f` would inherit all the following:

Inherited from `Frame`:

```
// Specify a title to appear at the top of this frame
public void setTitle(String title);
```

Inherited from `Window`:

```
// Move this frame to the front of the screen, ahead of
// any other visible windows on the screen
public void toFront();
```

Inherited from `Container`:

```
// Add component c to this frame
public Component add(Component c);

// Remove component c from this frame
public void remove(Component c);
```

Inherited from `Component`:

```
// Set height and width of this frame
public void setSize(int width, int height);
// Set the (x, y) screen location of this frame
public void setLocation(int x, int y);
// Set the background color of this frame
public void setBackground(Color c);
// If b is true, make the frame and all components visible;
// otherwise, hide this frame and make it invisible
public void setVisible(boolean b);
```

Inherited from `Object`:

```
// Produce a textual description of this JFrame object
public String toString();
```

The following sections describe many of the classes in the hierarchy of Figure 12-1.

12.2.2 Containers

The first class we examine is `Container`. A **container** is a GUI component used to hold other components. `Container` itself is an abstract class, so it cannot be instantiated directly; instead, you instantiate one of its many subclasses. Every GUI is built from containers into which other components are placed. Multiple containers are often used to build a single interface.

In general, you perform four steps to build a GUI:

1. Create a container object to hold other objects and components, and set the desired properties of the container, such as its size, font, and color.
2. Create a new object (called a *component*) to put into the container, such as a button, menu, or check box.
3. Drop the new component into the container and specify where to locate the component.
4. Go back to Step 2 and repeat until no more components need to be added.

As you can see in Figure 12-1, Swing has a number of `Container` subclasses. The most important are described in the following sections.

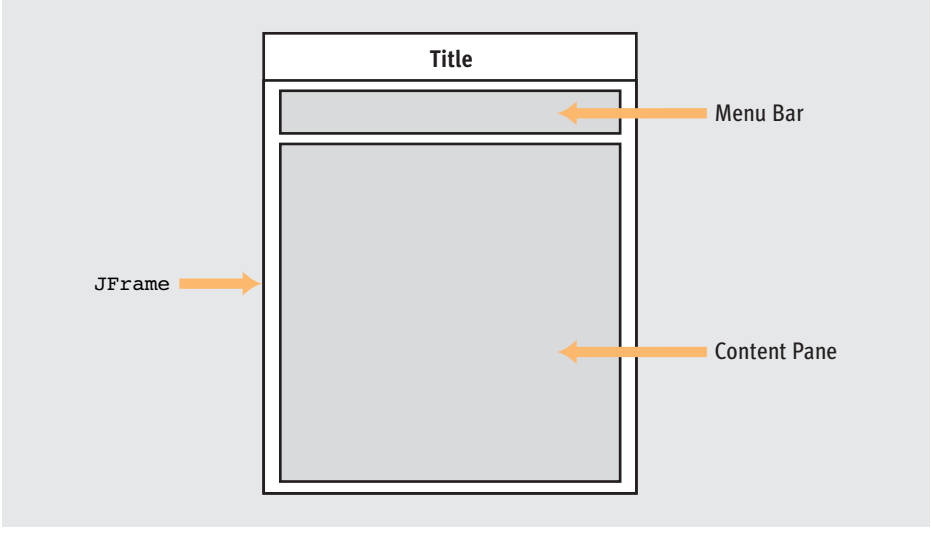
12.2.2.1 `JFrame`

The `JFrame` class represents a resizable application window that is the most widely used `Container` class in Java. An example of a `JFrame` container is shown in Figure 12-2.

[FIGURE 12-2] Example of a `JFrame` container

A `JFrame` includes most features we expect in a GUI container, including a title bar and close, resize, and iconify buttons. Virtually every Java program uses a `JFrame` for its top-level window. All examples in this chapter use `JFrame`.

In addition to the title bar across the top, a `JFrame` object has two sections into which you can place components: the **menu bar** and the **content pane**.



The menu bar, which is itself a `Container`, is where you can place menu options. The content pane, also a `Container`, holds all other nonmenu components (such as buttons, check boxes, and text fields) that can be stored in a `JFrame`. When placing a new component in a `JFrame`, you specify whether it belongs in the menu bar or the content pane.

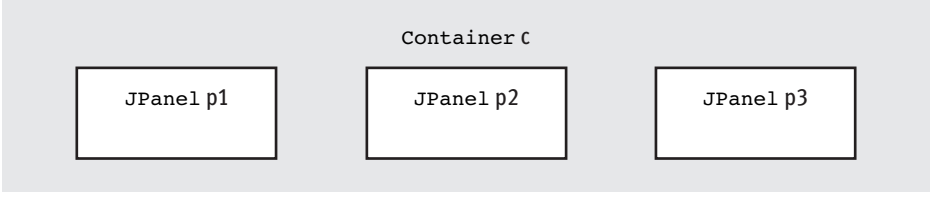
To create a `JFrame` object, you can use a one-parameter constructor in which the parameter specifies the text to display in the title bar. For example, the following declaration creates the `JFrame` object shown in Figure 12-2:

```
JFrame f = new JFrame( "A JFrame" ); // Put "A JFrame" in title bar
```

12.2.2.2 JPanel

This "invisible window" inside a window is a borderless container that you can place inside another container. Because it has no borders, the panel itself cannot be seen; only the components inside the panel are visible. Panels are often used to simplify the process of building interfaces. You can create a panel, place components into it, and then drop the entire panel and all its objects into another container. This divides the job of constructing a single large container into smaller tasks of building individual panels—another example of the "divide and conquer" strategy mentioned in Chapter 1.

This strategy is diagrammed in Figure 12-3, which shows a single `Container` object `c` and three `JPanel` objects `p1`, `p2`, and `p3`. (In this figure, the borders are shown as solid lines to highlight the existence of the three panels. However, these borders would not actually be visible; you would only see the components inside the panels.)



[FIGURE 12-3] Example of using panels to simplify the creation of containers

12.2.2.3 JDialog

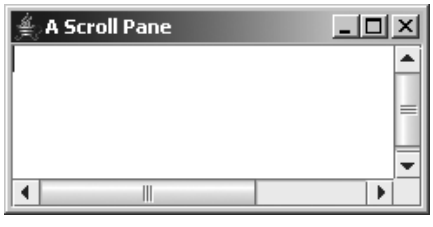
`JDialog` is a pop-up dialog box that displays special information or warning messages for the user (see Figure 12-4).

[FIGURE 12-4] Example of a `JDialog` container object

A `JDialog` container can be **modal**, which means it blocks user input to other windows until the user reads the information in the dialog box and then closes it by clicking OK, as in Figure 12-4. You often see this type of container when there is an error or unusual event. However, this container is not limited to warnings; a `JDialog` object is a fully functional container that can hold any component.

12.2.2.4 JScrollPane

`JScrollPane` is a container that is "attached" to a larger container object, called the **client**, that is too big to display all at once. When a container such as a `JFrame` is too large to fit on the screen, you can create a `JScrollPane` object that is roughly the size of the screen's visible portion and attach it to the larger container. `JScrollPanes` allow the user to move horizontally and vertically through the larger container using a rectangular cross-section called a **viewport**. Figure 12-5 shows an example of a `JScrollPane` object.

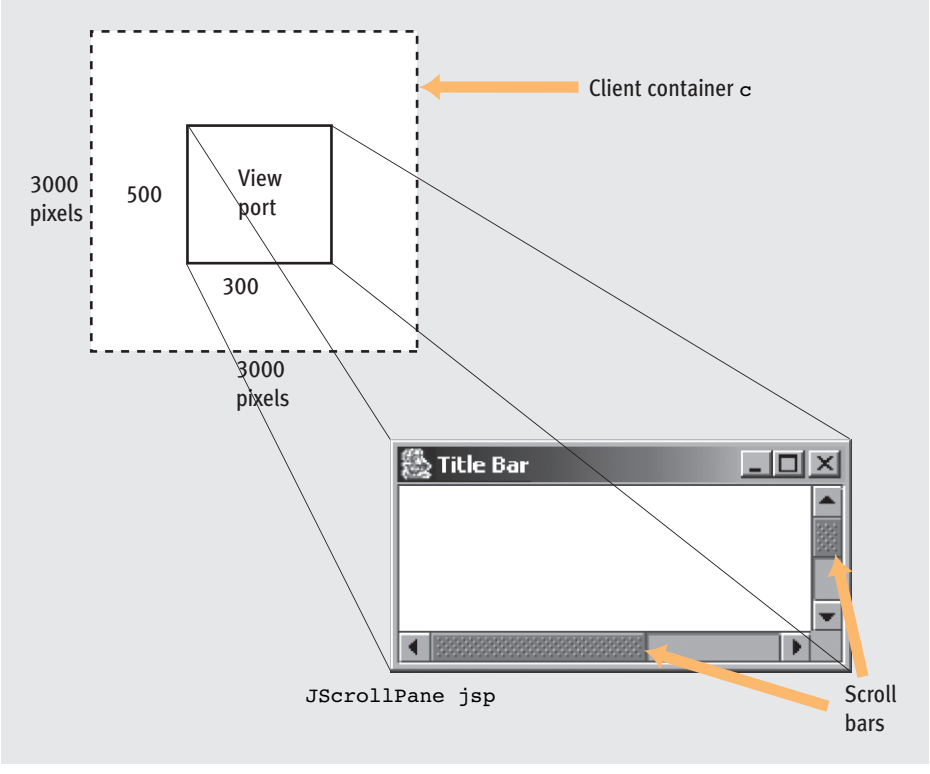


[FIGURE 12-5] Example of a `JScrollPane` object

For example, assume that container `c` is 3000 × 3000 pixels, which is too large to display on most screens at one time. To deal with this, we create a `JScrollPane` object, `jsp`; for example, we can make it 500 × 300 pixels in size, attach it to `c`, and use `jsp` to view a rectangular 500 × 300 pixel "slice" of `c`. To attach a scroll pane to a specific client, you use the scroll pane class constructor in the following way:

```
// c is the client of the scroll pane object jsp
JScrollPane jsp = new JScrollPane(c);
```

If the `JScrollPane` is smaller than the container to which it is attached, then horizontal and vertical scroll bars are added automatically. The scroll bars shown in Figure 12-5 are used to move the viewport, and they determine which part of the larger container appears on the screen. The relationship between the `JScrollPane` object `jsp` and its larger container `c` is shown in Figure 12-6.



[FIGURE 12-6] Relationship between scroll pane objects and the object to which they are attached

Although every GUI library has its own set of container classes, the concept of a container is universal. The containers you encounter in other packages will almost certainly be similar to those just described—regular windows (windows, frames, dialogs), borderless windows (panels), and scrolling windows (scroll panes). The examples that follow make the most use of the `JFrame` and `JPanel` container classes.

12.2.3 Layout Managers

The previous section outlined the basic steps of designing a GUI:

1. Create a container.
2. Create a component.
3. Place the component into the container.

After reading Step 3, you might wonder where the component goes. Specifically, how do you control the placement of component objects inside a container? This is called the **layout problem**; it addresses such issues as positioning components in a window, determining the amount of space between components, and resizing components to make them fit, if necessary.

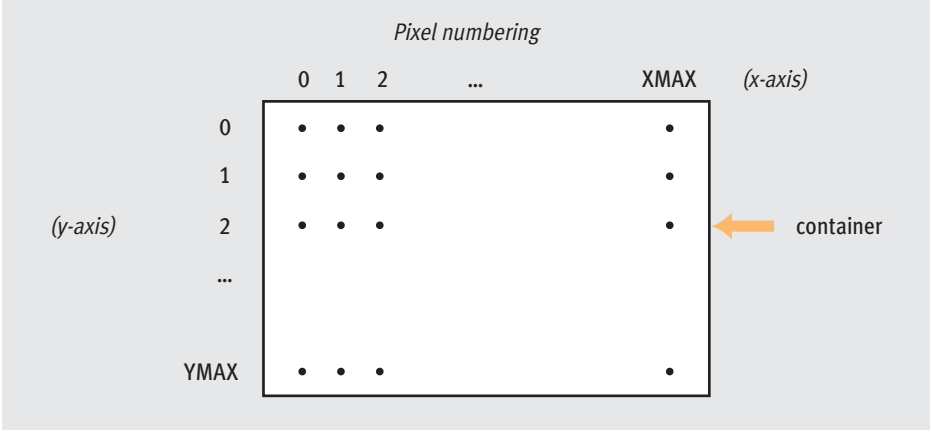
12.2.3.1 Handling Layouts Yourself

There are two ways to handle the layout problem. First, you can perform all layout operations yourself. That is, each time you create a component, you specify exactly where it should go. Every Swing component inherits from the `Component` class method `setLocation()`:

```
public void setLocation(int x, int y);
```

The (x, y) location is specified in terms of the horizontal and vertical distance, in pixels, from the container's origin point (0, 0). To position a component, you provide the (x, y) coordinates of its top and far-left pixel. Knowing the location of this point and the component's overall size and shape allows you to determine its proper placement within the container.

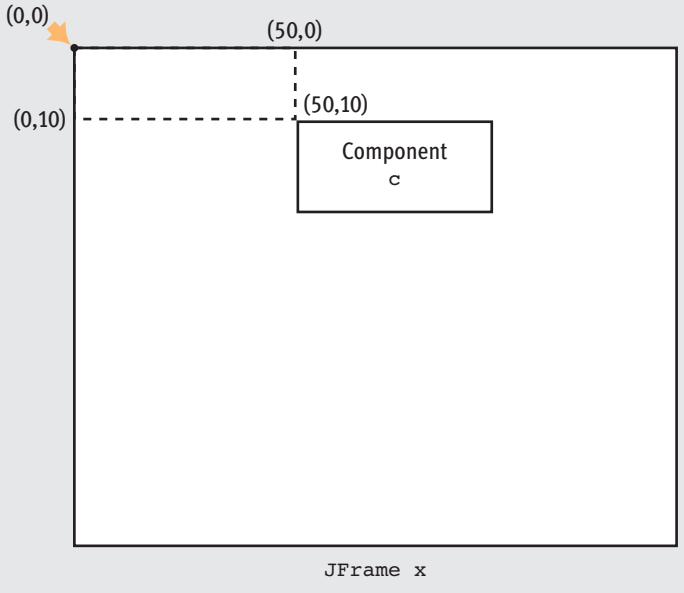
Pixel numbering begins at (0, 0) in the upper-left corner of the container. The first number, the x-axis value, increases as you move right, and the second number, the y-axis value, increases as you move down. This numbering scheme is diagrammed in Figure 12-7.



[FIGURE 12-7] Pixel numbering scheme used by containers

Figure 12-8 shows a component `c` placed at position (50, 10) within a `JFrame` container `x`. The point (50, 10) is the location of the top left pixel of `c`. We specified the position of `c` using the following command:

```
c.setLocation(50,10);
```



[FIGURE 12-8] Component placement using the pixel numbering scheme of Figure 12-7

After specifying the position of the upper-left pixel of component `c`, we add it to the `JFrame` container `x` using the `add()` method inherited from `Container`. Here is how we add component `c` from Figure 12-8 to container `x`:

```
x.add(c);     // Add c to the content pane area of JFrame x.
              // It goes into the position specified by the
              // setLocation method, which is position (50, 10).
```

The values `XMAX` and `YMAX` in Figure 12-7 specify the width and height of the container. You can set these values using the following method:

```
public void setSize(int XMAX, int YMAX);
```

which is inherited by all container objects from the AWT `Component` class.

The following code creates a `JFrame` object `f` with a light gray background, dimensions of 300 by 400 pixels, and the text *Example* in the title bar. It then makes the frame visible on the screen.

```
// Put "Example" into the title bar
JFrame f = new JFrame( "Example" );

// Set f to 300 pixels wide by 400 pixels high
// with a light gray background color
f.setSize(300, 400);
f.setBackground(Color.LIGHT_GRAY);

// Make the frame visible on the screen
f.setVisible(true);
```

To place a button object `b` at position (50, 10) within this new `JFrame` object `f`, we do the following:

```
// Assume that b is a button object. We see how
// to create this component later in the chapter.

// Set button size to 70 pixels wide by 90 pixels high
b.setSize(70, 90);

// Position upper-left corner of b to (50,10)
b.setLocation(50, 10);

// Now put b into the content pane
// portion of the JFrame created above
f.add(b);
```

This code places the rectangular button object into pixel locations [50…120] on the x-axis and [10…100] on the y-axis. You repeat this explicit sizing and placement process for every component you put into `f`.

As you can imagine, this is a difficult and cumbersome process, because you may have dozens of components to size, locate, and add. Getting them all into the correct position without overlapping can take a good deal of time, and can require design skills that may not be the strong suit of many programmers. Also, if you resize the container, it may clip off some of the components placed in the window. Clearly, we need an easier way to handle the layout problem. The next section provides the solution.

SCIENCE MEETS ART AND DESIGN

Designing a functional, elegant, and easy-to-use graphical interface requires a great deal of technical computing knowledge. However, good GUI design also requires aesthetics and graphics design skills—topics that are unfamiliar to most computer science students.

Graphics design is the study of communicating information through visual media. A graphics designer studies the basic elements of visualization, such as form, shape, color, mass, texture, shading, and light, and learns how these elements affect the interpretation, mood, and message of the final image. Graphics design is extremely important in any visual aspect of computing, including GUIs and Web pages. Computer science students might create interfaces that are technically correct, but if they have no knowledge of artistic design, the interfaces may be inelegant, boring, and cumbersome to use. Conversely, a designer with no knowledge of computing may produce Web pages that are flashy and filled with eye-popping design features but which lack essential functionality, such as the ability to interface the Web page to an SQL database.

In the past, when computer science students asked which courses would be most useful in supporting their major, the answer was invariably mathematics and physics. Other suggestions were psychology for AI majors, linguistics for compiler and programming language specialists, or business for students going into MIS. Today, however, graphics design is one of the hottest supporting areas in computer science. The artistic and aesthetic skills that students acquire are useful in many technical areas, including GUIs, Web design, digital imaging, game design, font design, document layout, and virtual reality. The days of the computer science "tekkie" with little or no design skills are fast coming to an end.

12.2.3.2 Using a Layout Manager

The second approach for handling the layout problem is to use a layout manager. This approach is more popular by far, and we will use it to build our GUIs in this chapter.

A **layout manager** is an object that automatically positions and resizes objects within a container. It provides platform-independent layout services that help you create a GUI. You can concentrate on the conceptual, high-level design aspects of the interface and leave such messy details as sizing and placement to the layout manager.

The `LayoutManager` interface specifies a set of methods for managing the way objects are arranged within a container. Every container object in Java is assigned a default layout manager; you can either use it, specify an alternative layout manager yourself using the container's `setLayout()` method, or explicitly provide the location of components yourself

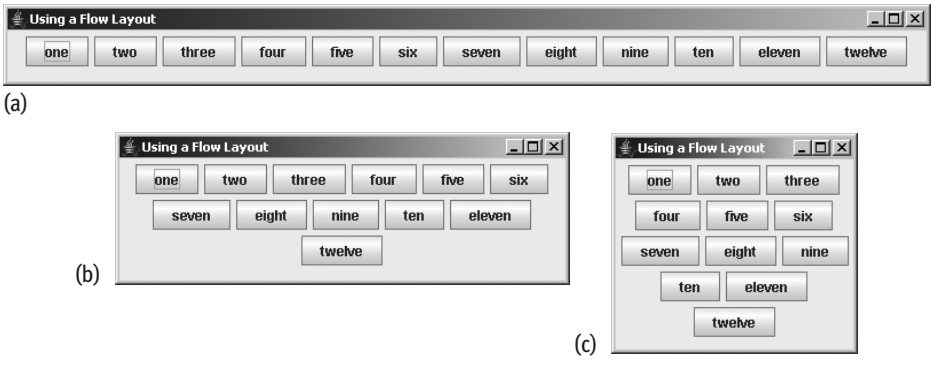
using the techniques described in the previous section. If you use a layout manager, you should decide which one to use before adding any components. If you do not specify a layout manager, the default manager will be used, and it may not provide the proper placement of components.

Dozens of classes implement the `LayoutManager` interface, and this section describes three of the most popular. The first, called a **flow layout**, places components into a container in strict left-to-right, top-to-bottom positions in the order you add them to the container. (You can reset the motion to right-to-left if you want.) In other words, a flow layout places components into positions 1, 2, 3, 4, 5, 6, …, as shown:

container f

position 1	position 2	position 3	position 4
position 5	position 6	position 7	position 8
…			

The number of components that can fit into a single row depends on their size as well as the width of the container. Layout managers also have the following useful feature: if you resize the container, the position of all components is automatically recomputed based on the container's new height and width. We can see this property in Figure 12-9, which shows a window containing 12 buttons labeled *one*, *two*, *three*, and so on. As the size of the window is changed, the layout manager determines how many buttons can fit on a single line and determines the proper location of each.



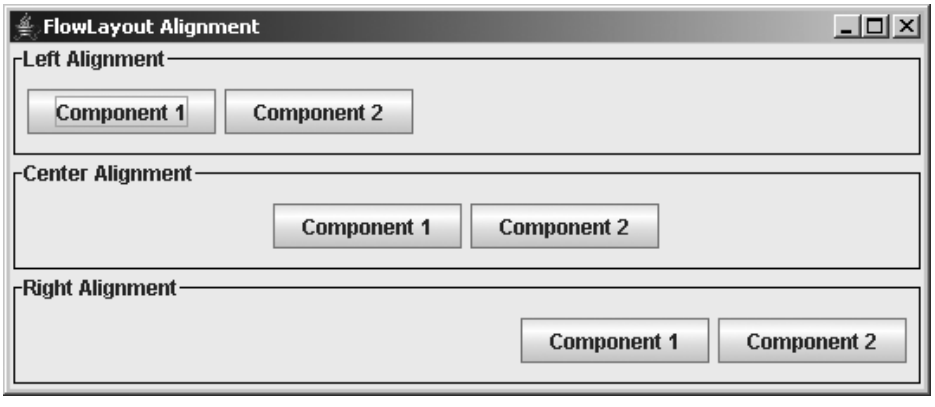
[FIGURE 12-9] Using a flow layout

To associate a flow layout with a container `f`, we use the `setLayout(LayoutManager lm)` method:

```
// Container f is using a flow layout manager
f.setLayout( new FlowLayout() );
```

When `f` is a `JPanel`, the preceding statement is unnecessary because `FlowLayout` is the default layout manager for `JPanel` objects.

In addition to the default constructor `FlowLayout()`, you can use the one-parameter constructor `FlowLayout(int align)`, which specifies how to align objects in a row that is not yet filled. For example, you could use this constructor to align the last row in Figure 12-9(c), which holds a single component. The alignment value can be one of the three static constants `LEFT`, `CENTER`, or `RIGHT` in the `FlowLayout` class. If we assume that a row can hold four components, and it currently has only two, then these three constants behave as shown in Figure 12-10.



[FIGURE 12-10] Alignment within a flow layout manager

Another layout manager is the **grid layout**, which divides a container into a rectangular grid with *m* rows and *n* columns. For example, a grid layout manager with four rows and five columns is divided as follows:

(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)
(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)

You specify the number of rows and columns in a grid layout using the following two-parameter `GridLayout` constructor:

```
GridLayout( int rows, int columns );
```

The container object is divided into the specified number of rows and columns, and each grid rectangle has the same size. Thus, to create the 4×5 grid shown above, you use the following code:

```
f.setLayout(new GridLayout(4, 5));
```

However, there will be a problem if you ever attempt to add more than $m \times n$ components to the container, because there is not enough room. For example, the preceding 4×5 grid layout has room for 20 components; trying to add a 21st causes a fatal error. For this reason, the grid layout behaves in a different way from the flow layout.

When the number of rows and columns in the `GridLayout` constructor are both greater than zero, the value you specify for the number of columns is actually ignored. Instead, the number of columns is determined dynamically from the number of rows you specify and the total number of components you add to the container.

For example, if you declare a grid layout with four rows and two columns, you only have room for eight components. If you then added 12 components to the container, they would appear in a grid of four rows of *three* columns each, rather than two. The grid layout manager automatically increases the number of columns to accommodate the total number of components, using the following formula:

Number of columns = Total number of components / Number of rows

So, if we added 20 components, the layout manager would use 20/4 = 5 columns. Because of this feature, programmers simply enter a 0 in the column field of the grid layout manager's constructor.

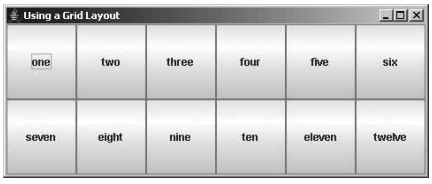
The grid layout manager places the first component at position (0, 0) of the grid. The next object is placed in the next available slot in the current row, position (0, 1), and the layout manager moves right one position. When all the columns of a single row are filled, the layout manager moves to column 0 of the next row and fills it in the same left-to-right fashion. Because all slots in the grid are the same size, the layout manager resizes the components as needed to fit into the available area.

To associate a grid layout with a container object `f`, we do the following:

```
// A GridLayout with m rows and n columns
// usually n is set to 0 because the layout manager
// automatically determines the proper number of columns.
f.setLayout(new GridLayout(m, n));
```

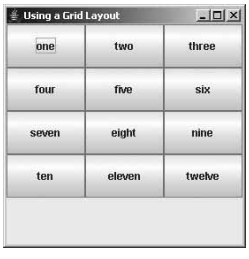
Figure 12-11 shows several examples of grid layouts that contain 12 components. The number of columns actually created is not necessarily the number declared. All the grid elements are the same size, and this size is recomputed as the dimensions of the container change.

(a) `gridLayout( 2, 6 )`



(b) `gridLayout( 2, 6 )`

(c) `gridLayout( 2, 0 )`



(d) `gridLayout( 5, 0 )`

[FIGURE 12-11] Examples of a grid layout

The final layout manager we present is the **border layout**. This layout divides the container into five regions, called north, south, east, west, and center. Each region can hold a single component, although you can use a `JPanel` as the component and place other components within it. The five areas in the layout are defined by the static constants `NORTH`, `SOUTH`, `EAST`, `WEST`, and `CENTER` in the `BorderLayout` class. The north and south regions represent the top and bottom parts of the container, respectively. They extend the full width of the container from left to right, and are exactly as high as the component placed inside them. The west and east regions represent the left and right sides of the container, respectively. They extend vertically from the north region to the south, and are exactly as wide as the component placed inside them. The center region takes up the remaining space, and is resized to be as large as possible.

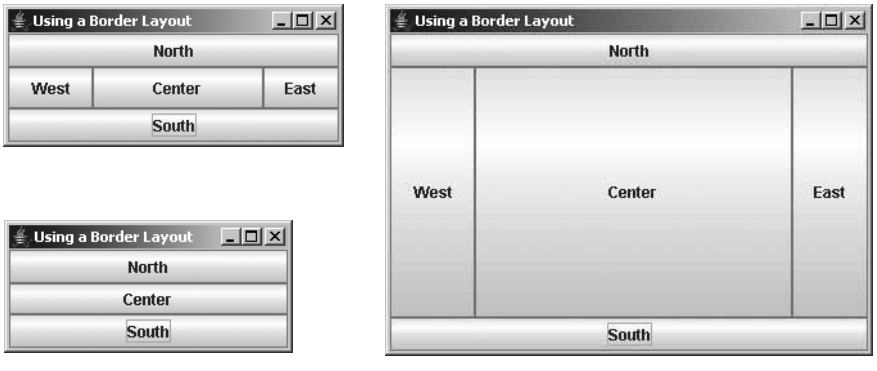
When adding a component to a container `f` using a border layout, you must use the following `add` command to put the component in a specific region:

```
f.add(regionName, component)    // Put component into
                                // the specified region
```

For example, you might want to place five buttons named North, South, East, West, and Central into a `JFrame` object named `win` that uses a border layout. Furthermore, you want to place these buttons in their correspondingly named regions. You place the buttons in the following way:

```
// Create a JFrame called win
JFrame win = new JFrame( "Using a Border Layout" );

// Select a border layout
win.setLayout(new BorderLayout());

// Now add buttons to each region of the screen;
// we learn how to create these buttons in the next section
win.add( BorderLayout.NORTH, new JButton( "North" ));
win.add( BorderLayout.SOUTH, new JButton( "South" ));
win.add( BorderLayout.EAST, new JButton( "East" ));
win.add( BorderLayout.WEST, new JButton( "West" ));
win.add( BorderLayout.CENTER, new JButton( "Center" ));
```

Depending on the size of the five buttons, the preceding operations produce a frame like the ones shown in Figure 12-12. You can also leave a region empty. For example, the WEST and EAST regions are not used in the lower-left window in Figure 12-12, and are therefore not visible.



[FIGURE 12-12] Border layout

We have now reached the point where we can write a program to build our first GUI, although it is extremely simple and has no functionality. The program (see Figure 12-13) follows the steps outlined earlier in this chapter for creating a GUI:

- Create a container and set its properties.
- Create a component and set its properties.
- Place the component in the container.

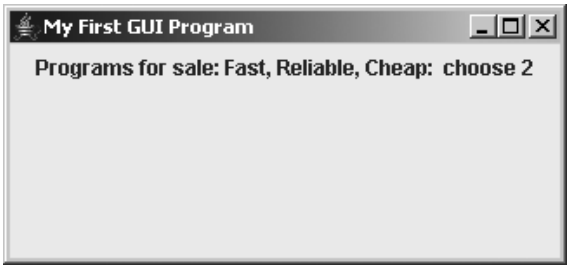
```
/**
 * Display a JFrame with a simple message
 */

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

public class SwingFrame    {
    // The size of the frame
    public static final int WIDTH = 325;
    public static final int HEIGHT = 150;

    public static void main( String args[] ) {
        // Create the frame
        JFrame win = new JFrame( "My First GUI Program" );

        // Establish the layout, size, and color of the frame
        win.setLayout( new FlowLayout() );
        win.setSize( WIDTH, HEIGHT );
        win.setBackground( Color.LIGHT_GRAY );

        // Put the message in the frame
        String msg = "Programs for sale: Fast, " +
                     "Reliable, Cheap:  choose 2";
        JLabel label = new JLabel( msg );
        win.add( label );

        // Make it visible
        win.setVisible( true );
    }

} // SwingFrame
```

[FIGURE 12-13] Example of a simple graphical user interface

In Figure 12-13, we use the class `JFrame` to create `win`, the top-level window of our GUI. An instance of `JFrame` is commonly used to create the main window for a Java application, because it includes all the functionality we want in a window. Next we set the layout

manager to `FlowLayout` because the default layout manager for a `JFrame` is `BorderLayout`. We set the properties of `win`, such as its size (325 by 150 pixels) and its background color (light gray). Finally, we add our single component, a `JLabel`, and make the window and its contents visible on the screen.

The result of running the program in Figure 12-13 is shown in Figure 12-14.



[FIGURE 12-14] Our first graphical user interface

Clearly, this is not an exciting interface. The next section describes more interesting components, such as buttons, menus, and text fields.

12.3 GUI Components

Components are items that you put into containers. They are objects you can display on the screen and that can interact with the user. (Although it is technically a container, `JPanel` is also a component and can be placed inside other containers.) AWT and Swing have far too many components to discuss in this section. Instead, we describe some interesting and widely used components to help you understand their behavior. You can then read more on the Web about components that are not discussed here. (A good place to start is the Java tutorial on Swing components at *http://java.sun.com/docs/books/tutorial/uiswing/components/components.html*.)

The nine Swing component types we discuss and use in our examples are summarized in Figure 12-15.

COMPONENT	FUNCTION
`JButton`	A push button that triggers an event when clicked
`JComboBox`	A labeled button-like component that displays a drop-down list of items when clicked; the user can select one of these items
`JCheckBox`	An on/off selection box that a user can toggle; a checkmark appears in the box when it is selected
`JList`	A list from which the user can select one or more items
`JLabel`	A text string that is displayed on the screen
`JTextField`	A one-line area where text can be entered, displayed, and edited
`JTextArea`	A multiline area where text can be entered, displayed, and edited
`JMenuBar`	A menu bar that can be placed in the menu pane of a container
`JRadioButton`	A set of buttons, only one of which can be selected at one time. Selecting one button deselects the others.

[FIGURE 12-15] Examples of Swing component types

Although there are many different types of components, the steps required to create them and add them to a container are similar. The first step is to create a new instance of the component using a constructor from the appropriate class. The constructor may or may not take parameters, depending on the component being instantiated. Figure 12-16 lists some of the available constructors for the components in Figure 12-15. You can find the full list of constructors in the online Java documentation under the corresponding class name.

COMPONENT	CONSTRUCTOR	ACTION
`JButton`	`JButton();`	Create a blank button
	`JButton(String s);`	Create a button labeled with text label `s`
	`JButton(Icon i);`	Create a button labeled with icon `i`
`JComboBox`	`JComboBox();`	Create a drop-down list with no items
	`JComboBox(Object[] i);`	Create a drop-down list that initially contains all the elements in array `i`

continued

`JCheckbox`	`JCheckbox();`	Create an unselected check box with no text
	`JCheckbox(String s);`	Create an unselected check box with label `s`
	`JCheckbox(Icon i);`	Create an unselected check box with icon `i`
`JLabel`	`JLabel(String s);`	Create a label with text `s`
`JList`	`JList();`	Create an empty list
`JTextField`	`JTextField(String s);`	Create a text field containing text `s`
	`JTextField(int n);`	Create a blank text field `n` columns wide
	`JTextField(String s, int n);`	Create a text field `n` columns wide containing text `s`
`JTextArea`	`JTextArea(String s);`	Create a text area containing text `s`
	`JTextArea(int r, int c);`	Create a blank text area containing `r` rows and `c` columns
	`JTextArea(String s, int r, int c);`	Create a text area with `r` rows and `c` columns containing text `s`
`JMenuBar`	`JMenuBar()`	Create a blank menu bar
`JRadioButton`	`JRadioButton()`	Create an initially unselected radio button with no set text

[FIGURE 12-16] Sample constructors for the components in Figure 12-15

Once we instantiate the new component, the second step is to set the component's **attributes**, also called its **properties**. A number of properties can be applied to all components; you can find them in the AWT `Component` class. Some of the more important properties are:

Appearance properties:

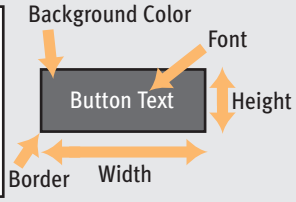
```
// Set the background color.
setBackground(Color c);
// Set the foreground color.
setForeground(Color c);
// Set the type font for any text in the component.
setFont(Font f);
// Set the border (edge) type of this component.
setBorder(Border b);
// Set the width (w) and height (h).
setSize(int w, int h);
// Set the (x,y) position of the component's upper-left pixel.
setLocation(int x, int y);
```

Other properties:

```
// Make the component visible or not visible on the screen.
setVisible(boolean b);
// Enable or disable (gray out) this component.
setEnabled(boolean b);
```

Every component has default values for these properties; you only need to reset the properties for which the default values are inappropriate. All the default values are specified in the Java AWT and Swing documentation. For example, Figure 12-17 illustrates some of these properties using a button labeled `theButton`.

```
JButton theButton = new JButton( "Button Text" );
theButton.setBackground( Color.BLUE );
theButton.setFont(
  new Font( null, Font.BOLD, 25 ) );
theButton.setBorder(
  BorderFactory.createLineBorder(Color.BLACK, 5));
theButton.setSize( 100,30 );
```



[FIGURE 12-17] Examples of component properties

Because this book uses a limited color palette, some colored GUI components in this chapter appear in black and white.

In addition to the general properties, other properties are unique to a particular component type, and the methods that handle them are included in the class for that component. Figure 12-18 contains a sample of properties associated with a particular component type.

CLASS	METHODS	PURPOSE
`JButton b`	`String s = b.getText();`	Return the label on this button
`JComboBox b`	`Object o = b.getSelectedItem();`	Return the currently selected item in this combo box
`JLabel l`	`Component c = l.getText();`	Return the text associated with this label
`JTextField f`	`int i  = f.getColumns();`	Return the number of columns in this text field

[FIGURE 12-18] Examples of specific component properties

After we have set the properties of our component (whether general to all components or specific to this type), we can add it to our container. If the container `f` is an instance of `JFrame`, we add the new component to the `JFrame` pane using one of the `add()` methods in the `Container` class. For example:

```
f.add(Component c);              // Add component c to the
                                 // JFrame container f
f.add(String S, Component c);    // Add component c to region S
                                 // of the JFrame f; this
                                 // assumes that f uses
                                 // a border layout
```

Figure 12-19 shows an extension to the program in Figure 12-13 that adds some components we introduced in this section.

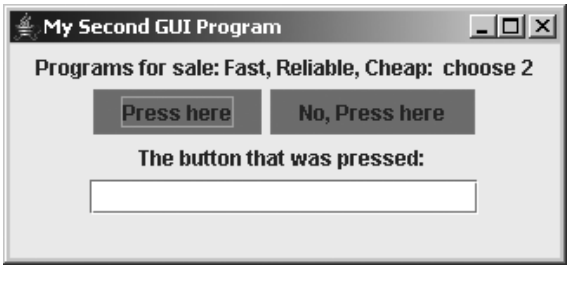
```
/**
 * Display a JFrame with a simple message and a few buttons
 */

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

public class SwingFrame2 {
    // The size of the frame
    public static final int WIDTH = 325;
    public static final int HEIGHT = 150;

    // The length of the text field
    public static final int FIELD_LENGTH = 20;

    public static void main( String args[] ) {
     // Create the frame
     JFrame win = new JFrame( "My Second GUI Program" );

     // Establish the layout, size, and color of the frame
     win.setLayout( new FlowLayout() );
     win.setSize( WIDTH, HEIGHT );
     win.setBackground( Color.LIGHT_GRAY );

     // Put the message in the frame
     String msg = "Programs for sale: Fast, " +
                  "Reliable, Cheap:  choose 2";
     JLabel label1 = new JLabel( msg );
     win.add( label1 );
```

continued

```
        // Add some more components to make things interesting
        JButton pushButton1 = new JButton( "Press here" );
        pushButton1.setBackground( Color.RED );
        win.add( pushButton1 );

        JButton pushButton2 = new JButton( "No, Press here" );
        pushButton2.setBackground( Color.RED );
        win.add( pushButton2 );

        String msg2 = "The button that was pressed: ";
        JLabel label2 = new JLabel( msg2 );
        win.add( label2 );

        JTextField field = new JTextField( FIELD_LENGTH );
        win.add( field );

        // Make it visible
        win.setVisible( true );
      }

} // SwingFrame2
```

[FIGURE 12-19] A graphical user interface with additional components

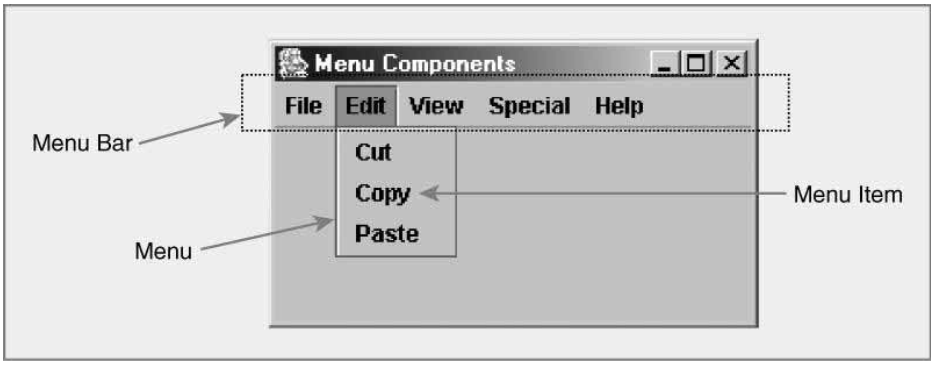
The program in Figure 12-19 includes four new components: two red buttons labeled *Press here* and *No, press here*, a label containing the text *The button that was pressed:*, and a 20-column text field, which is blank. Because we are using a flow layout manager to control positioning, these four components are added to the `JFrame` object in left-to-right sequence. The program in Figure 12-19 generates a graphical interface that looks like Figure 12-20.



[FIGURE 12-20] The GUI produced by the program in Figure 12-19

The last component we present in this section is a menu, a part of virtually every user interface. Unlike the other components we have described, the menu is not added to the content pane section of a `JFrame`, but is added to the menu bar section using the `setJMenuBar()` method in the `JFrame` class.

Menus are more complex than the examples shown earlier. Menus have three parts, and each must be correctly created and placed in the right order. These three parts are the **menu bar**, the **menu**, and the **menu item**. The relationship between these parts is shown in Figure 12-21.



[FIGURE 12-21] The three parts of a menu component

The first step in creating a menu is to create an empty menu bar and place it in the menu bar section of the frame. You do this using the constructor in the `JMenuBar` class, and then you add the new menu bar object to your frame with the `setJMenuBar()` method.

```
JMenuBar mb = new JMenuBar();   // Create an empty menu bar mb
f.setJMenuBar(mb);              // Put it into the menu bar
                                // section of JFrame f
```

The second step is to add the names of the pull-down menus that appear in this new menu bar. For example, in Figure 12-21, the five menu names on the menu bar are File, Edit, View, Special, and Help.

To create these menus, you first invoke the constructor in the `JMenu` class. Then you use the `add()` method from the `JMenuBar` class to add these menus to the menu bar one at a time, from left to right. These pull-down menus are initially empty. Here is how we might add the five menu items to the menu bar `mb` created earlier:

```
JMenu fileMenu = new JMenu("File");  // Create a File menu
mb.add(fileMenu);                    // Add it to the menu bar
JMenu editMenu = new JMenu("Edit");  // Create an Edit menu
mb.add(editMenu);                    // Add it to the menu bar
JMenu viewMenu = new JMenu("View");  // Create a View menu
mb.add(viewMenu);                    // Add it to the menu bar
```

```
JMenu specialMenu =
      new JMenu("Special");          // Create a Special menu
mb.add(specialMenu);                 // Add it to the menu bar
JMenu helpMenu = new JMenu("Help");  // Create a Help menu
mb.add(helpMenu);                    // Add it to the menu bar
```

We now have a menu bar that contains the names of five pull-down menus. The last step is to add the individual menu items to each menu. For example, when you pull down the Edit menu in Figure 12-21, you see three items: Cut, Copy, and Paste. You create these menu items using the constructor in the `JMenuItem` class. You then add these items to a pull-down menu using the `add()` routine in the `JMenu` class.

Here is how we add the three items Cut, Copy, and Paste to the Edit menu:

```
JMenuItem mCut = new JMenuItem("Cut");
editMenu.add(mCut);
JMenuItem mCopy = new JMenuItem("Copy");
editMenu.add(mCopy);
JMenuItem mPaste = new JMenuItem("Paste");
editMenu.add(mPaste);
```

We have constructed a menu that looks just like the one in Figure 12-21 and added it to our frame. However, although the menu looks nice, nothing happens if we click Copy in the Edit menu. In fact, no functionality is associated with any of the menu items in Figure 12-21. We have the same problem with the program shown in Figure 12-19. When we click one of the two buttons, it would be nice if its label (either *Press here* or *No, press here*) appeared in the text field labeled *The button that was pressed:*. Unfortunately, when you run the program in Figure 12-19 and click one of the buttons, nothing happens. Similarly, if you click the close box (X) in the upper-right corner of the `JFrame` in Figure 12-20, the window does not close.

We have learned how to create components, such as buttons, text fields, and menus, and we have shown how to add them to our frame, but the components do not yet carry out any meaningful operations. Adding functionality to a component is the last step in implementing a GUI; we discuss this important operation next.

12.4 Events and Listeners

12.4.1 Introduction

Things happen in a graphical user interface because of **events**, which are actions or conditions that occur outside your program's normal flow of control. For example, an event could be a mouse click, a keystroke, or a menu selection.

In the normal sequential style of programming, we can determine when a given statement is executed. For example, given the following sequence:

```
S1;    // Si are any Java statements
S2;
S3;
…
```

statement S_2 is executed at the completion of statement S_1, assuming it is not a branching statement. S_3 is executed next. However, we have no idea when an event will occur, because we cannot know when a user will click a button or enter data. Therefore, we cannot write GUI programs that assume events occur in either a specific order or within a specific time period.

Instead, we use the fact that whenever an event occurs, it generates an **event object** instantiated from a class in `java.awt.event` or `javax.swing.event`. Examples of event classes in these packages are listed in Figure 12-22.

EVENT CLASS	ACTION THAT WOULD CAUSE THE EVENT
`ActionEvent`	Clicking a button object in a GUI
`CaretEvent`	Selecting and modifying text on the screen
`KeyEvent`	Pressing a key on the keyboard
`ItemEvent`	Selecting an item from a pull-down menu
`MouseEvent`	Dragging the mouse or clicking a mouse button
`TextEvent`	Changing text within a text field
`WindowEvent`	Opening, closing, or resizing a window

[FIGURE 12-22] Example event classes in the `java.awt.event` and `javax.swing.event` packages

The object `e` generated by an event is sent to your program; it contains detailed information about the event that just occurred. What do we do with this event object, and how do we handle it? The answer is that we write special methods, called **event handlers**, that are automatically invoked upon receipt of an event object.

This is an extremely important point: An event handler method is not executed because we reach a given point in the program; it is executed because an event has occurred and an object that describes the event has been received. It is similar to working at your desk when the phone rings. You hear a ring (the event) that tells you someone is calling. You stop whatever you are doing and handle the phone call (the event handler). You have no idea when this event will occur.

This concept is called **event-driven programming**. We learn how to write these types of programs in subsequent sections.

12.4.2 Event Listeners

To respond to an event, we use an **event listener interface**. These interfaces specify the methods that handle the events generated by GUI components.

One event listener interface is usually associated with each type of component—a mouse listener interface, a button listener interface, a window listener interface, and so on. However, some components do not generate events (for example, a `JLabel`), so they have no interface. Even when a component does generate events, you may choose to ignore them. For example, you may not be interested in obtaining mouse location information from a `JPanel`. In that case, you would not bother to write a mouse event handler, and the program would not be notified when a mouse event occurs.

The process of creating an event handler for a component proceeds in three steps. The first step is to determine what listener interface is associated with a particular component. Different components generate different types of events—you can click a button but you cannot click a label—so you must implement the listener interface that responds to the proper event types. Figure 12-23 lists the component types introduced in the previous section, the listener interface associated with the component, and the methods specified by the listener.

COMPONENT	LISTENER	EVENTS GENERATED	METHODS
JButton	ActionListener	ActionEvent	actionPerformed()
JComboBox	ActionListener ItemListener	ActionEvent ItemEvent	actionPerformed() itemStateChanged()
JCheckBox	ActionListener ItemListener	ActionEvent ItemEvent	actionPerformed() itemStateChanged()
JLabel	None	None	None
JTextField	ActionListener CaretListener	ActionEvent CaretEvent	actionPerformed() caretUpdate()
JTextArea	ActionListener CaretListener	ActionEvent CaretEvent	actionPerformed() caretUpdate()
JMenuItem	ActionListener ItemListener	ActionEvent ItemEvent	actionPerformed() itemStateChanged()
JFrame	WindowListener	WindowEvent	windowActivated() windowClosed() windowClosing() windowDeactivated() windowDeiconified() windowIconified() windowOpened()
JRadioButton	ActionListener	ActionEvent	actionPerformed()

[FIGURE 12-23] Listener interfaces and events associated with components

For example, to create an event handler for the two buttons labeled *Press here* and *No, Press here* in Figure 12-20, we scan Figure 12-23 to discover that `JButton` objects generate an event called `ActionEvent`, and the correct listener interface to respond to this particular event is `ActionListener`. Similarly, if we want to handle events generated by a `JFrame`, we need a `WindowListener` that properly responds to a `WindowEvent`.

The second step in building your event handler is to write a class that implements this listener interface and that can serve as the event handler for this component. The listeners specified in the LISTENER column in Figure 12-23 are *interfaces*, not classes. That is, they specify the methods that must be written for a class to be qualified to serve as a handler for the component, but they do not contain any code themselves. If you think about it, this makes complete sense. A general-purpose listener for all buttons knows that it must respond to button clicks, but it cannot possibly know what it should do with that information. The response depends on the specific GUI and application.

The solution is to create a class that implements the interface by coding all of its methods. This is the list of names in the METHODS column in Figure 12-23. The interface often includes only a single method, but in some cases (for example, `WindowListener`) multiple methods must be written.

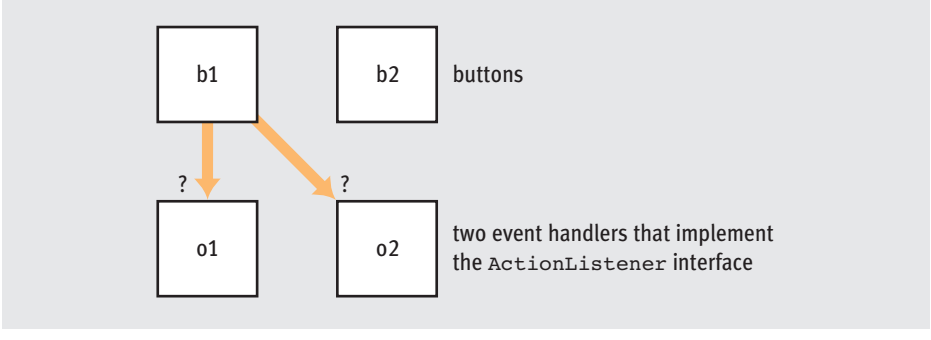
For example, Figure 12-24 shows a `MyButtonListener` class that implements the `ActionListener` interface—the single method `actionPerformed()`. The method implements the desired functionality of the two buttons in Figure 12-20—it puts the name of the button the user clicked into the text field.

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

public class MyButtonListener implements ActionListener {
    // The text field used to display the button information
    private JTextField display;

    /**
     * Create a button listener that uses the given
     * text field to display button information
     *
     * @param textField the field used to display button
           information
     */
    public MyButtonListener( JTextField theDisplay ) {
      display = theDisplay;
    }

    /**
     * This method is invoked whenever a button is pressed.
     * The label of the button pressed is displayed
     * in the text field associated with this listener.
     *
     * @param e the event that caused this method to be invoked
     */
    public void actionPerformed(ActionEvent e) {
     // Get the label of the button that was pressed
     String buttonName = e.getActionCommand();

     // Display the label in the text field
     display.setText( buttonName );
    }

} // MyButtonListener
```

[FIGURE 12-24] Example of a class that implements `ActionListener`

The class `MyButtonListener` in Figure 12-24 implements the `ActionListener` interface, as declared in the class header. This means it contains code for the one method specified by the interface, `actionPerformed()`. This method has a single input parameter—the event object `e` generated when you click a button. This event object contains descriptive information about the type of event that occurred and the component that caused it. The event handler uses the `getActionCommand()` method in the `ActionEvent` class to identify the button the user pressed. It returns a `String` that contains the label on the button that was pressed, either *Press here* or *No, press here*. We take this approach because the one event handler is handling events generated by two different buttons. We then take the string and store it in the `JTextField` using the `setText()` method.

How do we know that a specific instance of a `MyButtonListener` object is the one that should respond to a specific button event? Two or more classes may implement the `ActionListener` interface, possibly for other buttons or other components in the GUI, and each one includes an `actionPerformed()` method. For example, Figure 12-25 shows two buttons labeled b1 and b2 and two event handler objects o1 and o2 that both implement the `ActionListener` interface.



[FIGURE 12-25] Associating a component with a specific event handler

When button b1 is clicked, as shown in Figure 12-25, should we execute the `actionPerformed()` method in object o1 or the one in o2?

The answer lies in the third and final step in creating an event handler, called **registration**—creating an association between a component that generates an event and the specific object that implements the proper listener interface and responds to the event. This registration process uses a **listener registration method** that is contained in every component class or in one of its superclasses.

For example, the `JButton` class includes an instance method called `addActionListener()`:

```
public void addActionListener(ActionListener al);
```

When this method is executed by `b`, an instance of `JButton`, the `ActionListener` `al` becomes the "official" event handler associated with button `b`. Whenever `b` generates an event—for example, when it is clicked—the `actionPerformed(e)` method in object `al` is executed, where `e` is the event object generated by button `b`.

In the `MyButtonListener` class in Figure 12-24, we could register an event handler for the two button objects in Figure 12-20 using the following code:

```
MyButtonListener listener = new MyButtonListener(message);
pushButton1.addActionListener(listener);
pushButton2.addActionListener(listener);
```

In this example, we registered the same object as the handler for both buttons. This is not a problem because we can use the `getActionCommand()` method to identify the specific button the user clicked.

We could also have written two separate event handlers and associated each one with its own button, but it was not necessary in this example.

Whenever a button event is generated by either `pushButton1` or `pushButton2`, the event object `e` is sent to the `MyButtonListener` object registered above. Figure 12-26 shows a `ListenerDemo` class that adds the concept of button event handling to the program in Figure 12-19.

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

/**
 * A GUI that uses buttons, labels, and text fields.
 * It also includes event listeners for these objects.
 */
public class ListenerDemo
    extends JFrame implements ActionListener {

    public static final int WIDTH = 350;      // Frame width
    public static final int HEIGHT = 100;     // Frame height
    public static final int ROWS = 0;         // Rows in grid
    public static final int COLS = 2;         // Columns in grid
    public static final int FIELD_LEN = 20;   // Text field size
```

continued

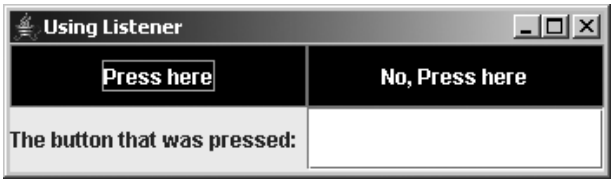
```
// Displays the label of the button the user pressed. This
// variable must be accessed by the actionPerformed()
// method, so it is declared to be part of the state
// of the object.
private JTextField message;

public ListenerDemo( String title ) {
    super( title );

    // Set up the main frame
    setLayout( new GridLayout( ROWS, COLS ) );
    setSize( WIDTH, HEIGHT );
    setBackground( Color.LIGHT_GRAY );

    // Create and configure the first button
    JButton pushButton1 = new JButton( "Press here" );
    pushButton1.setBackground( Color.BLACK );
    pushButton1.setForeground( Color.WHITE );
    add( pushButton1 );

    // Call the actionPerformed() method of this object
    // whenever the button is pressed
    pushButton1.addActionListener( this );

    // Create and configure the second button
    JButton pushButton2 = new JButton( "No, Press here" );
    pushButton2.setBackground( Color.BLACK );
    pushButton2.setForeground( Color.WHITE );
    add( pushButton2 );
    pushButton2.addActionListener( this );

    // This label describes the contents of the text field
    JLabel l = new JLabel( "The button that was pressed: " );
    add( l );

    // When a button is pressed, the label that appears on
    // the button is displayed in this text field
    message = new JTextField( FIELD_LEN );
    add( message );
}

/**
 * This method is invoked whenever the user presses one
 * of the two buttons in the GUI. The label that appears
 * in the button the user pressed is displayed in the
 * text field on the screen.
```

continued

```
     *
     * @param e the event that caused this method to be invoked
     */
    public void actionPerformed( ActionEvent e ) {
        // Get the label of the button that was pressed
        String buttonName = e.getActionCommand ();

        // Display the label in the text field
        message.setText( buttonName );
    }

    /**
     * Create an instance of a ListenerDemo and display it on
     * the screen
     */
    public static void main( String args[] ) {
        ListenerDemo win = new ListenerDemo( "Using Listener" );

        win.setVisible( true );
    }

} // ListenerDemo
```

[FIGURE 12-26] GUI program with event handlers added

When we run the program in Figure 12-26 and click one of the two buttons in the interface, we get the desired behavior—the name of the button we pressed appears in the text field. In Figure 12-27, the user has not pressed a button, so no text appears in the text field.



[FIGURE 12-27] Result of executing the program in Figure 12-26

To summarize, the following steps create an event handler and attach it to a component:

1. Determine the appropriate listener interface required by this component. You can make this determination using Figure 12-23 or the Java documentation.
2. Implement the listener interface by writing a class `c` that contains code for all methods in the interface. These methods describe the behaviors that you want to occur when this event is generated.
3. Register an instance of class `c` as the event handler for this component.

Let's do a second example, this time using menus. Selecting a menu item from a pull-down menu generates an `ActionEvent` and uses the `ActionListener` interface, as we can see in Figure 12-23. If we assume the menu structure in Figure 12-21, then a possible event handler for the items in the Edit menu is shown in Figure 12-28. This example assumes that if you select the Cut menu item, the system invokes a method called `cutProcedure()`, not shown here, which handles the details associated with the cut operation. We assume that similar procedures exist for the Copy and Paste options in the Edit menu pane. (These routines would typically be part of a word-processing software package.) This method uses the `getActionCommand()` method to retrieve the specific command selected by the user.

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

public class MyMenuListener implements ActionListener {

    public void actionPerformed( ActionEvent e ) {
        String command = e.getActionCommand();
        if ( command.equals( "Cut" ) ) {
            cutProcedure();
        }
        else if ( command.equals( "Copy" ) ) {
            copyProcedure();
        }
        else if ( command.equals( "Paste" ) ) {
            pasteProcedure();
        }
    }

} // MyMenuListener
```

[FIGURE 12-28] Event handler for the menu shown in Figure 12-21

The `MyMenuListener` class in Figure 12-28 was written to handle events generated by the selection of a menu item. Usually one class handles all events from components of the same type. For example, you might write a class to handle all button events, one to handle all menu events, and a third to handle all mouse events. However, you can also write a single

event handler that handles events from different types of components, as long as they all use the same listener interface. In that case, you can use the `getSource()` method in the `Event` class to identify the object that caused the event. Then, using the **instanceof** operator, you determine the class of the object. This sequence is quite common in event handlers because different components often generate the same event type. For example, in Figure 12-23, we see that both `JButton` and `JMenuItem` components generate an `ActionEvent`. Similarly, both `JCheckBox` and `JComboBox` generate an `ItemEvent`. If you are writing an event handler to handle events from both button and menu objects, you must first determine where the event came from and what type of object produced it. Most event classes contain methods for handling these identification operations. For example, the `ActionEvent` class includes the following methods:

- `getSource()`, which returns the object that generated this event
- `getActionCommand()`, which returns a string naming the object that generated the event, either the button label or menu item name
- `getID()`, which returns a constant that specifies the event type

This section provides one more example of creating and implementing an event handler. The window in Figure 12-27 contains a close box (X) in the upper-right corner. Most GUIs use this box as an easy way to close a window and terminate execution of a program. However, the default behavior of the close box in a `JFrame` is to hide the frame, not terminate the program. There are two ways to change this default behavior. The first is to write an event handler that implements the desired behavior and have it invoked whenever the user clicks the close box.

According to Figure 12-23, the listener interface that handles `JFrame` operations is `WindowListener`. This interface contains seven methods that handle the seven different events generated by containers like `JFrame`—events such as opening and closing the window, hiding the window, and iconifying it. In this example, the only method we care about is `WindowClosing()`, which is activated whenever we click the mouse inside the close box. The easiest way to terminate the program is to invoke `System.exit(0)`. However, when implementing an interface, we must provide code for *all* methods in the interface, not just the methods of interest. Therefore, we also must write "do-nothing" routines, called **stubs**, for the other six methods in the interface. If we do not, our program will not compile.

The `MyWindowListener` class in Figure 12-29 implements the complete `WindowListener` interface. You can use it to instantiate a `WindowListener` object that terminates a program when the user closes the window by clicking the close box.

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

public class MyWindowListener implements WindowListener {
    // We put in stubs for all methods except WindowClosing.
    // The stubs are necessary because this class
    // does not implement the interface without them.
    // Javadoc comments for these methods can be found in
    // documentation provided by Sun for the WindowListener
    // interface

    // Invoked after a window has been closed and disposed of
    public void windowClosed( WindowEvent e ) {}

    //Invoked the first time a window is opened
    public void windowOpened( WindowEvent e )  {}

    // Invoked when a window is restored
    public void windowDeiconified( WindowEvent e ) {}

    // Invoked when a window is minimized
    public void windowIconified( WindowEvent e ) {}

    // Invoked when the window becomes the active window
    public void windowActivated( WindowEvent e ) {}

    // Invoked when the window no longer is the active window
    public void windowDeactivated( WindowEvent e ) {}

    // Invoked when the user attempts to close a window
    public void windowClosing( WindowEvent e ) {
        // Simply terminate the program
        System.exit(0);
    }

} //MyWindowListener
```

[FIGURE 12-29] A class that implements the `WindowListener` interface

For most `WindowListener` handlers, you only want to write code for one or two of the seven methods in the interface. For example, the `WindowListener` in Figure 12-29 includes code for only the `windowClosing()` method. The remaining six methods are stubs. To make things more convenient, the AWT package includes adapter classes for some of the listener interfaces. **Adapter classes** provide stubs for all methods in the corresponding interface. In a sense, they are complete "do-nothing" classes that implement the corresponding interface. You use an adapter class by writing a new class that *extends* the adapter and overrides the specific method(s) that deal with the events of interest. All other classes remain as stubs that are inherited from their parent class.

For example, the `WindowAdapter` class in AWT can be extended to create handlers for window events. The `MyWindowListener` class in Figure 12-30 provides the same functionality as Figure 12-29, but because it extends the `WindowAdapter` class, it only needs to provide code for the `windowClosing()` method. The other six methods have already been implemented as stubs in the adapter class.

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

public class MyWindowListener extends WindowAdapter {
    // We only need to override the methods that correspond
    // to the events we are interested in handling. The stubs
    // are provided by the superclass.

    // Invoked when the user attempts to close a window
    public void windowClosing( WindowEvent e ) {
        // Simply terminate the program
        System.exit(0);
    }

} //MyWindowListener
```

[FIGURE 12-30] Using the `WindowAdapter` class to implement an event handler

Finally, we must register an instance of the window listener class from either Figure 12-29 or Figure 12-30, using the `addWindowListener()` method in the `Window` class. This associates the event handler with our `JFrame` object:

```
// Use a window listener to terminate the program if
// the user closes the window. We can use either
// the class shown in Figure 12-29 or Figure 12-30.
addWindowListener( new MyWindowListener() )
```

When these lines are added to Figure 12-24, along with the `MyWindowListener` class, the GUI has the correct close box functionality. When you click the mouse button within the close box, the program terminates. You might want to try adding this feature to Figure 12-24 to confirm that it works.

The second way to handle a window closing event is to use the `setDefaultCloseOperation()` method to specify which default operation executes when the user closes a window. The method takes an integer parameter that specifies the action to take when the window is closed. The four constants you can pass to the `setDefaultCloseOperation()` method and the action they specify are listed in Figure 12-31.

CONSTANT	ACTION SPECIFIED
`WindowConstants.DO_NOTHING_ON_CLOSE`	Don't do anything; if the window closing event must be handled, the programmer must provide a `WindowListener` object to handle the event
`WindowConstants.HIDE_ON_CLOSE`	Hide the frame after notifying any registered `WindowListener` objects
`WindowConstants.DISPOSE_ON_CLOSE`	Hide and dispose of the frame after notifying any registered `WindowListener` objects
`JFrame.EXIT_ON_CLOSE`	Terminate the program using the `System.exit(0)` method

[FIGURE 12-31] Valid arguments to `setDefaultCloseOperation()`

Rather than writing event handlers to implement the window closing operation, the remaining examples use the `setDefaultCloseOperation()` method, passing it the constant value `JFrame.EXIT_ON_CLOSE` to specify that the program should be terminated when the user closes the window.

12.4.3 Mouse Events

The handling of mouse events is similar in most ways to the discussion in the previous section. The primary difference is that mouse events include motion and position information, and we may need to deal with (x, y) coordinate locations on the screen. Figure 12-32 lists the listeners and events associated with the movement of a mouse.

EVENT LISTENER	EVENTS GENERATED	METHODS

`MouseListener`	`MouseEvent`	`mousePressed()`
		`mouseReleased()`
		`mouseClicked()`
		`mouseEntered()`
		`mouseExited()`

`MouseMotionListener`	`MouseEvent`	`mouseDragged()` `mouseMoved()`
`MouseWheelListener`	`MouseWheelEvent`	`mouseWheelMoved()`

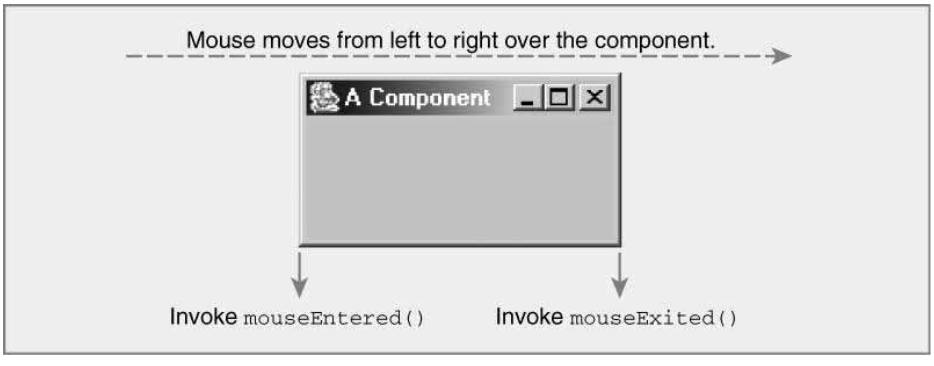
[FIGURE 12-32] Mouse events and listeners

As we can see from Figure 12-32, five methods must be included in a class that implements the `MouseListener` interface. The `mousePressed()` method is invoked when the mouse button is pressed and the cursor is located within the boundary of some component. The `mouseReleased()` method is invoked whenever a mouse button is released over a component. The `mouseClicked()` method combines the behavior of the previous two methods. It is called whenever the user presses and releases a mouse button without any intervening movement. The `MouseEvent` class inherits the method `getID()` that allows you to differentiate between these different types of mouse events, such as `MOUSE_PRESSED`, `MOUSE_RELEASED`, and `MOUSE_CLICKED`.

The methods `getX()` and `getY()` retrieve the x- and y-coordinates of the mouse event's location. These coordinates are specified in terms of pixels. Referring back to Figure 12-7, we see that pixel numbering begins at (0,0) in the upper-left corner and increases as you move right (the x-axis) and down (the y-axis). The `getClickCount()` method specifies how many times the mouse button was clicked. This method is useful when you want to perform an operation only if the mouse button was double-clicked. If your mouse has two or more buttons, the `getButton()`method can identify which button changed state.

Two methods are associated with movement of the cursor over a component. The `mouseEntered()` method is invoked whenever a cursor moves across the boundary of a component. The `mouseExited()`method is invoked when the cursor exits a component. These two methods can be used, for example, to change the shape of the cursor when it is located within the boundaries of a component. You can use the `getComponent()` method, inherited from `ComponentEvent`, to determine which component was entered or exited.

Figure 12-33 demonstrates what happens when these two routines are invoked.



[FIGURE 12-33] Mouse movement events

The `MyMouseListener` class in Figure 12-34 illustrates how we can use the `MouseAdapter` class to create a listener that responds to mouse entry and mouse exit events. Because we are using an adapter class, we do not have to write stubs for the other three methods in the interface.

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

public class MyMouseListener extends MouseAdapter {
    // Where button information is displayed
    private JTextField displayField;

    /**
     * Create a mouse listener that displays information
     * using the specified text fields
     *
     * @param display where button information is displayed
     */
    public MyMouseListener( JTextField display ) {
        displayField = display;
    }

    /**
     * Invoked when the mouse is used to enter a component.
     * This method assumes that only buttons are looking
     * for this event and will display the button text
     * using the designated text field.
     *
     * @param e the event that caused this method to be invoked
     */
    public void mouseEntered( MouseEvent e ) {
        // Determine the button that generated this event
        JButton button = (JButton)e.getComponent();

        // Display the button text in the text field
        displayField.setText( button.getText() );
    }

    /**
     * Invoked when the mouse leaves a component.
     * Clears the display text field.
     *
     * @param e the event that caused this method to be invoked
     */
    public void mouseExited( MouseEvent e )  {
```

continued

```
        // We have exited a component; blank out the text field
        displayField.setText( "" );
    }

} // MyMouseListener
```

[FIGURE 12-34] A class that implements `MouseListener`

Figure 12-35 shows a program that illustrates how the `MyMouseListener` class in Figure 12-34 can handle mouse events. This GUI contains two buttons and two text fields. Whenever the cursor is positioned over one of the two buttons, the text field contains the label of that button; otherwise, the text field is blank. This example gives you an idea of how mouse events are handled.

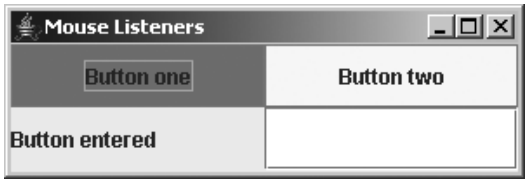
```
import java.awt.*;
import javax.swing.*;

public class MouseDemo extends JFrame {
    // The size of the frame
    private static final int WIDTH = 300;   // Frame width
    private static final int HEIGHT = 100;  // Frame height
    private static final int FIELD_SIZE = 20;   // Input field length

    /**
     * Create a GUI that demonstrates how to use a MouseListener
     *
     * @param title the title of the frame
     */
    public MouseDemo( String title ) {
        super(title);

        JTextField display = new JTextField( FIELD_SIZE );
        JButton button = null;

        // The listener that will handle mouse events
        MyMouseListener listener = new MyMouseListener( display );

        // Set up the frame
        setSize( WIDTH, HEIGHT );
        setLayout( new GridLayout( 0, 2 ) );
        setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

        // Create two buttons and add them to the frame
        button = new JButton( "Button one" );
```

continued

```
        button.setBackground( Color.RED );
        button.addMouseListener( listener );
        add( button );

        button = new JButton( "Button two" );
        button.setBackground( Color.YELLOW );
        button.addMouseListener( listener );
        add( button );

        // Label and text field for the mouse information
        add( new JLabel( "Button entered" ) );
        add( display );
    }

    /**
     * Display a MouseDemo GUI
     */
    public static void main( String args[] ) {
        MouseDemo win = new MouseDemo( "Mouse Listeners" );
        win.setVisible( true );
    }

} // MouseDemo
```

[FIGURE 12-35] Using the `MouseListener` class of Figure 12-34

Figure 12-36 illustrates the GUI produced by the program in Figure 12-35; note that the cursor is not positioned over either button.



[FIGURE 12-36] The GUI produced by the program in Figure 12-35

This section demonstrated a number of important GUI concepts, such as creating a component, adding it to a container, and implementing functionality using event handlers. The following section puts all these ideas together; it shows you how to design and implement a complete user interface as part of some interesting and important applications.

STEVEN JOBS, 21ST CENTURY ENTREPRENEUR

Steve Jobs is one of the best-known people in computing. Most know him for his groundbreaking innovations at Apple and NeXT. Others know him from his work at Pixar, a leader in computer animation that has produced such hit movies as *Toy Story*, *Monsters Inc.*, *Finding Nemo*, *The Incredibles*, and *Cars*. He helped create the iPod and iTunes music store, which are revolutionizing the music industry. As a result, Jobs has achieved an almost cultlike reputation, which is only enhanced by his quirky persona and unusual behavior—he once backpacked around India to achieve spiritual enlightenment, returning with a shaved head and wearing Indian robes.

Jobs was born in San Francisco in 1955. He demonstrated an early interest in computing and worked for Hewlett-Packard in high school, where he met Steve Wozniak. He dropped out of college after one semester and took a job with Atari, one of the first producers of video games. He began to frequent computer clubs in the Bay Area with Wozniak, who had already designed his own personal computer. In 1976, they started Apple Computer to market this new machine, which they named the Apple I. It did not do well, but the Apple II, released in 1977, was an enormous success and brought personal computing to the general public. The Apple II also brought Jobs fame and wealth at the tender age of 22. In 1984, Apple released the Macintosh—the first commercial machine to use a graphical interface and mouse. Its design was greatly influenced by the Xerox Alto created at Xerox PARC, which Jobs had seen on a tour in the late 1970s. (See "Missed Opportunities" earlier in this chapter.)

In spite of his amazing successes, Jobs was stripped of his management role at Apple after an internal power struggle, and he resigned from the company. He founded NeXT Computer, which produced innovative and technologically advanced hardware and software systems, and he bought the computer graphics division of LucasFilm. Jobs renamed the company Pixar, which quickly moved to the forefront in the field of computer animation.

In 1996, Apple bought NeXT, and Jobs triumphantly returned as CEO to the company he had cofounded 20 years earlier. He has helped the company develop such important new products as Mac OS X, the iPod, Airport Wireless Networking, and iTunes. Although Apple has only a 4 percent share of the PC market, it is a recognized leader in technical innovation.

12.5 GUI Examples

This section includes three complete programming examples. Their purpose is to give you experience with interface design and to demonstrate techniques for building GUIs.

12.5.1 GUI Example #1: A Course Registration System

The first example is a visual interface for a course registration system that allows students to select courses they want to take. This example makes use of two components we have discussed—`JCheckBox` and `JTextArea`. A check box is a Boolean component that can either be selected or not selected, but not both. A text area is a two-dimensional text field that has both rows and columns.

The following example includes a check box for each computer science course being offered. Students look over this list and check up to three courses they want to take. When they have completed their selection, they click the Finished button. The text area then lists the selected courses. If a student checked no courses or more than three, an error message explains the mistake and asks the student to redo the registration.

A `JCheckBox` component requires an event handler that implements the `ItemListener` interface. This interface has a single method—`ItemStateChanged`. It is invoked whenever a user checks or unchecks a box to select or deselect a course. Text areas and buttons both use the `ActionListener` interface we have seen several times. This handler is called when a user clicks the Finished button. Its task is to see if the user selected the correct number of courses and, if so, to display them in the text area. If not, it displays an appropriate error message. Notice that when we register our listeners, we use the reference **this** to indicate that the object itself serves as the event handler for both components. The program (see Figure 12-37) uses a border layout for the interface, with an explanatory message in the North area, the course list in the West area, the Finished button in the South area, and the list of selected courses in the Center area.

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

/**
 * A data input screen that allows students to select courses.
 * Students select 1 to 3 courses and then press the Finished
```

continued

```
 * button. The course names appear in the text area.
 */
public class GUIExample1
    extends JFrame implements ItemListener, ActionListener {

    private static final int WIDTH = 500;   // Frame width
    private static final int HEIGHT = 250;  // Frame height

    // The names of the courses
    private static final String courses[] = { "CS 101",
                                              "CS 102",
                                              "CS 210",
                                              "CS 215",
                                              "CS 217",
                                              "CS 302" };

    private Set<String> selectedCourses;  // Selected courses
    private JTextArea display;            // Message display area

    /**
     * Create a course selection screen
     *
     * @param title the title to be placed in the frame
     */
    public GUIExample1( String title ) {
        super( title );

        // Use a tree set to display courses alphabetically
        selectedCourses = new TreeSet<String>();

        // Set frame attributes
        setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        setSize( WIDTH, HEIGHT );
        setLayout( new BorderLayout() );

        // Instructions for the user
        JLabel l1 =
            new JLabel( "Please select the courses you wish " +
                        " to take.  Click on the Finished " +
                        "button when done." );

        add(l1, BorderLayout.NORTH );

        // The check boxes for course selection go in a panel
        JPanel chkPanel = new JPanel();
        chkPanel.setLayout( new GridLayout( courses.length, 0 ) );
```

continued

```
        chkPanel.setBorder( BorderFactory.createEtchedBorder() );

        // Create the check boxes and add them to the panel
        for ( int i = 0; i < courses.length; i++ ) {
            JCheckBox box = new JCheckBox( courses[ i ] );

            // This object handles the state change events
            box.addItemListener( this );

            chkPanel.add( box );
        }

        add( chkPanel, BorderLayout.WEST );

        // Create the Finished button
        JButton done = new JButton( "Finished" );
        done.addActionListener( this );

        add( done, BorderLayout.SOUTH );

        // Create the text box that displays messages
        display = new JTextArea( 10, 20 );
        display.setBorder(
            BorderFactory.createTitledBorder(
                BorderFactory.createEtchedBorder(),
                "Courses Selected" ) );
        display.setBackground( getBackground() );

        add( display, BorderLayout.CENTER );
    }

    /**
     * This method is called whenever the user clicks a
     * check box. If the user selects a course, it is
     * added to the set of selected courses. If the user
     * deselects a course, it is removed from the set.
     *
     * @param event the event that occurred
     */
    public void itemStateChanged( ItemEvent event ) {
        // The label on the check box is the name of the course
        String courseName =
            ((JCheckBox)event.getItem() ).getText();

        if ( event.getStateChange() == ItemEvent.SELECTED ) {
            // A user is adding the course
```

continued

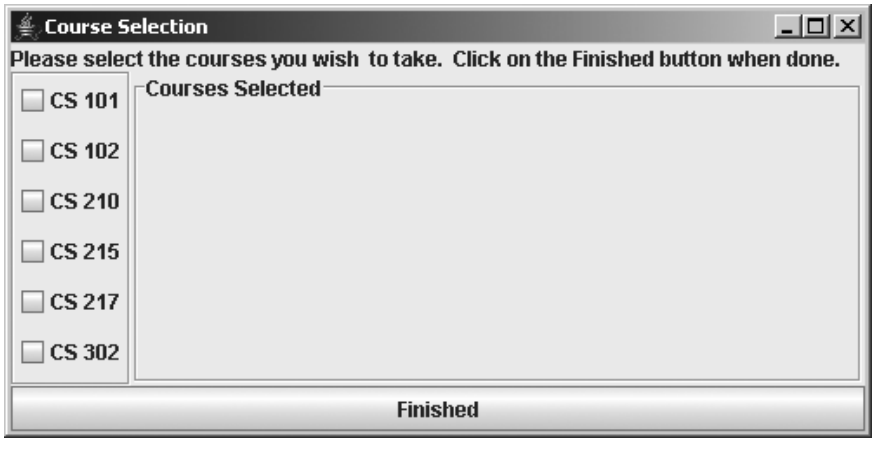
```
            selectedCourses.add( courseName );
        }
        else {
            // A user is dropping the course
            selectedCourses.remove( courseName );
        }
    }

    /**
     * This method is invoked whenever the user clicks
     * the Finished button. If between 1 and 3 courses have
     * been selected, the course names are displayed in
     * the GUI. If the user selects an incorrect number
     * of courses, an error message appears.
     *
     * @param event the action event that occurred
     */
    public void actionPerformed( ActionEvent event ) {
        // The information displayed on the screen
        String text = "";

        if ( selectedCourses.size() < 1 ||
             selectedCourses.size() > 3 ) {
            // Too few or too many courses
            text = "You must select between 1 and 3 courses";
        }
        else {
            // Use an iterator to get the course names out of the
            // set and format them into a string separated by
            // newline characters
            Iterator<String> i = selectedCourses.iterator();
            while ( i.hasNext() ) {
                text = text + i.next() + "\n";
            }
        }

        // Display the message
        display.setText( text );
    }

    /**
     * Run the course selection program
     *
     * @param args command-line arguments (ignored)
     */
    public static void main( String args[] ) {
```

continued

```
        // Create the GUI
        GUIExample1 screen =
            new GUIExample1( "Course Selection" );

        // Display it on the screen
        screen.setVisible( true );
    }

} // GUIExample1
```

[FIGURE 12-37] Example GUI using `JCheckBox` and `JTextArea`

When the program in Figure 12-37 executes, the window in Figure 12-38 appears.



[FIGURE 12-38] Screen produced by executing the program in Figure 12-37

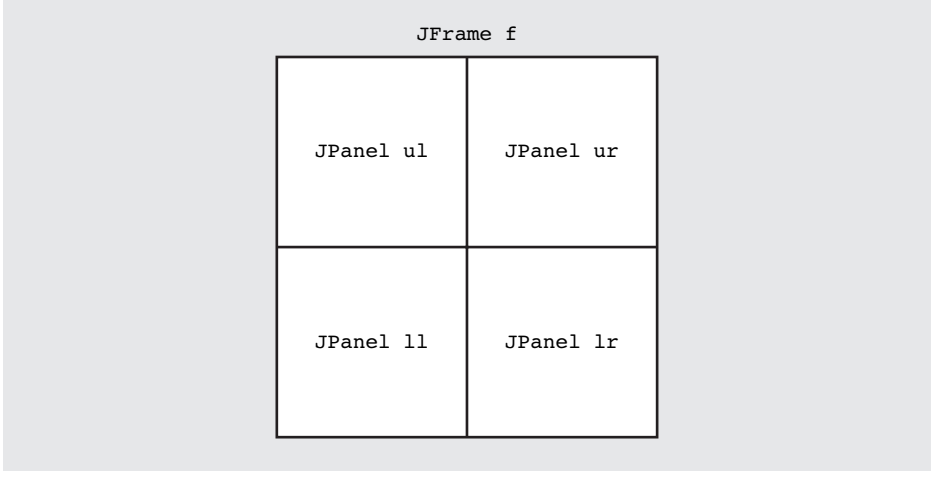
12.5.2 GUI Example #2: A Data Entry Screen

Our second example is a demonstration of a GUI used for data entry. In this application, the user is asked to enter personal information, including name, school, major, hometown, and home state. Users type in the data and then click the Store button when they finish. The information is then displayed in a text area. Of course, in real life the information would not simply be displayed; it would also be stored as a record in a data file. Such operations are typical when you register as a new user for an online service.

The most important new feature presented in this example is the use of panels to simplify the GUI-building task. In this example, the interface is not constructed as one large container; instead, it is built from other containers of type `JPanel`.

Panels, which we first mentioned in Section 12.2.2, are borderless containers that can simplify GUI construction. A `JPanel` is a fully functional container that has its own properties, layout manager, and components. For example, it is perfectly permissible for a `JPanel` object to use a `FlowLayout` manager and enter components in a strict left-to-right fashion. When the panel is added to a `JFrame` object, the frame itself may use a totally different scheme, such as a border layout.

We use `JPanel` objects to implement a "divide and conquer" GUI design strategy. Rather than think about the entire screen at once, we can work on individual regions. For example, assume we are designing a `JFrame f` that contains a huge number of components. Instead of dealing with the entire collection, we construct a `JPanel` called `ul` that contains only the components in the upper-left quadrant of `f`. Next, we construct a second panel, `ur`, that contains the components in the upper-right quadrant. We repeat this process for as many subproblems as we want to create. When we finish, we add the individual panels, which are themselves components, to `JFrame f`. This process is diagrammed in Figure 12-39.



[FIGURE 12-39] Using four panels to construct a frame

This example also demonstrates another new component—a `JComboBox`. This is a drop-down list of items, only one of which can be selected at a time. At first, only the selected item is visible on the screen. When you place the cursor over the combo box and click the mouse, a list of all items appears. You move the cursor through the list and select an item, which is then displayed in the visible portion of the combo box. The items in the drop-down list are added to the combo box using the `addItem()` method in the

JComboBox class or are initialized to the values contained in an array passed to the JComboBox constructor. You can specify the initially selected item using the setSelectedIndex()method, where the first item in the list has index 0.

In this example, users select their home state from the State combo box. Assume that when the GUI is created, Minnesota (MN) has been selected. The user sees the first window on the left:

When the user places the mouse over this component and clicks the mouse button, a drop-down list appears with the entries MN, WI, and IA, as shown in the middle window. You can move the cursor up or down to select any of the three items. For example, if you select MN, you see the screen in the middle just before releasing the mouse. If you select WI and then release the mouse button, the drop-down list disappears and the selected item, WI, appears on the screen, as shown in the screen at right.

Another Swing component, **radio buttons** (also known as option buttons), combine the look and feel of the multiselection JCheckBox from the first example with the single-selection JComboBox component. A ButtonGroup class is a collection of buttons that permit only one selection at a time. If the user selects a radio button that is part of a ButtonGroup, the previously selected button is automatically deselected. New radio buttons are added to a ButtonGroup using the add() method in the ButtonGroup class. You determine the button's initial selection using the setSelected() method.

As an alternative, we could use a ButtonGroup rather than a JComboBox to implement the state selection process. If we created three radio buttons and added them to a ButtonGroup, they would look like the buttons in the following window, assuming that the MN button is the initial selection:

If a user selected the WI button, the selection indicator on the MN button would be automatically removed:

Although we do not use the `ButtonGroup` component in our example, it is a good example of the wide range of component types that a modern interface design package provides. Our discussions in this chapter introduce only a sampling of the many resources in packages such as AWT and Swing.

A GUI that implements the data entry operation we just described is shown in Figure 12-40. The screen created by the program is shown in Figure 12-41.

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

/**
 * Display a simple data entry screen. This program
 * illustrates the use of listeners and a JComboBox.
 */
public class DataEntryGUI
    extends JFrame implements ActionListener {

    // Component dimensions
    private static final int WIDTH = 775;
    private static final int HEIGHT = 250;
    private static final int PROMPT_WIDTH = 75;
    private static final int PROMPT_HEIGHT = 15;
    private static final int DATA_LENGTH = 30;
    private static final int NUM_FIELDS = 5;

    // Constants used to identify the data entry fields
    private static final int NAME = 0;
    private static final int SCHOOL = 1;
    private static final int MAJOR = 2;
    private static final int HOMETOWN = 3;
    private static final int STATE = 4;

    // The states listed by the JComboBox
    private static final String stateList[] = {
        "AK", "AL", "AR", "AZ", "CA", "CO", "CT", "DE", "FL",
        "GA", "HI", "ID", "IL", "IN", "IA", "KS", "KY", "LA",
        "ME", "MD", "MA", "MI", "MN", "MS", "MO", "MT", "NE",
        "NV", "NH", "NJ", "NM", "NY", "NC", "ND", "OH", "OK",
        "OR", "PA", "RI", "SC", "SD", "TN", "TX", "UT", "VT",
         "VA", "WA", "WV", "WI", "WY"
    };
```

continued

```
// The names of the fields in the GUI
private static final String fieldName[] = {
    "Name:",
    "School:",
    "Major:",
    "Hometown:"
};

// This component allows the user to select a state
private JComboBox stateSelector;

// The input areas will be stored in this array
private JTextField field[];

// Where the completed information will be stored
private JTextArea displayArea =
    new JTextArea( NUM_FIELDS, DATA_LENGTH );

/**
 * Create a data entry screen
 *
 * @param title the frame title
 */
public DataEntryGUI( String title ) {
    super( title );  // Let the JFrame initialize things

    // Make prompts all the same size
    Dimension promptSize =
        new Dimension( PROMPT_WIDTH, PROMPT_HEIGHT );

    // Create the array that will hold the text fields
    field = new JTextField[ fieldName.length ];

    // Create the data entry panel
    JPanel dataEntry = new JPanel();
    dataEntry.setLayout( new GridLayout( 0, 1 ) );

    // For each field in the GUI, create a panel that
    // holds the name of the field and the associated
    // text field. This way, we can be sure that
    // the field description and the text area
    // are in the same row.
    for ( int i = 0; i < field.length; i = i + 1 ) {
        JPanel row = new JPanel();

        // Create the text field
        field[ i ] = new JTextField( DATA_LENGTH );
```

continued

```
        // Create the description for the field
        JLabel prompt = new JLabel( fieldName[ i ] );
        prompt.setPreferredSize( promptSize );

        // Put them in the panel
        row.add( prompt );
        row.add( field[ i ] );

        // Put the panel in the main frame
        dataEntry.add( row );
    }

    // The state is handled differently because it is
    // a combo box and not a text field like the others

    JPanel row = new JPanel();
    row.setLayout( new FlowLayout( FlowLayout.LEFT ) );

    JLabel prompt = new JLabel( "State" );
    prompt.setPreferredSize( promptSize );

    stateSelector = new JComboBox( stateList );

    row.add( prompt );
    row.add( stateSelector );

    dataEntry.add( row );

    // The data entry panel and the display area are
    // placed in a single panel so that they occupy the
    // center area of the frame
    JPanel centerPanel = new JPanel();
    centerPanel.setLayout( new FlowLayout() );

    centerPanel.add( dataEntry );
    centerPanel.add( displayArea );

    // The buttons go in a separate panel and
    // eventually appear in the south area of the frame
    JPanel buttonPanel = new JPanel();
    buttonPanel.setLayout( new FlowLayout() );

    JButton button = new JButton( "Store" );
    button.addActionListener( this );
    buttonPanel.add( button );
```

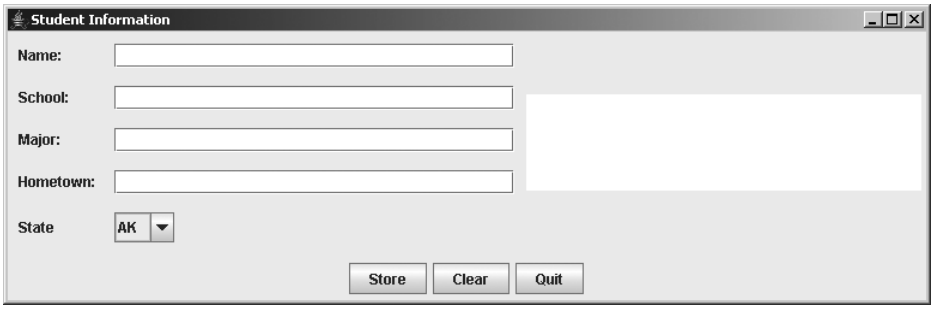
continued

```
        button = new JButton( "Clear" );
        button.addActionListener( this );
        buttonPanel.add( button );

        button = new JButton( "Quit" );
        button.addActionListener( this );
        buttonPanel.add( button );

        // Set up the main frame
        setSize( WIDTH, HEIGHT );
        setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

        // Add the subcomponents to the main frame
        setLayout( new BorderLayout() );
        add( BorderLayout.CENTER, centerPanel );
        add( BorderLayout.SOUTH, buttonPanel );
    }

    /**
     * This method is invoked whenever a user
     * presses a button in the GUI
     *
     * @param e the event that caused this method to be invoked
     */
    public void actionPerformed( ActionEvent e ) {
        // Get the label on the button the user pressed
        String arg = e.getActionCommand();

        // Determine which button was pressed
        if ( arg.equals( "Store" ) ) {
            // If the Store button was pressed, display the
            // data entered in the GUI display area

            // Erase any text being displayed
            displayArea.setText( "" );

            // Copy the contents of the text fields and put
            // them in a single string separated by newline
            // characters
            for ( int i = 0; i < field.length; i = i + 1 ) {
                displayArea.append( field[ i ].getText() +
                                    '\n' );
            }

            // Get the state from the combo box
            String s = (String) stateSelector.getSelectedItem();
```

continued

```
            // Display the results
            displayArea.append( s );
        }
        else if ( arg.equals( "Clear" ) ) {
            // Erase any text being displayed
            displayArea.setText( "" );
        }
        else {
            // Terminate the program
            System.exit( 0 );
        }
    }

    /**
     * Create an instance of a DataEntryGUI and make it visible
     */
    public static void main( String args[] ) {
        DataEntryGUI mainWindow =
            new DataEntryGUI( "Student Information" );
        mainWindow.setVisible( true );
    }

} // DataEntryGUI
```

[FIGURE 12-40] A GUI that demonstrates data entry operations



[FIGURE 12-41] Display produced by the program in Figure 12-40

12.5.3 GUI Example #3: A Four-function Calculator

Our final example implements a classic GUI-based application that is part of virtually every computer sold today—a four-function calculator (see Figure 12-42). This example uses many of the components we have discussed throughout the chapter.

[FIGURE 12-42] Interface for a four-function calculator

We construct our interface using a `JPanel` named `buttonPanel` that holds the 16 buttons of the calculator—10 number buttons and six function buttons. We use a `JLabel` to display the computed numerical result.

From looking at Figure 12-42, it seems reasonable to select a grid layout and place the buttons into a grid of four rows by four columns. We can use a border layout for the entire frame; the `JLabel` that contains the display can go in the `NORTH` region, and the button panel can be placed in the `CENTER` region. (We do not use the East, South, or West regions.)

The entire interface has a single event handler for the `actionPerformed()` method, which is called when a user clicks any of the 16 buttons. The program uses the `getActionCommand()` method to determine which button was selected and passes the information to `handleButton()`, an instance method of the class `CalcLogic`. This class implements the arithmetic functions of the calculator; in other words, it performs addition, subtraction, multiplication, and division operations.

The code in Figure 12-43 implements all visual aspects of the interface. Because the actual calculations are not directly related to the interface design, we leave the implementation of the `CalcLogic` class as an exercise at the end of the chapter.

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

/**
 * A simple four-function calculator
 */
public class Calculator
    extends JFrame implements ActionListener {
```

continued

```
    // The labels on the buttons
    private static final String labels = "789X456/123-0C=+";

    // Frame size
    private static final int WIDTH = 200;
    private static final int HEIGHT = 200;

    // The font to use in the display
    private static final Font
        DISPLAY_FONT = new Font( null, Font.BOLD, 20 );

    // Number of rows and columns in the calculator
    private static final int NUM_ROWS = 4;
    private static final int NUM_COLS = 4;

    // The object that knows how to do the arithmetic
    private CalcLogic myCalc;

    // The display portion of the GUI
    private JLabel display;

    /**
     * Create a new calculator
     */
    public Calculator( String name ) {
        super( name );

        JPanel buttonPanel = new JPanel();

        // The object that knows how to process keys. This GUI
        // simply catches button presses and passes them
        // to this object, which knows what to do with them.
        myCalc = new CalcLogic();

        // Configure the frame
        setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        setLayout( new BorderLayout() );
        setSize( WIDTH, HEIGHT );
        setResizable( false );

        // Create the button panel
        buttonPanel.setLayout(
            new GridLayout( NUM_ROWS, NUM_COLS ) );

        // Create the buttons and place them in the panel
        for ( int i = 0 ; i < labels.length() ; i = i + 1 ) {
            JButton b =
                new JButton( labels.substring( i, i + 1 ) );
```

continued

```
            b.addActionListener( this );
            buttonPanel.add( b );
        }

        // Create the display
        display = new JLabel( "0", JLabel.RIGHT );
        display.setFont( DISPLAY_FONT );

        // "Assemble" the calculator
        add( BorderLayout.NORTH, display );
        add( BorderLayout.CENTER, buttonPanel);
    }

    /**
     * Return the current contents of the display portion
     * of the GUI
     *
     * @return the contents of the display
     */
    public String getDisplay() {
      return display.getText();
    }

    /**
     * Set the contents of the display
     *
     * @param text the value to place in the display
     */
    public void setDisplay( String text ) {
      display.setText( text );
    }

    /**
     * Invoked whenever a user presses a button. The button
     * press is passed to the calculator engine,
     * which processes the button and updates the
     * display if required.
     *
     * @param e the event that caused this method to be invoked
     */
    public void actionPerformed( ActionEvent e ) {
        // Get the name of the button that was pressed
        String s = e.getActionCommand();

        // Let the logic object handle it
        myCalc.handleButton( s, this ) ;
    }
```

continued

```
    /**
     * Create and display a calculator
     */
    public static void main( String args[] ) {
        Calculator calcGUI =
            new Calculator( "A Java Calculator" );
        calcGUI.setVisible( true );
    }

} // Calculator
```

[FIGURE 12-43] GUI visualization code for a four-function calculator

12.6 Summary

As we said earlier in this chapter, GUI design and implementation is a huge topic. For example, we did not even introduce the capabilities of the `Graphics` class, which allows you to create line drawings, polygons, icons, and images, as well as a range of graphical manipulations such as shading, area filling, clipping, and image painting. This information would fill another chapter or even an entire book. We leave these and many other interesting capabilities of typical GUI toolkits to later classes in computer science and graphics design. (Also, the Java Web site contains an excellent tutorial for building GUIs using Swing; see *java.sun.com/docs/books/tutorial/ui/index.html*.)

The important point to take away from this discussion is not the myriad of details we introduced. When you need answers to these questions, you can find them in the reference manual for the toolkit you are using or at a Web site such as *http://java.sun.com/j2se/1.5/docs/api/*. The truly important points to remember are the basic ideas we emphasized throughout the chapter. You should understand the inheritance hierarchy used to organize the GUI toolkit; you should feel comfortable creating components and placing them into containers; and you should know the properties of components and how to customize them—for example, their size, location, color, and font. You should be able to deal with component layout, understand the concepts of events and listeners, and be able to write event-driven programs. Once you are comfortable with these ideas, you can deal with any new and improved GUI package that comes down the road.

EXERCISES

1. Describe the differences between the following pairs of GUI components:

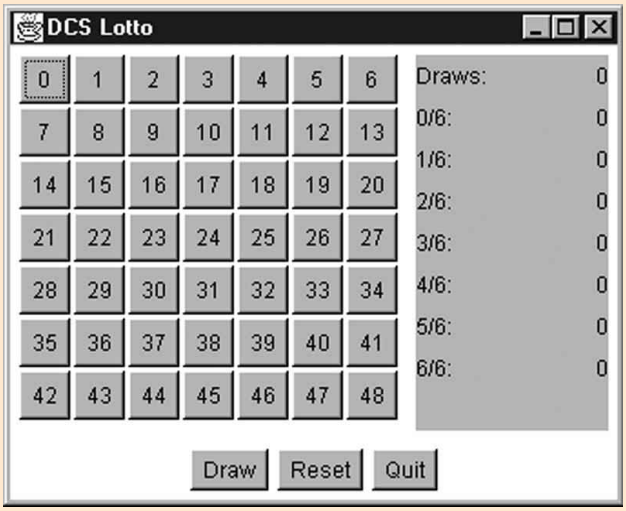
   a. Panels and frames

   b. Buttons and radio buttons

   c. Text fields and text areas

2. The standard Java distribution includes a number of demonstration programs. One of these is `SwingSet2`, which is in the `demo/jfc/swingSet2` directory where Java is installed. The `SwingSet2` program displays all the components supported by Swing and provides examples of how to use each one. Locate the `SwingSet2` program and explore Swing components that are not discussed in this chapter.

3. Describe the process you would use to create and lay out the components displayed in the following figure. Your answer does not have to include code; you will address the code in the Challenge Work Exercise at the end of the chapter.



Exercises 4 through 8 refer to the same GUI. You should do these exercises in sequence.

4. Create and display a `JFrame` object `f` that is 200 pixels wide by 250 pixels high and is labeled *My First Attempt*. Set its background color to blue and its layout manager to flow layout. After you display it on the screen, change its size to 500 × 500 and its color to red. Try some other size and color values as well. This exercise gives you some experience with the properties of `Container` objects.

5 Place a label inside `JFrame f` that says *This is a Label Object*. Use the `setFont()` and `setLocation()` methods to change both the appearance and the location of the label. This exercise gives you some experience with label components.

6 Add four buttons labeled b1, b2, b3, and b4 to the frame `f` in Exercise 5. (You do not have to write event handlers for them yet.) Note where the flow layout places the buttons. Resize the window and observe how the layout manager automatically determines the proper position of these buttons.

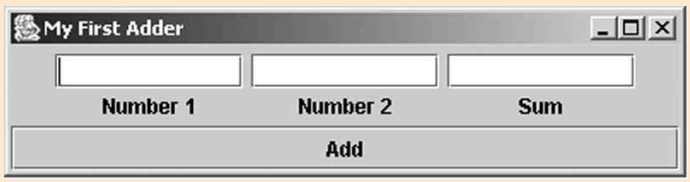
7 Change the layout manager for `f` to a grid layout. Notice where the buttons are placed in the frame. How does this compare with the flow layout in Exercise 6? What happens to the buttons when the frame is resized?

8 Change the layout manager for frame `f` to a border layout. Place the label from Exercise 5 in the North area, and place the four buttons b1, b2, b3, and b4 from Exercise 6 in the West, Central, East, and South areas, respectively.

9 Create a GUI with seven buttons that are stacked vertically down the left side of the frame. These buttons are labeled RED, BLUE, GREEN, PINK, ORANGE, CYAN, and MAGENTA. Initially, the frame has a white background. When a user clicks any of these seven buttons, the background changes to the color specified by the button.

10 Repeat the GUI described in Exercise 9, but present the list of color options in a menu rather than with a series of buttons. Your menu bar should contain menus for Application and Color. The Application menu contains two items: Reset and Quit. The Color menu contains the seven colors listed in Exercise 9. The frame starts with the background set to a default color that you determine. When the user selects a color in the Color menu, the frame's background is reset to that color. In the Application menu, the Reset option resets the frame to its default color. The Quit option terminates the GUI.

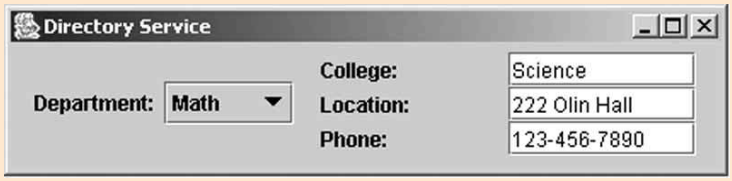
11 a Create a GUI with three text fields labeled Number 1, Number 2, and Sum, one button labeled Add, and a label field entitled *My First Adder*. The GUI layout should look like the following window:



b Enter an integer in the Number 1 text field and another in the Number 2 text field. When you click the Add button, the sum of the two values appears in the Sum field.

c Add another button labeled Clear to the right of the Add button. When the user clicks this button, all three fields are reset to blank.

12 Build a GUI that looks like the following:



The GUI contains a `JComboBox` that lists all departments at your school. (Use a subset if your institution has a huge number of departments.) When you select any department, information about it appears in the text fields on the right side of the frame. For this exercise, display the name of the college that includes the department, the building and room number of the department office, and its phone number. This is a simple example of an application called a **directory service**.

13 Write a Mouse Tracker GUI that continuously tracks the location of the cursor. Put three buttons on the screen, labeled B1, B2, and B3. Also include three text fields labeled X, Y, and Button. The X and Y text fields should constantly display the (x, y) coordinates of the cursor. The Button field is blank if the cursor is not inside any of the three buttons, or it contains the name of the button (B1, B2, or B3) in which the cursor is located. Your Mouse Tracker should look like this:

14 Implement the four mathematical functions of the calculator shown in Figures 12-42 and 12-43. This involves writing the `CalcLogic` class referenced in the code.

15 **a** A computer science student wrote the following program to experiment with the mouse listener interface. The student wanted the program to print the word *Click* when the user clicked the mouse button anywhere in the window. The program compiles and runs, but does not print anything when the mouse is clicked. Describe what is wrong with the program.

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

public class Mouse extends JFrame implements MouseListener {
         public Mouse( String title ) {
              super( title );
               setSize( 200, 200 );
    }

    public void mouseClicked( MouseEvent e ) {
        System.out.println( "Click" );
    }

    // Included to satisfy the MouseListener interface
    public void mouseEntered( MouseEvent e ) {}
    public void mouseExited( MouseEvent e ) {}
    public void mousePressed( MouseEvent e ) {}
    public void mouseReleased( MouseEvent e ) {}

    public static void main( String args[] ) {
        Mouse win = new Mouse( "A Cheesy Program" );
        win.setVisible( true );
    }
}
```

b Correct the error(s) identified in Exercise 15a so that the program behaves correctly and prints the word *Click* whenever a user clicks the mouse button.

16 Modify the program in Exercise 15 so that, instead of printing *Click*, it prints the (x, y) coordinates of the cursor location when the user clicks the mouse button.

17 The code shown at the end of this exercise creates and displays a simple graphical user interface. Take a moment to examine the code and then answer the following questions.

a What will the interface look like on the screen? Draw a sketch of the interface.

b What happens to the GUI when the user presses the Grumpy `JButton`?

c What happens when the user presses the `JButton` labeled *Mouse*?

d How would the appearance of the GUI change if you used a flow layout instead of a grid layout in the panel named `panel2`?

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

public class GrumpyGUI extends JFrame implements ActionListener
{
    private static final
        String BUTTONS[] = { "Buster", "Grumpy", "Mouse" };

    private static final
        String LABELS[] = { "Woof", "Meow", "Feed me" };

    private JLabel theLabels[] = new JLabel[ BUTTONS.length ];
    private int lastButton = 0;

    public GrumpyGUI( String title ) {
      super( title );

      JPanel panel1 = new JPanel();
      JPanel panel2 = new JPanel();
      JButton button = null;

      panel1.setLayout( new FlowLayout() );
      panel2.setLayout( new GridLayout( BUTTONS.length, 0 ) );

      for ( int i = 0; i < BUTTONS.length; i++ ) {
          button = new JButton( BUTTONS[ i ] );
          button.addActionListener( this );

          theLabels[ i ] = new JLabel( LABELS[ i ] );
          theLabels[ i ].setForeground( Color.BLACK);

          panel1.add( theLabels[ i ] );
          panel2.add( button );
      }

      setLayout( new BorderLayout() );
      add( BorderLayout.CENTER, panel2 );
      add( BorderLayout.SOUTH, panel1 );

      setSize( 150, 250 );
    }

    public void actionPerformed( ActionEvent e ) {
      int pressed = -1;
```

continued

```
      for ( int i = 0; pressed == -1; i++ ) {
          if ( BUTTONS[ i ].equals( e.getActionCommand() ) ) {
            pressed = i;
          }
      }

      theLabels[ lastButton ].setForeground( Color.BLACK );
      theLabels[ pressed ].setForeground( Color.RED );
      lastButton = pressed;
    }

    public static void main( String args[] ) {
      GrumpyGUI win = new GrumpyGUI( "Cats Rule" );
      win.setVisible( true );
    }
} // GrumpyGUI
```

18 Build a GUI for totaling a customer's bill in a convenience store. The GUI contains two areas. The area on the left, labeled ITEM, contains a set of buttons, one for each product sold in the store. The text area on the right, labeled BILL, itemizes the cost of each item purchased and displays the total. Include a Done button under the text areas.

The clerk first clicks the buttons that correspond to the items the user has purchased. As each button is clicked, the name and cost of the item appear in the text area on the right. When the clerk clicks the Done button, the cost of all the items purchased is displayed, along with a 4% sales tax and the grand total. Your GUI looks like the following:

CHALLENGE WORK EXERCISE

The GUI in Exercise 3 is an interface for a simple gambling game played in many states. You play the game by selecting six numbers between 1 and 48. The computer then draws six numbers at random in the same range. If the computer selects the same six numbers you picked, you are the big winner. There are smaller payoffs for picking 5 out of 6 numbers, 4 out of 6, and so on. The exact amounts of the payoffs depend on the generosity of the house.

You start the game by pressing the Reset button, which sets all values to 0 and all buttons to a gray background. Next, select a number by clicking any of the numbered buttons, 1 to 48. When you select a number, its button turns red. If you make a mistake or change your mind, you unselect a button by clicking it again, and it resets to gray. Repeat this operation until you are satisfied with your choices and all six of them are highlighted in red.

After you select your six numbers, click the Draw button. The counter opposite the line labeled *Draws* increases by 1, and the computer now selects six numbers at random and displays them on the board. If you and the computer select the same number, the corresponding button turns green. Any number the computer selects that you did not is colored yellow. After choosing six values, the computer counts the number of matches (green buttons) and increments the counter opposite that row—for example, if you matched two numbers, the counter opposite the label 2/6 is incremented by 1. If you want to play again, click the Draw button to repeat the process.

You can play as many times as you want. When you finish, click the Quit button; the computer determines how much you have won or lost based on the following formula:

Each draw:	Costs 1 unit
0/6	Returns 0 units
1/6	Returns 0 units
2/6	Returns 1 unit
3/6	Returns 5 units
4/6	Returns 50 units
5/6	Returns 500 units
6/6	Returns 50,000 units

The amount won or lost (in units) is displayed below the 6/6 line in the right text area, preceded by the text *The number of units won is* or *The number of units lost is*.

Write a Java program to implement the interface and play the game we just described. After you write the program and test it thoroughly, play the game several times and determine whether the payoffs listed above are "fair," in the sense that they don't favor the player or the house. That is, after you play many games, the amount won or lost should be close to zero. If you think the results are unfair, try other payoffs to develop a formula that favors neither the player nor the house.