

16.1 EVENT-DRIVEN PROGRAMMING 783

Events and Listeners 783

**16.2 BUTTONS, EVENTS, AND OTHER SWING
BASICS** 784

Example: A Simple Window 785

Pitfall: Forgetting to Program the Close-Window
Button 790Pitfall: Forgetting to Use `getContentPane` 791

Buttons 792

Action Listeners and Action Events 793

Pitfall: Changing the Heading for
`actionPerformed` 795

Tip: Ending a Swing Program 796

Example: A Better Version of Our First Swing GUI 796

Labels 800

Color 800

Example: A GUI with a Label and Color 802

16.3 CONTAINERS AND LAYOUT MANAGERS 805

Border Layout Managers 805

Flow Layout Managers 809

Grid Layout Managers 810

Panels 815

Example: A Tricolor Built with Panels 815

The `Container` Class 816

Tip: Code a GUI's Look and Actions Separately 823

The Model-View-Controller Pattern ❖ 823

16.4 MENUS AND BUTTONS 825

Example: A GUI with a Menu 825

Menu Bars, Menus, and Menu Items 825

Nested Menus ❖ 831

The `AbstractButton` Class 832The `setActionCommand` Method 833

Listeners As Inner Classes ❖ 835

16.5 TEXT FIELDS AND TEXT AREAS 838

Text Areas and Text Fields 838

Tip: Labeling a Text Field 845

Tip: Inputting and Outputting Numbers 845

A Swing Calculator 846

CHAPTER SUMMARY 852**ANSWERS TO SELF-TEST EXERCISES** 852**PROGRAMMING PROJECTS** 860

It Don't Mean a Thing If It Ain't Got That Swing

Song Title, *Duke Ellington*

INTRODUCTION

Swing GUI This is the first of three chapters that present the basic classes in the **Swing** package and teach the basic techniques for using these classes to define **GUIs**. **GUIs** are windowing interfaces that handle user input and output. **GUI** is pronounced “gooey” and stands for **graphical user interface**. Entire books have been written on Swing, so we will not have room to give you a complete description of Swing in three chapters. However, we will teach you enough to allow you to write a variety of windowing interfaces.

GUI

Windowing systems that interact with the user are often called **GUIs**. **GUI** is pronounced “gooey” and stands for **graphical user interface**.

AWT The **AWT** (**Abstract Window Toolkit**) package is an older package designed for doing windowing interfaces. Swing can be viewed as an improved version of the AWT. However, Swing did not completely replace the AWT package. Some AWT classes are replaced by Swing classes, but other AWT classes are needed when using Swing. We will use classes from both Swing and the AWT.

Swing GUIs are designed using a particular form of object-oriented programming that is known as *event-driven programming*. Our first section begins with a brief overview of event-driven programming.

PREREQUISITES

Before covering this chapter (and the next two chapters on applets and more Swing), you need to have covered Chapters 1 through 5, Chapters 7 (inheritance), Chapter 13 (interfaces and inner classes), and Section 8.2 of Chapter 8 (abstract classes). (Section 8.2 of Chapter 8 does not require Section 8.1.) Except for one subsection at the end of this chapter, you need not have read any of the other chapters that precede this chapter.

To cover the last subsection entitled “A Swing Calculator,” you need to first read Chapter 9, which covers exceptions. If you have not yet read Chapter 9, you can skip that last section.

16.1

Event-Driven Programming

My duty is to obey orders.

Thomas Jonathan (Stonewall) Jackson

Event-driven programming is a programming style that uses a signal-and-response approach to programming. Signals to objects are things called *events*, a concept we explain in this section.

event-driven
programming

EVENTS AND LISTENERS

Swing programs use events and event handlers. An **event** is an object that acts as a signal to another object known as a **listener**. The sending of the event is called **firing the event**. The object that fires the event is often a GUI component, such as a button. The button fires the event in response to being clicked. The listener object performs some action in response to the event. For example, the listener might place a message on the screen in response to a particular button being clicked. A given component may have any number of listeners, from zero to several listeners. Each listener might respond to a different kind of event, or multiple listeners might respond to the same events.

event
listener
firing an event

If you have read Chapter 9 on exception handling, then you have already seen one specialized example of event-driven programming.¹ An exception object is an event. The throwing of an exception is an example of firing an event (in this case firing the exception event). The listener is the `catch` block that catches the event.

In Swing GUIs an event often represents some action such as clicking a mouse, dragging the mouse, pressing a key on the keyboard, clicking the close-window button on a window, or any other action that is expected to elicit a response. A listener object has methods that specify what will happen when events of various kinds are received by the listener. These methods that handle events are called **event handlers**. You the programmer will define (or redefine) these event-handler methods. The relationship between an event-firing object, such as a button, and its event-handling listener is shown diagrammatically in Display 16.1.

event handler

Event-driven programming is very different from most programming you've seen before now. All our previous programs consisted of a list of statements executed in order. There were loops that repeat statements and branches that choose one of a list of statements to execute next. However, at some level, each run of a program consists of a list of statements performed by one agent (the computer) that executes the statements one after the other in order.

Event-driven programming is a very different game. In event-driven programming, you create objects that can fire events, and you create listener objects to react to the events. For the most part, your program does not determine the order in which things happen. The events determine that order. When an event-driven program is running,

¹ If you have not yet covered Chapter 9 on exceptions, you can safely ignore this paragraph.

Display 16.1 Event Firing and an Event Listener

The component (for example, a button) fires an event.



This listener object invokes an event handler method with the `event` as an argument.

the next thing that happens depends on the next event. It's as though the listeners were robots that interact with other objects (possibly other robots) in response to events (signals) from these other objects. You program the robots, but the environment and other robots determine what any particular robot will actually end up doing.

If you have never done event-driven programming before, one aspect of it may seem strange to you: *You will be writing definitions for methods that you will never invoke in any program.* This will likely feel a bit strange at first, because a method is of no value unless it is invoked. So, somebody or something other than you, the programmer, must be invoking these methods. That is exactly what does happen. The Swing system automatically invokes certain methods when an event signals that the method needs to be called.

Event-driven programming with the Swing library makes extensive use of inheritance. The classes you define will be derived classes of some basic Swing library classes. These derived classes will inherit methods from their base class. For many of these inherited methods, library software will determine when these methods are invoked, but you will override the definition of the inherited method to determine what will happen when the method is invoked.

16.2

Buttons, Events, and Other Swing Basics

One button click is worth a thousand key strokes.

Anonymous

In this section we present enough about Swing to allow you to do some simple GUI programs.

Example

A SIMPLE WINDOW

Display 16.2 contains a Swing program that produces a simple window. The window contains nothing but a button on which is written "Click to end program.". If the user follows the instructions and clicks the button with his or her mouse, the program ends.

The `import` statements give the names of the classes used and which package they are in. What we and others call the *Swing library* is the package named `javax.swing`. The *AWT library* is the package `java.awt`. Note that one package name contains an "x" and one does not.

This program is a simple class definition with only a `main` method. The first line in the `main` method creates an object of the class `JFrame`. That line is reproduced below:

```
JFrame firstWindow = new JFrame();
```

This is an ordinary declaration of a variable named `firstWindow` and an invocation of the no-argument constructor for the class `JFrame`. A `JFrame` object is a basic window. A `JFrame` object includes a border and the usual three buttons for minimizing the window down to an icon, changing the size of the window, and closing the window. These buttons are shown in the upper-right corner of the window, which is typical, but if your operating system normally places these buttons someplace else, that is where they will likely be located in a `JFrame` on your computer.

`javax.swing`

`java.awt`

`JFrame`

JFrame

An object of the class `JFrame` is what you think of as a window. It automatically has a border and some basic buttons for minimizing the window and similar actions. As you will see, a `JFrame` object can have buttons and many other components added to the window and programmed for action.

The initial size of the `JFrame` window is set using the `JFrame` method `setSize`, as follows:

```
firstWindow.setSize(WIDTH, HEIGHT);
```

In this case `WIDTH` and `HEIGHT` are defined `int` constants. The units of measure are pixels, so the window produced is 300 pixels by 200 pixels. (The term *pixel* is defined in the box entitled "Pixel.") As with other windows, you can change the size of a `JFrame` by using your mouse to drag a corner of the `JFrame` window.

The buttons for minimizing the window down to an icon and for changing the size of the window behave as they do in any of the other windows you have used. The minimization button shrinks the window down to an icon. (To restore the window, you click the icon.) The second button

`setSize`

Display 16.2 A First Swing Demonstration Program (Part 1 of 2)

```
1 import javax.swing.JFrame;
2 import javax.swing.JButton;

3 public class FirstSwingDemo
4 {
5     public static final int WIDTH = 300;
6     public static final int HEIGHT = 200;

7     public static void main(String[] args)
8     {
9         JFrame firstWindow = new JFrame();
10        firstWindow.setSize(WIDTH, HEIGHT);

11        firstWindow.setDefaultCloseOperation(
12            JFrame.DO_NOTHING_ON_CLOSE);

13        JButton endButton = new JButton("Click to end program.");
14        EndingListener buttonEar = new EndingListener();
15        endButton.addActionListener(buttonEar);
16        firstWindow.getContentPane().add(endButton);

17        firstWindow.setVisible(true);
18    }
19 }
```

This program is not typical of the style we will use in Swing programs.

This is the file FirstSwingDemo.java.

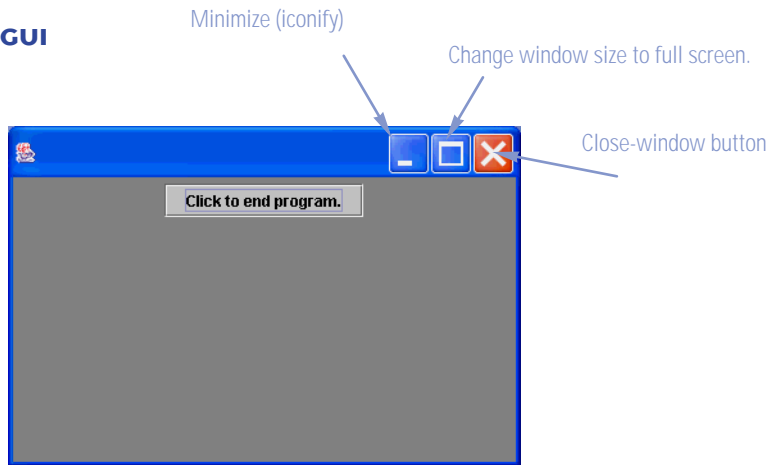
This is the file EndingListener.java.

```
1 import java.awt.event.ActionListener;
2 import java.awt.event.ActionEvent;

3 public class EndingListener implements ActionListener
4 {
5     public void actionPerformed(ActionEvent e)
6     {
7         System.exit(0);
8     }
9 }
```

Display 16.2 A First Swing Demonstration Program (Part 2 of 2)

RESULTING GUI



PIXEL

A **pixel** is the smallest unit of space on which your screen can write. With Swing, both the size and the position of objects on the screen are measured in pixels. The more pixels you have on a screen, the greater the screen resolution.

pixel

RESOLUTION'S RELATIONSHIP TO OBJECT SIZE

The relationship between resolution and size can seem confusing at first. A high-resolution screen is a screen of better quality than a low-resolution screen, so why does an object look smaller on a high-resolution screen and larger on a low-resolution screen? Consider a very simple case, namely a one-pixel "dot." For a screen of fixed size, if there are very many pixels (high resolution), then the one-pixel dot will be very small. If there are fewer pixels (low resolution) for the same size screen, then each pixel must be larger since the smaller number of pixels cover the same screen. So, if there are fewer pixels, the one-pixel dot will be larger. Similarly, a two-pixel figure or a figure of any number of pixels will look larger on a low-resolution (fewer pixels) screen.

changes the size of the window back and forth from full screen to a smaller size. The close-window button can behave in different ways depending on how it is set by your program.

The behavior of the close-window button is set with the `JFrame` method `setDefaultCloseOperation`. The line of the program that sets the behavior of the close-window button is reproduced below:

```
firstWindow.setDefaultCloseOperation(
    JFrame.DO_NOTHING_ON_CLOSE);
```

In this case the argument `JFrame.DO_NOTHING_ON_CLOSE` is a defined constant named `DO_NOTHING_ON_CLOSE`, which is defined in the `JFrame` class. This sets the close-window button so that when it is clicked nothing happens (unless we programmed something to happen, which we have not done). Other possible arguments are given in Display 16.3.

Display 16.3 Some Methods in the Class `JFrame` (Part 1 of 2)

The class `JFrame` is in the `javax.swing` package.

```
public JFrame()
```

Constructor that creates an object of the class `JFrame`.

```
public JFrame(String title)
```

Constructor that creates an object of the class `JFrame` with the title given as the argument.

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

Sets the action that will happen by default when the user clicks the close-window button. The argument should be one of the following defined constants:

`JFrame.DO_NOTHING_ON_CLOSE`: Do nothing. The `JFrame` does nothing, but if there are any registered window listeners, they are invoked. (Window listeners are explained in Chapter 18.)

`JFrame.HIDE_ON_CLOSE`: Hide the frame after invoking any registered `WindowListener` objects.

`JFrame.DISPOSE_ON_CLOSE`: Hide and *dispose* the frame after invoking any registered window listeners. When a window is **disposed** it is eliminated but the program does not end. To end the program, you use the next constant as an argument to `setDefaultCloseOperation`.

`JFrame.EXIT_ON_CLOSE`: Exit the application using the `System.exit` method. (Do not use this for frames in applets. Applets are discussed in Chapter 17.)

If no action is specified using the method `setDefaultCloseOperation`, then the default action taken is `JFrame.HIDE_ON_CLOSE`.

Throws an `IllegalArgumentException` if the argument is not one of the values listed above. ^a

Throws a `SecurityException` if the argument is `JFrame.EXIT_ON_CLOSE` and the Security Manager will not allow the caller to invoke `System.exit`. (You are not likely to encounter this case.)

^a If you have not yet covered Chapter 9 on exceptions, you can safely ignore all references to “throwing exceptions.”

Display 16.3 Some Methods in the Class JFrame (Part 2 of 2)

```
public void setSize(int width, int height)
```

Sets the size of the calling frame so that it has the width and height specified. Pixels are the units of length used.

```
public void setTitle(String title)
```

Sets the title for this frame to the argument string.

```
public Container getContentPane()
```

Returns the content pane of the calling JFrame object. Container is a class in the package java.awt.

To add a component to the JFrame use

```
getContentPane().add(Component componentAdded)
```

You do not use the add method directly on a JFrame, but instead use add with the content pane of the JFrame. Use of the add method with a JFrame calling object will produce a run-time error.

To set the layout manager use

```
getContentPane().setLayout(LayoutManager manager)
```

Layout managers are discussed later in this chapter.

```
public void setJMenuBar(JMenuBar menubar)
```

Sets the menubar for the calling frame. (Menus and menu bars are discussed later in this chapter.)

```
public void dispose()
```

Eliminates the calling frame and all its subcomponents. Any memory they use is released for reuse. If there are items left (items other than the calling frame and its subcomponents), then this does not end the program. (The method dispose is discussed in Chapter 18.)

The method `setDefaultCloseOperation` takes a single `int` argument, and each of the constants described in Display 16.3 is an `int` constant. However, do not think of them as `int` values. Think of them as policies for what happens when the user clicks the close-window button. It was convenient to name these policies by `int` values. However, they could just as well have been named by `char` values or `String` values or something else. The fact that they are `int` values is an incidental detail of no real importance.

Descriptions of some of the most important methods in the class `JFrame` are given in Display 16.3. Some of these methods will not be explained until later in this chapter. A more complete list of methods for the class `JFrame` is given in Appendix 4.

A `JFrame` can have components added, such as buttons, menus, and text labels. However, things are not added directly to the `JFrame`. Instead, they are added to something called the *content*

content pane

getContentPane

pane of the `JFrame`. A `JFrame` has various layers that can have components added to them. The main layer, and the only one we will use, is called the **content pane**. For our purposes you can think of the content pane as the “inside” of the `JFrame`. You retrieve the content pane of the `JFrame` with the accessor method `getContentPane`; you then add components to the content pane using the method `add` of the content pane. In Display 16.2 these two actions are combined into the following line, which adds the `JButton` object `endButton` to the (content pane of the) `JFrame`:

```
firstWindow.getContentPane().add(endButton);
```

The description of how the `JButton` named `endButton` is created and programmed will be given in the two subsections entitled “Buttons” and “Action Listeners and Action Events” a little later in this section. We end this subsection by jumping ahead to the last line of the program, which is

```
firstWindow.setVisible(true);
```

setVisible

This makes the `JFrame` window visible on the screen. At first glance this may seem strange. Why not have windows automatically become visible. Why would you create a window if you did not want it to be visible? The answer is that you may not want it to be visible at all times. You have certainly experienced windows that disappear and reappear. To hide the window, which is not desirable in this example, you would replace the argument `true` with `false`.

Pitfall

FORGETTING TO PROGRAM THE CLOSE-WINDOW BUTTON

The following lines from Display 16.2 ensure that when the user clicks the close-window button, nothing happens:

```
firstWindow.setDefaultCloseOperation(  
    JFrame.DO_NOTHING_ON_CLOSE);
```

If you forget to program the close-window button, then the default action is as if you had set it the following way:

```
firstWindow.setDefaultCloseOperation(  
    JFrame.HIDE_ON_CLOSE);
```

In the program in Display 16.2 this would mean that if the user clicks the close-window button, the window will hide (become invisible and inaccessible), but the program will not end, which is a pretty bad situation. Since the window would be hidden, there would be no way to click the “Click to end program.” button. You would need to use some operating system command that forces the program to end. That is an operating system topic, not a Java topic, and the exact command depends on which operating system you are using.

THE setVisible METHOD

Many classes of Swing objects have a `setVisible` method. The `setVisible` method takes one argument of type `boolean`. If `w` is an object, such as a `JFrame` window, that can be displayed on the screen, then the call

```
w.setVisible(true);
```

will make `w` visible. The call

```
w.setVisible(false);
```

will hide `w`.

SYNTAX:

```
Object_For_Screen.setVisible(Boolean_Expression);
```

EXAMPLE (FROM DISPLAY 16.2):

```
public static void main(String[] args)
{
    JFrame firstWindow = new JFrame();
        .
        .
        .
    firstWindow.setVisible(true);
}
```

Pitfall

FORGETTING TO USE getContentPane

Recall that in Display 16.2 we added the button `endButton` to the `JFrame` named `firstWindow` as follows:

```
firstWindow.getContentPane().add(endButton);
```

Because you are “adding `endButton` to `firstWindow`,” you might be tempted to use the following instead:

```
firstWindow.add(endButton);
```

However, if you omit `getContentPane()`, your program will not work correctly. Moreover, the compiler will not warn you about this mistake. You will, however, get a run-time error message.

Self-Test Exercises

1. What Swing class do you normally use to define a window? Any window class that you define would normally be an object of this class.
2. What units of measure are used in the following call to `setSize` that appeared in the `main` method of the program in Display 16.2? In other words, 300 what? Inches? Centimeters? Light years? And similarly, 200 what?

```
firstWindow.setSize(WIDTH, HEIGHT);
```

which is equivalent to

```
firstWindow.setSize(300, 200);
```

3. What is the method call to set the close-window button of the `JFrame` `someWindow` so that nothing happens when the user clicks the close-window button in `someWindow`?
4. What is the method call to set the close-window button of the `JFrame` `someWindow` so that the program ends when the user clicks the close-window button in `someWindow`?
5. What happens when you click the minimizing button of the `JFrame` shown in Display 16.2?
6. Suppose `someWindow` is a `JFrame` and `n` is an `int` variable with some value. Give a Java statement that will make `someWindow` visible if `n` is positive and hide `someWindow` otherwise.

■ BUTTONS

A button object is created in the same way that any other object is created, but you use the class `JButton`. For example, the following example from Display 16.2 creates a button:

`JButton`

```
JButton endButton = new JButton("Click to end program.");
```

The argument to the construct, in this case "Click to end program.", is a string that will be written on the button when the button is displayed. If you look at the picture of the GUI in Display 16.2, you will see that the button is labeled "Click to end program."

adding a button

We have already discussed adding components, such as buttons, to a `JFrame`. The button is added to the content pane of the `JFrame` by the following line from Display 16.2:

```
firstWindow.getContentPane().add(endButton);
```

In the next subsection we explain the lines from Display 16.2 involving the method `addActionListener`.

THE JButton CLASS

An object of the class `JButton` is displayed in a GUI as a component that looks like a button. You click the button with your mouse to simulate pushing it. When creating an object of the class `JButton` using `new`, you can give a string argument to the constructor and the string will be displayed on the button.

You can add `JButton` objects to a `JFrame` by using the method `add` to add the button *to the content pane* of the `JFrame`. You will later see that you can also add buttons to other GUI objects (known as “containers”) in a similar way.

A button’s action is programmed by registering a listener with the button using the method `addActionListener`.

EXAMPLE:

```
JButton niceButton = new JButton("Click here");
niceButton.addActionListener(new SomeActionListenerClass());
someJFrame.getContentPane().add(niceButton);
```

THE CLOSE-WINDOW BUTTON IS NOT IN THE CLASS JButton

The buttons that you add to a GUI are all objects of the class `JButton`. The close-window button and the other two accompanying buttons are not objects of the class `JButton`. They are part of the `JFrame` object.

ACTION LISTENERS AND ACTION EVENTS

Clicking a button with your mouse (or activating certain other items in a GUI) creates an object known as an event and sends the event object to another object (or objects) known as the listener(s). This is called firing the event. The listener then performs some action. When we say that the event is “sent” to the listener object, what we really mean is that some method in the listener object is invoked with the event object as the argument. This invocation happens automatically. Your Swing GUI class definition will not normally contain an invocation of this method. However, your Swing GUI class definition does need to do two things:

First, for each button, it needs to specify what objects are listeners that will respond to events fired by that button; this is called **registering** the listener.

Second, it must define the methods that will be invoked when the event is sent to the listener. Note that these methods will be defined by you, but in normal circumstances, you will never write an invocation of these methods. The invocations will take place automatically.

registering a
listener

addAction-
Listener

The following lines from Display 16.2 create an `EndingListener` object named `buttonEar` and register `buttonEar` as a listener to receive events from the button named `endButton`:

```
EndingListener buttonEar = new EndingListener();
endButton.addActionListener(buttonEar);
```

The second line says that `buttonEar` is registered as a listener to `endButton`, which means `buttonEar` will receive all events fired by `endButton`.

action event
action listener

Different kinds of components require different kinds of listener classes to handle the events they fire. A button fires events known as **action events**, which are handled by listeners known as **action listeners**.

Action-
Listener

An action listener is an object whose class implements the `ActionListener` interface. For example, the class `EndingListener` in Display 16.2 implements the `ActionListener` interface. The `ActionListener` interface has only one method heading that must be implemented, namely the following:

action-
Performed

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

In the class `EndingListener` in Display 16.2, the `actionPerformed` method is defined as follows:

```
public void actionPerformed(ActionEvent e)
{
    System.exit(0);
}
```

If the user clicks the button `endButton`, that sends an action event to the action listener for that button. But `buttonEar` is the action listener for the button `endButton`, so the action event goes to `buttonEar`. When an action listener receives an action event, the event is automatically passed as an argument to the method `actionPerformed` and the method `actionPerformed` is invoked. If the event is called `e`, then the following invocation takes place in response to `endButton` firing `e`:

```
buttonEar.actionPerformed(e);
```

In this case the parameter `e` is ignored by the method `actionPerformed`. The method `actionPerformed` simply invokes `System.exit` and thereby ends the program. So, if the user clicks `endButton` (the one labeled "Click to end program."), the net effect is to end the program and so the window goes away.

Note that you never write any code that says

```
buttonEar.actionPerformed(e);
```

This action does happen, but the code for this is embedded in some class definition inside the Swing and/or AWT libraries. Somewhere the code says something like

```
bla.actionPerformed(e);
```

and somehow `buttonEar` gets plugged in for the parameter `bla` and this invocation of `actionPerformed` is executed. But, all this is done for you. All you do is define the method `actionPerformed` and register `buttonEar` as a listener for `endButton`.

Note that the method `actionPerformed` must have a parameter of type `ActionEvent`, even if your definition of `actionPerformed` does not use this parameter. This is because the invocations of `actionPerformed` were already programmed for you and so must allow the possibility of using the `ActionEvent` parameter `e`. As you will see, in other Swing GUIs the method `actionPerformed` does often use the event `e` to determine which button was clicked. This first example is a special, simple case because there is only one button. Later in this chapter we will say more about defining the `actionPerformed` method in more complicated situations.

Pitfall

CHANGING THE HEADING FOR `actionPerformed`

When you define the method `actionPerformed` in an action listener, you are implementing the method heading for `actionPerformed` that is specified in the `ActionListener` interface. Thus, the header for the method `actionPerformed` is determined for you, and you cannot change the heading. It must have exactly one parameter, and that parameter must be of type `ActionEvent`, as in the following:

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

If you change the type of the parameter or if you add (or subtract) a parameter, you will not have given a correct definition of an action listener.² The only thing you can change is the name of the parameter `e`, since it is just a placeholder. So the following change is acceptable:

```
public void actionPerformed(ActionEvent theEvent)
```

Of course, if you make this change, then inside the body of the method `actionPerformed`, you will use the identifier `theEvent` in place of the identifier `e`.

You also cannot add a `throws` clause to the method `actionPerformed`.³ If a checked exception is thrown in the definition of `actionPerformed`, then it must be caught in the method `actionPerformed`. (Recall that a checked exception is one that must be either caught in a `catch` block or declared in a `throws` clause.)

² Although it would be rather questionable style, you can overload the method named `actionPerformed` so that you have multiple versions of the method `actionPerformed`, each with a different parameter list. But only the version of `actionPerformed` shown above has anything to do with making a class into an action listener.

³ If you have not yet covered exception handling (Chapter 9), you can safely ignore this paragraph.

Tip**ENDING A SWING PROGRAM**

A GUI program is normally based on a kind of infinite loop. There may not be a Java loop statement in a GUI program, but nonetheless the GUI program need not ever end. The windowing system normally stays on the screen until the user indicates that it should go away (for example, by clicking the "Click to end program." button in Display 16.2). If the user never asks the windowing system to go away, it will never go away. When you write a Swing GUI program, you need to use `System.exit` to end the program when the user (or something else) says it is time to end the program. Unlike the kinds of programs we saw before this chapter, a Swing program will not end after it has executed all the code in the program. A Swing program does not end until it executes a `System.exit`.⁴

`System.exit`

Self-Test Exercises

7. What kind of event is fired when you click a `JButton`?
8. What method heading must be implemented in a class that implements the `ActionListener` interface?
9. Change the program in Display 16.2 so that the window displayed has the title "My First Window". Hint: Consult the description of constructors in Display 16.3.

Example**A BETTER VERSION OF OUR FIRST SWING GUI**

Display 16.4 is a rewriting of the demonstration program in Display 16.2 that includes a few added features. This new version produces a window that is similar to the one produced by the program in Display 16.2. However, this new version is done in the style you should follow in writing your own GUIs. Notice that the window is produced by defining a class (`FirstWindow`) whose objects are windows of the kind we want. The window is then displayed by a program (`DemoWindow`) that uses the class `FirstWindow`.

⁴ As we will see when we discuss more possible arguments for `setDefaultCloseOperation`, the `System.exit` may be in some library code and need not be explicitly given in your code.

Display 16.4 The Normal Way to Define a JFrame (Part 1 of 2)

```
1  import javax.swing.JFrame;
2  import javax.swing.JButton;

3  public class FirstWindow extends JFrame
4  {
5      public static final int WIDTH = 300;
6      public static final int HEIGHT = 200;

7      public FirstWindow()
8      {
9          super();
10         setSize(WIDTH, HEIGHT);

11         setTitle("First Window Class");

12         setDefaultCloseOperation(
13             JFrame.DO_NOTHING_ON_CLOSE);

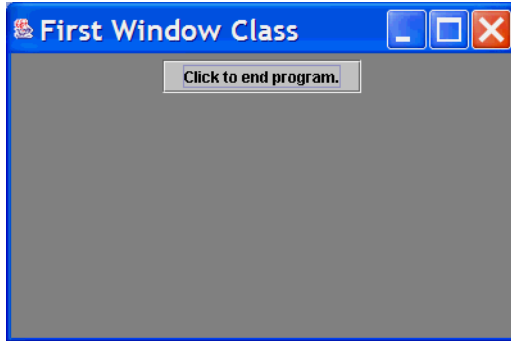
14         JButton endButton = new JButton("Click to end program.");
15         endButton.addActionListener(new EndingListener());
16         getContentPane().add(endButton);
17     }
18 }
```

The class `EndingListener` is defined in Display 16.2.

This is the file `FirstWindow.java`.

This is the file `DemoWindow.java`.

```
1  public class DemoWindow
2  {
3      public static void main(String[] args)
4      {
5          FirstWindow w = new FirstWindow();
6          w.setVisible(true);
7      }
8  }
```

Display 16.4 The Normal Way to Define a JFrame (Part 2 of 2)**RESULTING GUI**

derived class

Observe that `FirstWindow` is a derived class of the class `JFrame`. This is the normal way to define a windowing interface. The base class `JFrame` gives some basic window facilities, and then the derived class adds whatever additional features you want in your window interface.

constructor

Note that the constructor in Display 16.4 starts by calling the constructor for the parent class `JFrame` with the line

```
super();
```

As we noted in Chapter 7, this ensures that any initialization that is normally done for all objects of type `JFrame` will in fact be done. If the base class constructor you call has no arguments, then it will be called automatically, whether you include `super()`; or not, so we could have omitted the invocation of `super()` in Display 16.4. However, if the base class constructor needs an argument, as it may in some other situations, then you must include a call to the base class constructor, `super`.

Note that almost all the initializing for the window `FirstWindow` in Display 16.4 is placed in the constructor for the class. That is as it should be. The initialization, such as setting the initial window size, should be part of the class definition and not actions performed by objects of the class (as they were in Display 16.2). All the initializing methods, such as `setSize`, `setDefaultCloseOperation`, and `getContentPane`, are inherited from the class `JFrame`. Because they are invoked in the constructor for the window, the window itself is the calling object. In other words, a method invocation such as

```
setSize(WIDTH, HEIGHT);
```

is equivalent to

```
this.setSize(WIDTH, HEIGHT);
```

Similarly, the method invocations

```
setDefaultCloseOperation(  
    JFrame.DO_NOTHING_ON_CLOSE);
```

and

```
getContentPane().add(endButton);
```

are equivalent to

```
this.setDefaultCloseOperation(  
    JFrame.DO_NOTHING_ON_CLOSE);
```

and

```
this.getContentPane().add(endButton);
```

In the class `FirstWindow` (Display 16.4) we added the title "First Window Class" to the window as follows:

```
setTitle("First Window Class");
```

You can see where the title is displayed in a `JFrame` by looking at the picture of the GUI given in Display 16.4.

One thing we did differently in Display 16.4 from Display 16.2 is to use an anonymous object

```
endButton.addActionListener(new EndingListener());
```

instead of the following, which we used in Display 16.2:

```
EndingListener buttonEar = new EndingListener();  
endButton.addActionListener(buttonEar);
```

In Display 16.2 we were trying to be extra clear and so we used these two steps. However, it makes more sense to use the anonymous object `new EndingListener()` since this listener object is never referenced again and so does not need a name.

The program `DemoWindow` in Display 16.4 simply displays an object of the class `FirstWindow` on the screen.

Almost all of the initialization details for the window in Display 16.4 have been moved to the constructor for the class `FirstWindow`. However, we have placed the invocations of the method `setVisible` in the application program that uses the window class `FirstWindow`. We could have placed an invocation of `setVisible` in the constructor for `FirstWindow` and omitted the invocation of `setVisible` from the application program `DemoWindow` (Display 16.4). If we had done so, we would have produced the same results when we ran the application program. However, in normal situations, the application program knows when the window should be displayed, so it is normal to put the invocation of the method `setVisible` in the application program. The programmer writing the class `FirstWindow` cannot anticipate when a programmer who uses the window will want to make it visible (or hide it).

`setTitle`

JFrame CLASSES

When we say that a class is a **JFrame class** we mean the class is a descendent class of the class `JFrame`. For example, the class `FirstWindow` in Display 16.4 is a `JFrame` class. When we say an object is a **JFrame** we mean that it is an instance of a `JFrame` class.

Self-Test Exercises

10. Change the program in Display 16.4 so that the title of the `JFrame` is not set by the method `setTitle` but is instead set by the call to the base class constructor. Hint: Recall Self-Test Exercise 9.
11. Change the program in Display 16.4 so that there are two ways to end the GUI program: The program can be ended by either clicking the "Click to end program." button or clicking the close-window button.

LABELS

We have seen how to add a button to a `JFrame`. If you want to add some text to your `JFrame`, use a label instead of a button. A **label** is an object of the class `JLabel`. A label is little more than a line of text. The text for the label is given as an argument to the `JLabel` constructor as follows:

```
JLabel greeting = new JLabel("Hello");
```

The label `greeting` can then be added to a `JFrame` just as a button is added. For example, the following might appear in a constructor for a derived class of `JFrame`:

```
JLabel greeting = new JLabel("Hello");
getContentPane().add(greeting);
```

The next Programming Example includes a label in a `JFrame` GUI.

COLOR

You can set the color of a `JFrame` (or other GUI object). To set the background color of a `JFrame`, you use the method `setBackground`, which all derived classes of `JFrame` inherit from `JFrame`. For example, the following will set the color of the `JFrame` named `someFrame` to blue:

```
someFrame.setBackground(Color.BLUE);
```

Alternatively, if you set the color in the constructor for the `JFrame`, the invocation takes the form

```
setBackground(Color.BLUE);
```

label
JLabel

setBackground

Color.BLUE

THE JLabel CLASS

An object of the class `JLabel` is little more than one line of text that can be added to a `JFrame` (or, as we will see, added to certain other objects).

EXAMPLE: (INSIDE A CONSTRUCTOR FOR A DERIVED CLASS OF JFrame):

```
JLabel myLabel = new JLabel("Hi Mom!");  
getContentPane().add(myLabel);
```

Note that you use `getContentPane().add` to add a `JLabel` to a `JFrame`, as illustrated in the example above.

which is equivalent to

```
this.setBackground(Color.BLUE);
```

Display 16.6 gives an example of a `JFrame` object (in fact two of them) with color.

What kind of thing is a **color** when used in a Java Swing class? Like everything else in Java, a color is an object—in this case, an object that is an instance of the class `Color`. The class `Color` is in the `java.awt` package. (Note that the package name is `java.awt`, not `javax.awt`.)

In a later chapter you will see how you can define your own colors, but for now we will use the colors that are already defined for you, such as `Color.BLUE`, which is a constant named `BLUE` that is defined in the class `Color`. The constant, of course, represents the color blue. If you set the background of a `JFrame` to `Color.BLUE`, then the `JFrame` will have a blue background. The color constant `Color.BLUE` and other such constants are defined in the class `Color` and their type is `Color`. The list of color constants that are defined for you are given in Display 16.5. The next Programming Example has an example of a constructor with one parameter of type `Color`.

Display 16.5 The Color Constants

<code>Color.BLACK</code>	<code>Color.MAGENTA</code>
<code>Color.BLUE</code>	<code>Color.ORANGE</code>
<code>Color.CYAN</code>	<code>Color.PINK</code>
<code>Color.DARK_GRAY</code>	<code>Color.RED</code>
<code>Color.GRAY</code>	<code>Color.WHITE</code>
<code>Color.GREEN</code>	<code>Color.YELLOW</code>
<code>Color.LIGHT_GRAY</code>	

The class `Color` is in the `java.awt` package.

Example

A GUI WITH A LABEL AND COLOR

Display 16.6 shows a class for GUIs with a label and a background color. We have already discussed the use of color for this window. The label is used to display the text string "Close-window button works.". The label is created as follows:

```
JLabel aLabel = new JLabel("Close-window button works.");
```

The label is added to the content pane in the way we described in the subsection entitled "Labels," which is the same as the way a button is added, but in this example we have given the content pane a name with the following:

```
Container contentPane = getContentPane();
```

Display 16.6 A JFrame with Color (Part 1 of 2)

```

1  import javax.swing.JFrame;
2  import javax.swing.JLabel;
3  import java.awt.Color;
4  import java.awt.Container;
5
6  public class ColoredWindow extends JFrame
7  {
8      public static final int WIDTH = 300;
9      public static final int HEIGHT = 200;
10
11     public ColoredWindow(Color theColor)
12     {
13         super("No Charge for Color");
14         setSize(WIDTH, HEIGHT);
15         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16
17         Container contentPane = getContentPane();
18         contentPane.setBackground(theColor);
19
20         JLabel aLabel = new JLabel("Close-window button works.");
21         contentPane.add(aLabel);
22     }
23
24     public ColoredWindow()
25     {
26         this(Color.PINK);
27     }
28 }

```

This means the program will end when the user clicks the close-window button.

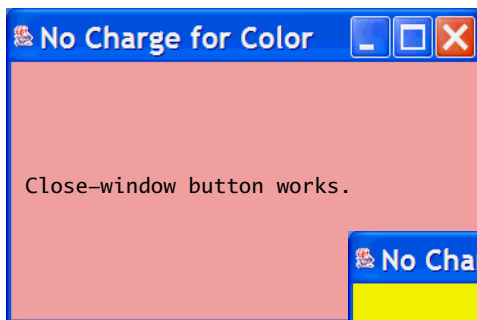
The content pane is of type Container.

This is an invocation of the other constructor.

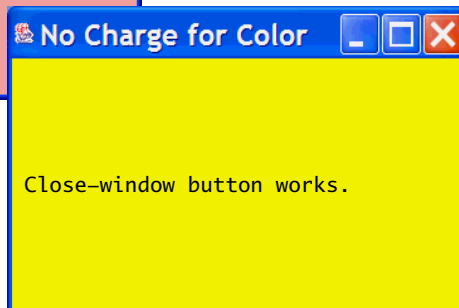
This is the file ColoredWindow.java.

Display 16.6 A JFrame with Color (Part 2 of 2)

```
1 import java.awt.Color;           This is the file DemoColoredWindow.java.
2 public class DemoColoredWindow
3 {
4     public static void main(String[] args)
5     {
6         ColoredWindow w1 = new ColoredWindow();
7         w1.setVisible(true);
8
9         ColoredWindow w2 = new ColoredWindow(Color.YELLOW);
10        w2.setVisible(true);
11    }
```

RESULTING GUI

You will need to use your mouse to drag the top window or you will not see the bottom window.



This gives us a name, `contentPane`, for the content pane of the `JFrame` windowing GUI we are defining. Thus, an invocation of the method `add` can be written in the simpler form

```
contentPane.add(aLabel);
```

instead of the slightly more complex (and slightly less efficient) expression

```
getContentPane().add(aLabel);
```

Container

The important thing to note here is that the method `getContentPane` produces an object of type `Container`. We will say a bit more about the class `Container` later in this chapter. For now, all you need to know about this class is that it is the type to use for the object returned by the method `getContentPane` (that is, for the content pane of the `JFrame`).

The GUI class `ColoredWindow` in Display 16.6 programs the close-window button as follows:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

This way, when the user clicks the close-window button, the program ends. Note that if the program has more than one window, as it does in Display 16.6, and the user clicks the close-window button in any one window of the class `ColoredWindow`, then the entire program ends and all windows go away.

Note that we set the title of the `JFrame` by making it an argument to `super` rather than an argument to `setTitle`. This is another common way to set the title of a `JFrame`.

If you run the program `DemoColoredWindow` in Display 16.6, then the two windows will be placed one on top of the other. To see both windows, you need to use your mouse to move the top window.

EXIT_ON_CLOSE

SETTING THE TITLE OF A JFrame

The two most common ways to set the title of a `JFrame` are to use the method `setTitle`, as illustrated in Display 16.4, or to give the title as an argument to the base class constructor `super`, as illustrated in Display 16.6. (This assumes the base class is `JFrame`, as it always is in this chapter, or a descendent of the class `JFrame` with a suitable constructor.)

Self-Test Exercises

12. How would you modify the class definition in Display 16.6 so that the window produced by the no-argument constructor is magenta instead of blue?
13. Suppose `myFrame` is an object of the class `JFrame`. Give a Java statement that will declare a variable named `contentPane`, and set it so that it names the content pane of `myFrame`.
14. What `import` statement do you need to be able to use the type `Container`?
15. Rewrite the following two lines from Display 16.6 so that the label does not have the name `aLabel` or any other name. Hint: Use an anonymous object.

```
JLabel aLabel = new JLabel("Close-window button works.");
contentPane.add(aLabel);
```


16.3

Containers and Layout Managers

Don't put all your eggs in one basket.

Proverb

There are two main ways to create new classes from old classes. One way is to use inheritance; this is known as the *Is-A relationship*. For example, an object of the class `ColoredWindow` in Display 16.6 *is a* `JFrame` because `ColoredWindow` is a derived class of the class `JFrame`. The second way to create a new class from an existing class (or classes) is to have instance variables of an already existing class type; this is known as *composition* or the *Has-A relationship*. The Swing library has already set things up so you can easily use composition. The actual code for declaring instance variables is in the Swing library classes, such as the class `JFrame`, and you program by adding components to a `JFrame` or objects of certain other classes using the `add` method, which does ultimately set some instance variable. In this section we discuss adding and arranging components in a GUI or subpart of a GUI.

Thus far, we have only added one component, either a button or a label, to the content pane of a `JFrame`. You can add more than one element to a content pane, but the `add` method simply tells which components are added to the content pane; it does not say how they are arranged, such as side by side or one above the other. To describe how the components are arranged, you need to use a [layout manager](#).

layout manager

In this section we will see that there are other classes of objects besides the content pane of a `JFrame` that can have components added with the `add` method and arranged by a layout manager. All these classes are known as [container classes](#).

container class

BORDER LAYOUT MANAGERS

Display 16.7 contains an example of a GUI that uses a layout manager to arrange three labels in a `JFrame`. The labels are arranged one below the other on three lines.

A layout manager is added to the content pane of the `JFrame` class in Display 16.7 with the following line:

```
contentPane.setLayout(new BorderLayout());
```

setLayout

`BorderLayout` is a layout manager class, so `new BorderLayout()` produces a new anonymous object of the class `BorderLayout`. This `BorderLayout` object is given the task of arranging components (in this case, labels) that are added to the content pane.

BorderLayout

It may help to note that the above invocation of `setLayout` is equivalent to the following:

```
BorderLayout manager = new BorderLayout();  
contentPane.setLayout(manager);
```

Display 16.7 The BorderLayout Manager (Part 1 of 2)

```

1  import javax.swing.JFrame;
2  import javax.swing.JLabel;
3  import java.awt.Container;
4  import java.awt.BorderLayout;

5  public class BorderLayoutJFrame extends JFrame
6  {
7      public static final int WIDTH = 500;
8      public static final int HEIGHT = 400;

9      public BorderLayoutJFrame()
10     {
11         super("BorderLayout Demonstration");
12         setSize(WIDTH, HEIGHT);
13         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

14         Container contentPane = getContentPane();
15         contentPane.setLayout(new BorderLayout());

16         JLabel label1 = new JLabel("First label");
17         contentPane.add(label1, BorderLayout.NORTH);

18         JLabel label2 = new JLabel("Second label");
19         contentPane.add(label2, BorderLayout.SOUTH);

20         JLabel label3 = new JLabel("Third label");
21         contentPane.add(label3, BorderLayout.CENTER);
22     }
23 }

```

This is the file BorderLayoutJFrame.java.

This is the file BorderLayoutDemo.java.

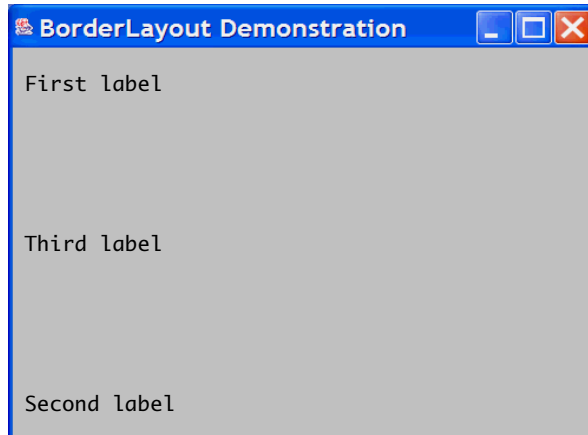
```

1  public class BorderLayoutDemo
2  {
3      public static void main(String[] args)
4      {
5          BorderLayoutJFrame gui = new BorderLayoutJFrame();
6          gui.setVisible(true);
7      }
8  }

```

Display 16.7 The BorderLayout Manager (Part 2 of 2)

RESULTING GUI



Note that the method `setLayout` is invoked not by the `JFrame` itself, but by the content pane of the `JFrame`, which in this case is named `contentPane`. This is because we actually add the labels to the content pane and not (directly) to the `JFrame`. You should invoke `setLayout` with the same object that you use to invoke `add` (and so far that has always been the content pane of a `JFrame`).

A `BorderLayout` manager places labels (or other components) into the five regions `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLayout.WEST`, and `BorderLayout.CENTER`. These five regions are arranged as shown in Display 16.8. The outside box represents the content pane (or other container to which you will add things). None of the lines in the diagram will be visible unless you do something to make them visible. We drew them in to show you where each region is located.

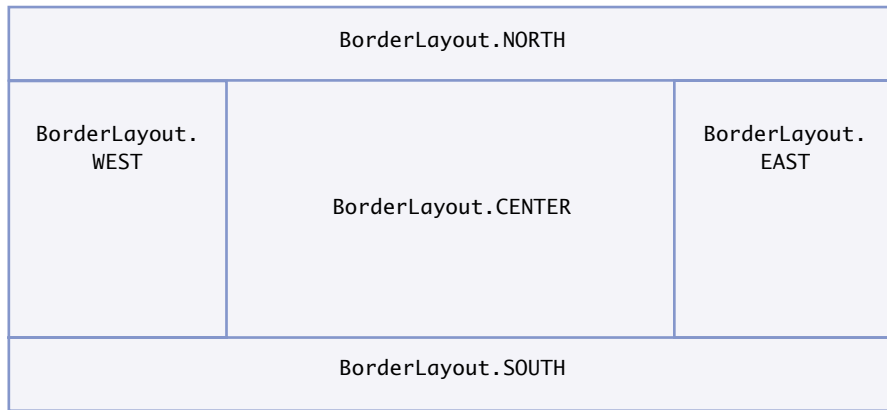
In Display 16.7, we added labels as follows:

```
JLabel label1 = new JLabel("First label");
contentPane.add(label1, BorderLayout.NORTH);

JLabel label2 = new JLabel("Second label");
contentPane.add(label2, BorderLayout.SOUTH);

JLabel label3 = new JLabel("Third label");
contentPane.add(label3, BorderLayout.CENTER);
```

Display 16.8 BorderLayout Regions



When you use a `BorderLayout` manager, you give the location of the component added as a second argument to the method `add`, as in the following:

```
content.add(label1, BorderLayout.NORTH);
```

The labels (or other components to be added) need not be added in any particular order, because the second argument completely specifies where the label is placed.

`BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLayout.WEST`, and `BorderLayout.CENTER` are five string constants defined in the class `BorderLayout`. The values of these constants are "North", "South", "East", "West", and "Center". Although you can use a quoted string such as "North" as the second argument to `add`, it is more consistent with our general style rules to use a defined constant like `BorderLayout.NORTH`.

You need not use all five regions. For example, in Display 16.7 we did not use the regions `BorderLayout.EAST` and `BorderLayout.WEST`. If some regions are not used, any extra space is given to the `BorderLayout.CENTER` region, which is the largest region.

(The space is divided between regions as follows: Regions are allocated space in the order first north and south, second east and west, and last center. So, in particular, if there is nothing in the north region, then the east and west regions will extend to the top of the space.)

From this discussion, it sounds as though you can place only one item in each region, but later in this chapter, when we discuss *panels*, you will see that there is a way to group items so that more than one item can (in effect) be placed in each region.

There are some standard layout managers defined for you in the `java.awt` package, and you can also define your own layout managers. However, for most purposes, the layout managers defined in the standard libraries are all that you need, and we will not discuss how you can create your own layout manager classes.

LAYOUT MANAGERS

The components that you add to a container class are arranged by an object known as a **layout manager**. You add a layout manager with the method `setLayout`, which is a method of every container class, such as the content pane of a `JFrame` or an object of any of the other container classes that we will introduce later in this chapter. If you do not add a layout manager, a default layout manager will be provided for you.

SYNTAX:

```
Container_Object.setLayout(new Layout_Manager_Class());
```

EXAMPLE (WITHIN A CONSTRUCTOR FOR A CLASS CALLED BorderLayoutJFrame):

```
public BorderLayoutJFrame()
{
    ...
    Container contentPane = getContentPane();
    contentPane.setLayout(new BorderLayout());

    JLabel label1 = new JLabel("First label");
    contentPane.add(label1, BorderLayout.NORTH);

    JLabel label2 = new JLabel("Second label");
    contentPane.add(label2, BorderLayout.SOUTH);

    ...
}
```

■ FLOW LAYOUT MANAGERS

The `FlowLayout` **manager** is the simplest layout manager. It arranges components one after the other, going from left to right, in the order in which you add them to the class using the method `add`. For example, if the class in Display 16.7 had used the `FlowLayout` manager instead of the `BorderLayout` manager, it would have used the following code:

```
contentPane.setLayout(new FlowLayout());

JLabel label1 = new JLabel("First label");
contentPane.add(label1);

JLabel label2 = new JLabel("Second label");
contentPane.add(label2);

JLabel label3 = new JLabel("Third label");
contentPane.add(label3);
```

Note that if we had used the `FlowLayout` manager, as in the preceding code, then the `add` method would have only one argument. With a `FlowLayout` manager, the items are displayed in the order they are added, so that labels above would be displayed all on one line as follows:

```
First label Second label Third label
```

extra code on CD

The full program is in the files `FlowLayoutJFrame.java` and `FlowLayoutDemo.java` on the accompanying CD. You will see a number of examples of GUIs that use the `FlowLayout` manager class later in this chapter.

■ GRID LAYOUT MANAGERS

GridLayout

A **GridLayout manager** arranges components in a two-dimensional grid with some number of rows and columns. With a `GridLayout` manager, each entry is the same size. For example, the following says to use a `GridLayout` manager with `aContainer`, which can be a content pane or other container:

```
aContainer.setLayout(new GridLayout(2, 3));
```

The two numbers given as arguments to the constructor `GridLayout` specify the number of rows and columns. This would produce the following sort of layout:

The lines will not be visible unless you do something special to make them visible. They are just included here to show you the region boundaries.

When using a `GridLayout` manager, each component is stretched so that it completely fills its grid position.

Although you specify a number of rows and columns, the rules for the number of rows and columns is more complicated than what we have said so far. If the values for the number of rows and the number of columns are both nonzero, then the number of columns will be ignored. For example, if the specification is `new GridLayout(2, 3)`, then some sample sizes are as follows: If you add six items, the grid will be as shown. If you add seven or eight items, a fourth column is automatically added, and so forth. If you add fewer than six components, there will be two rows and a reduced number of columns.

You can specify that the number of columns is to be ignored by setting the number of columns to zero, which will allow any number of columns. So a specification of `(2, 0)` is equivalent to `(2, 3)`, and in fact is equivalent to `(2, n)` for any nonnegative value of `n`. Similarly, you can specify that the number of rows is to be ignored by setting the number of rows to zero, which will allow any number of rows.

When using the `GridLayout` class, the method `add` has only one argument. The items are placed in the grid from left to right, first filling the top row, then the second row, and so forth. You are not allowed to skip any grid position (although you will later see that you can add something that does not show and so gives the illusion of skipping a grid position).

A sample use of the `GridLayout` class is given in Display 16.9.

Note that we have placed a demonstration `main` method in the class definition in Display 16.9. Normally, a Swing GUI class is used to create and display a GUI in a `main` method (or other method) in some class other than the class for the Swing GUI. However, it is perfectly legal and sometimes convenient to place a `main` method in the GUI class definition so that it is easy to display a sample of the GUI. Note that the `main` method that is given in the class itself is written in the same way as a `main` method that is in some other class. In particular, you need to construct an object of the class, as in the following line from the `main` method in Display 16.9:

```
GridLayoutJFrame gui = new GridLayoutJFrame(2, 3);
```

The three layout managers we have discussed are summarized in Display 6.10.

Next we will discuss *panels*, which will let you realize the full potential of layout managers.

Display 16.9 The GridLayout Manager (Part 1 of 2)

```
1 import javax.swing.JFrame;
2 import javax.swing.JLabel;
3 import java.awt.Container;
4 import java.awt.GridLayout;

5 public class GridLayoutJFrame extends JFrame
6 {
7     public static final int WIDTH = 500;
8     public static final int HEIGHT = 400;

9     public static void main(String[] args)
10    {
11        GridLayoutJFrame gui = new GridLayoutJFrame(2, 3);
12        gui.setVisible(true);
13    }

14    public GridLayoutJFrame(int rows, int columns )
15    {
16        super();
17        setSize(WIDTH, HEIGHT);
18        setTitle("GridLayout Demonstration");
19        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Display 16.9 The GridLayout Manager (Part 2 of 2)

```
20     Container contentPane = getContentPane();
21     contentPane.setLayout(new GridLayout(rows, columns ));

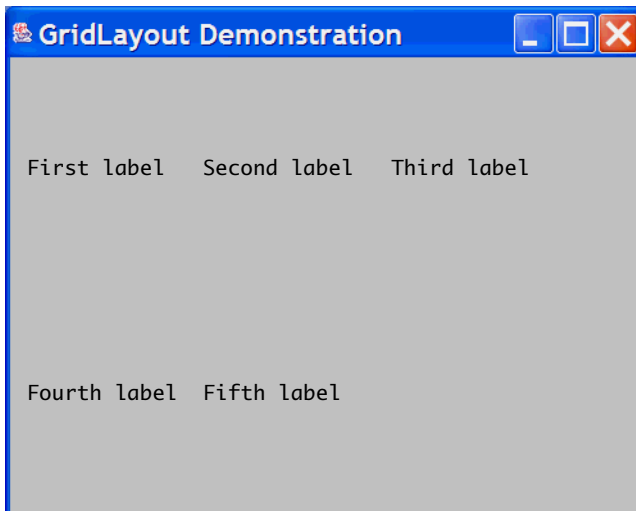
22     JLabel label1 = new JLabel("First label");
23     contentPane.add(label1);

24     JLabel label2 = new JLabel("Second label");
25     contentPane.add(label2);

26     JLabel label3 = new JLabel("Third label");
27     contentPane.add(label3);

28     JLabel label4 = new JLabel("Fourth label");
29     contentPane.add(label4);

30     JLabel label5 = new JLabel("Fifth label");
31     contentPane.add(label5);
32 }
33 }
```

RESULTING GUI

Display 16.10 Some Layout Managers

LAYOUT MANAGER	DESCRIPTION
These layout manager classes are in the <code>java.awt</code> package.	
FlowLayout	Displays components from left to right in the order in which they are added to the container.
BorderLayout	Displays the components in five areas: north, south, east, west, and center. You specify the area a component goes into in a second argument of the <code>add</code> method.
GridLayout	Lays out components in a grid, with each component stretched to fill its box in the grid.

Self-Test Exercises

16. In Display 16.7, would it be legal to replace

```
JLabel label1 = new JLabel("First label");  
contentPane.add(label1, BorderLayout.NORTH);
```

```
JLabel label2 = new JLabel("Second label");  
contentPane.add(label2, BorderLayout.SOUTH);
```

```
JLabel label3 = new JLabel("Third label");  
contentPane.add(label3, BorderLayout.CENTER);
```

with the following?

```
JLabel aLabel = new JLabel("First label");  
contentPane.add(aLabel, BorderLayout.NORTH);
```

```
aLabel = new JLabel("Second label");  
contentPane.add(aLabel, BorderLayout.SOUTH);
```

```
aLabel = new JLabel("Third label");  
contentPane.add(aLabel, BorderLayout.CENTER);
```

In other words, can we reuse the variable `aLabel` or must each label have its own variable name?

17. How would you modify the class definition in Display 16.7 so that the three labels are displayed as follows?

```
First label
Second label
Third label
```

(There may be space between each pair of lines.)

18. How would you modify the class definition in Display 16.7 so that the three labels are displayed as follows?

```
First label
                                     Second label
Third label
```

(There may be space between each pair of lines.)

19. Suppose you are defining a windowing GUI class in the usual way, as a derived class of the class `JFrame`, and suppose that the constructor obtains the content pane as follows:

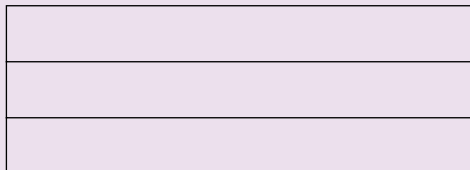
```
Container contentPane = getContentPane();
```

Now suppose you want to specify a layout manager for `contentPane` so as to produce the following sort of layout (that is, a one-row layout, typically having three columns):



What should the argument to `contentPane.setLayout` be?

20. Suppose the situation is as described in exercise 19, except that you want the following sort of layout (that is, a one-column layout, typically having three rows):



What should the argument to `setLayout` be?

PANELS

A GUI is often organized in a hierarchical fashion, with window-like containers, known as *panels*, inside of other window-like containers. A **panel** is an object of the class `JPanel`, which is a very simple container class that does little more than group objects. It is one of the simplest container classes, but an extremely useful one. A `JPanel` object is analogous to the braces used to combine a number of simpler Java statements into a single larger Java statement. It groups smaller objects, such as buttons and labels, into a larger component (the `JPanel`). You can then put the `JPanel` object in the content pane of a `JFrame`. Thus, one of the main functions of `JPanel` objects is to subdivide a `JFrame` (or other container) into different areas.

panel

For example, when you use a `BorderLayout` manager, you can place components in each of the five locations `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLayout.WEST`, and `BorderLayout.CENTER`. But what if you want to put two components at the bottom of the screen in the `BorderLayout.SOUTH` position? To do this, you would put the two components in a panel and then place the panel in the `BorderLayout.SOUTH` position.

You can give different layout managers to the content pane of a `JFrame` and to each panel in the `JFrame`. Since you can add panels to other panels and each panel can have its own layout manager, this enables you to produce almost any kind of overall layout of the items in your GUI.

For example, if you want to place two buttons at the bottom of your `JFrame` GUI, you might add the following to the constructor of your `JFrame` GUI:

```
Container contentPane = getContentPane();
contentPane.setLayout(new BorderLayout());

JPanel buttonPanel = new JPanel();
buttonPanel.setLayout(new FlowLayout());

JButton firstButton = new JButton("One");
buttonPanel.add(firstButton);

JButton secondButton = new JButton("Two");
buttonPanel.add(secondButton);

contentPane.add(buttonPanel, BorderLayout.SOUTH);
```

The next Programming Example makes use of panels within panels.

Example

A TRICOLOR BUILT WITH PANELS

When first run, the GUI defined in `Display 16.11` looks as shown in the first view. The entire background is light gray and there are three buttons at the bottom of the GUI labeled "Red",

"White", and "Blue". If you click any one of the buttons, a vertical stripe with the color written on the button appears. You can click the buttons in any order. In the last three views in Display 16.11 we show what happens if you click the buttons in left-to-right order.

The red, white, and blue stripes are the `JPanels` named `redPanel`, `whitePanel`, and `bluePanel`. At first the panels are not visible because they are all light gray, so no borders are visible. When you click a button, the corresponding panel changes color and so is clearly visible.

Be sure to notice that you add things to a `JFrame` and to a `JPanel` in slightly different ways. With a `JFrame`, you first get the content pane with `getContentPane`, and then you use the method `add` with the content pane. With a `JPanel`, you use the method `add` directly with the `JPanel` object. There is no content pane to worry about with a `JPanel`.

Notice how the action listeners are set up. Each button registers the `this` parameter as a listener, as in the following line:

```
redButton.addActionListener(this);
```

Because this line appears inside of the constructor for the class `PanelDemo`, the `this` parameter refers to `PanelDemo`, which is the entire GUI. Thus, the entire `JFrame` (the entire GUI) is the listener, not the `JPanel`. So when you click one of the buttons, it is the `actionPerformed` method in `PanelDemo` that is executed.

When a button is clicked, the `actionPerformed` method is invoked with the action event fired as the argument to `actionPerformed`. The method `actionPerformed` recovers the string written on the button with the following line:

```
String buttonString = e.getActionCommand();
```

The method `actionPerformed` then uses a multiway `if-else` statement to determine if `buttonString` is "Red", "White", or "Blue" and changes the color of the corresponding panel accordingly. It is common for an `actionPerformed` method to be based on such a multiway `if-else` statement, although we will see another approach in the subsection entitled "Listeners As Inner Classes" later in this chapter.

Display 16.11 also introduces one other small, but new technique. In addition to giving colors to the panels and the content pane, we also gave each button a color. We did this with the method `setBackground`, using basically the same technique that we used in previous examples. You can give a button or almost any other item a color using `setBackground`.

■ THE Container CLASS

Container

The class called `Container` is in the `java.awt` package. Any descendent class of the class `Container` can have components added to it (or, more precisely, can have components added to objects of the class). The class `JFrame` is a descendent class of the class `Container`, so any descendent class of the class `JFrame` can serve as a container to hold labels, buttons, panels, or other components.

Display 16.11 Using Panels (Part 1 of 3)

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import java.awt.Container;
4  import java.awt.BorderLayout;
5  import java.awt.GridLayout;
6  import java.awt.FlowLayout;
7  import java.awt.Color;
8  import javax.swing.JButton;
9  import java.awt.event.ActionListener;
10 import java.awt.event.ActionEvent;

11 public class PanelDemo extends JFrame implements ActionListener
12 {
13     public static final int WIDTH = 300;
14     public static final int HEIGHT = 200;

15     private JPanel redPanel;
16     private JPanel whitePanel;
17     private JPanel bluePanel;

18     public static void main(String[] args)
19     {
20         PanelDemo gui = new PanelDemo();
21         gui.setVisible(true);
22     }


23     public PanelDemo()
24     {
25         super("Panel Demonstration");
26         setSize(WIDTH, HEIGHT);
27         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28         Container contentPane = getContentPane();
29         contentPane.setLayout(new BorderLayout());

30         JPanel biggerPanel = new JPanel();
31         biggerPanel.setLayout(new GridLayout(1, 3));

32         redPanel = new JPanel();
33         redPanel.setBackground(Color.LIGHT_GRAY);
34         biggerPanel.add(redPanel);

35         whitePanel = new JPanel();
36         whitePanel.setBackground(Color.LIGHT_GRAY);
37         biggerPanel.add(whitePanel);
```

In addition to being the GUI class, the class `PanelDemo` is also the action listener class. An object of the class `PanelDemo` is the action listener for the buttons in that object.



We made these panels instance variables because we want to refer to them in both the constructor and the method `actionPerformed`.



Display 16.11 Using Panels (Part 2 of 3)

```

38     bluePanel = new JPanel();
39     bluePanel.setBackground(Color.LIGHT_GRAY);
40     biggerPanel.add(bluePanel);

41     contentPane.add(biggerPanel, BorderLayout.CENTER);

42     JPanel buttonPanel = new JPanel();
43     buttonPanel.setBackground(Color.LIGHT_GRAY);
44     buttonPanel.setLayout(new FlowLayout());

45     JButton redButton = new JButton("Red");
46     redButton.setBackground(Color.RED);
47     redButton.addActionListener(this);
48     buttonPanel.add(redButton);

49     JButton whiteButton = new JButton("White");
50     whiteButton.setBackground(Color.WHITE);
51     whiteButton.addActionListener(this);
52     buttonPanel.add(whiteButton);

53     JButton blueButton = new JButton("Blue");
54     blueButton.setBackground(Color.BLUE);
55     blueButton.addActionListener(this);
56     buttonPanel.add(blueButton);

57     contentPane.add(buttonPanel, BorderLayout.SOUTH);
58 }

59 public void actionPerformed(ActionEvent e)
60 {
61     String buttonString = e.getActionCommand();

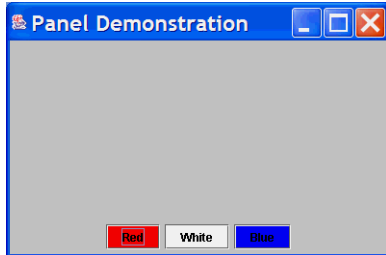
62     if (buttonString.equals("Red"))
63         redPanel.setBackground(Color.RED);
64     else if (buttonString.equals("White"))
65         whitePanel.setBackground(Color.WHITE);
66     else if (buttonString.equals("Blue"))
67         bluePanel.setBackground(Color.BLUE);
68     else
69         System.out.println("Unexpected error.");
70 }
71 }

```

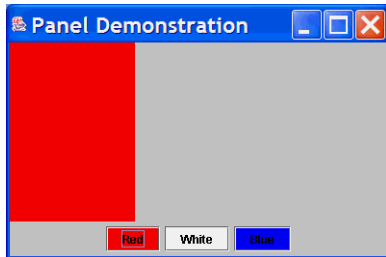
An object of the class `PanelDemo` is the action listener for the buttons in that object.

Display 16.11 Using Panels (Part 3 of 3)

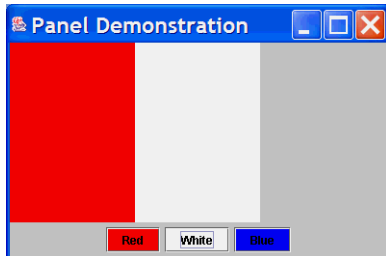
RESULTING GUI (When first run)



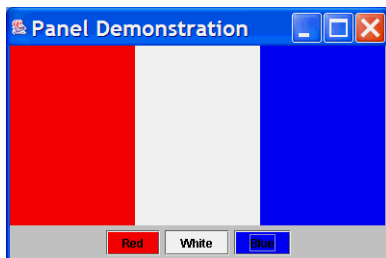
RESULTING GUI (After clicking Red button)



RESULTING GUI (After clicking White button)



RESULTING GUI (After clicking Blue button)



Similarly, the class `JPanel` is a descendent of the class `Container`, and any object of the class `JPanel` can serve as a container to hold labels, buttons, other panels, or other components. Display 16.12 shows a portion of the hierarchy of Swing and AWT classes. Note that the `Container` class is in the AWT library and not in the Swing library. This is not a major issue, but it does mean that the `import` statement for the `Container` class is

```
import java.awt.Container;
```

A **container class** is any descendent class of the class `Container`. The class `JComponent` serves a similar roll for components. Any descendent class of the class `JComponent` is called a `JComponent` or sometimes simply a **component**. You can add any `JComponent` object to any container class object.

container class
component

The class `JComponent` is derived from the class `Container`, so you can add a `JComponent` to another `JComponent`. Often, this will turn out to be a viable option; occasionally it is something to avoid.⁵

The classes `Component`, `Frame`, and `Window` shown in Display 16.12 are AWT classes that some readers may have heard of. We include them for reference value, but we will have no need for these classes. We will eventually discuss all the other classes shown in Display 16.12.

When you are dealing with a Swing container class, you have three kinds of objects to deal with:

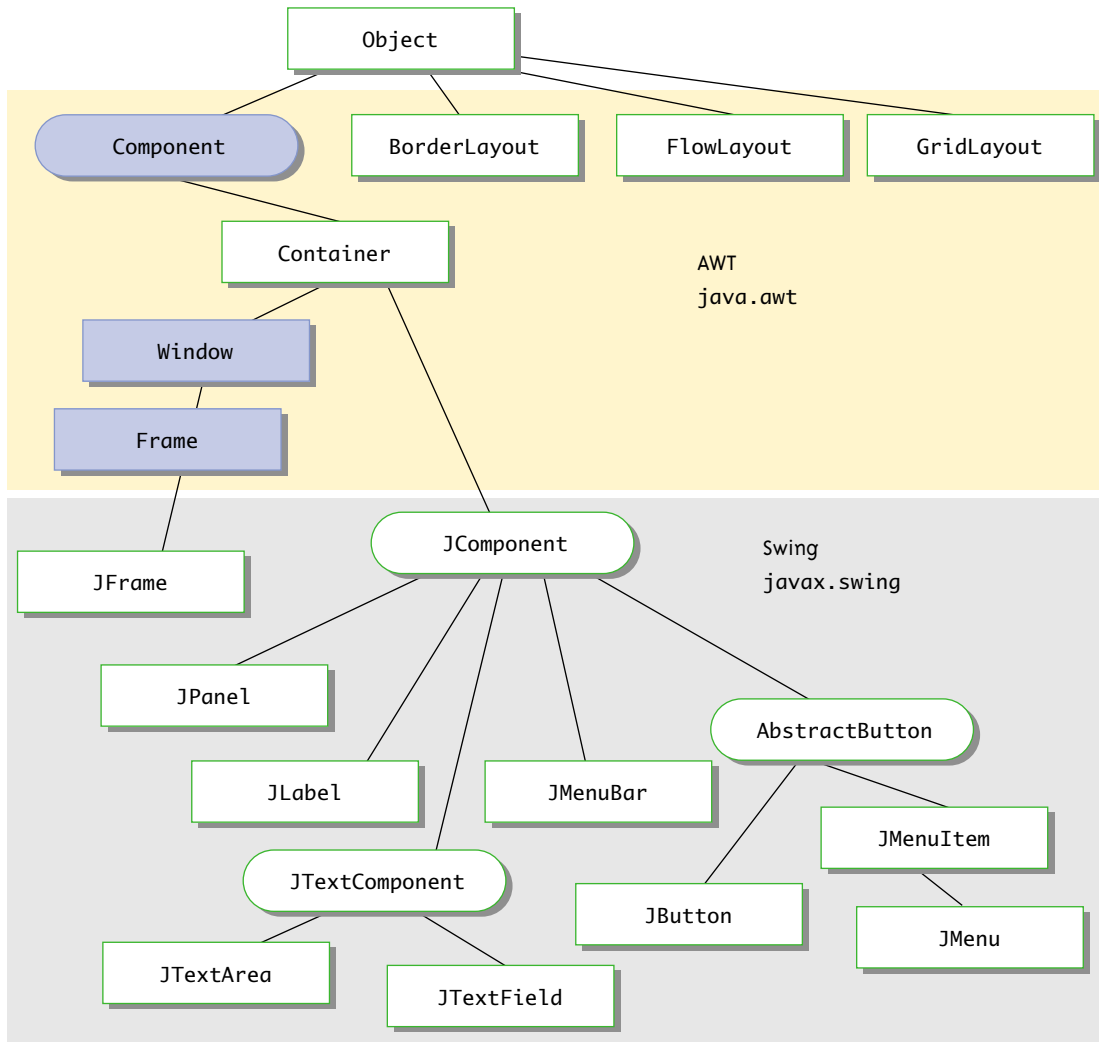
1. The container itself, probably some sort of panel or window-like object
2. The components you add to the container, like labels, buttons, and panels
3. A layout manager, which positions the components inside the container

You have seen examples of these three kinds of objects in almost every `JFrame` class we have defined. Almost every complete GUI you build, and many subparts of the GUIs you build, will be made up of these three kinds of objects.

The `JPanel` class inherits the method `add` from the class `Container`, and the method `add` can be used to directly add a component to a `JPanel`. `JFrame` is a different kind of container class. The class `JFrame` is a descendent class of the class `Container` and so it inherits the `add` method from the class `Container`. However, if you try to use the method `add` directly on a `JFrame` (instead of on the content pane of the `JFrame`), you will get a run-time error message. The content pane of a `JFrame` is also used for other things that we will not discuss in this book. For what we are doing in this book, the content pane of a `JFrame` is just an unavoidable nuisance.

⁵ In particular, it is legitimate and sometimes useful to add `JComponents` to a `JButton`. We do not have space in this book to develop techniques for doing this effectively, but you may want to give it a try. You have had enough material to do it.

Display 16.12 Hierarchy of Swing and AWT Classes



Abstract Class

Concrete Class

A line between two boxes means the lower class is derived from (extends) the higher one.

This blue color indicates a class that is not used in this text but is included here for reference. If you have not heard of any of these classes, you can safely ignore them. (The class Component does receive very brief treatment in Chapter 18.)

WHEN TO USE A CONTENT PANE

You add items to a `JFrame` by using a combination of `getContentPane` and `add`, as in the following:

```
Container contentPane = getContentPane();
...
JButton aButton = new JButton("Click me");
...
contentPane.add(aButton);
```

However, with a `JPanel`, you do not use `getContentPane`, but instead use the method `add` directly with the `JPanel` object, as illustrated below:

```
JPanel buttonPanel = new JPanel();
...
JButton aButton = new JButton("Click me");
...
buttonPanel.add(aButton);
```

With `JFrame` objects, you need to use the method `getContentPane` to get the content pane of the `JFrame` object and then use the method `add` with the content pane as the calling object. With `JApplets`, which are discussed in Chapter 17, you need to add things to a content pane in a similar way. With all other container classes discussed in this text, you use the method `add` directly with an object of the container class and do not use a content pane.

WHY DOES A JFrame HAVE A CONTENT PANE?

Unfortunately, the question does not have an easy answer. It has to do with ways of using a `JFrame` object that we will not go into in this book. If it seems to you that there is no intrinsic need for a content pane, take comfort in the fact that your observation is well taken. For what we are doing in this book, the `JFrame` class could have been defined so that it does not have a content pane. In fact, the precursor class of the class `JFrame` (in the older AWT library of classes) did not have a content pane. However, a `JFrame` object does have a content pane and you must deal with the content pane or your programs will not work correctly.

Self-Test Exercises

21. When adding components to a `JFrame`, do you need to use `getContentPane`? When adding components to a `JPanel`, do you need to use `getContentPane`?
22. What standard Java package contains the layout manager classes discussed in this chapter?
23. Is an object of the class `JPanel` a container class? Is it a component class?
24. With a `GridLayout` manager, you cannot leave any grid element empty, but you can do something that will make a grid element look empty to the user. What can you do?

25. You are used to defining derived classes of the Swing class `JFrame`. You can also define derived classes of other Swing classes. Define a derived class of the class `JPanel` that is called `PinkJPanel`. An object of the class `PinkJPanel` can be used just as we used objects of the class `JPanel`, but an object of the class `PinkJPanel` is pink in color (unless you explicitly change its color). The class `PinkJPanel` will have only one constructor, namely the no-argument constructor. (Hint: This is very easy.)

Tip

CODE A GUI'S LOOK AND ACTIONS SEPARATELY

You can divide the task of designing a Swing GUI into two main subtasks: (1) Designing and coding the appearance of the GUI on the screen; (2) Designing and coding the actions performed in response to button clicks and other user actions. This dividing of one big task into two simpler tasks makes the big task easier and less error prone.

For example, consider the program in Display 16.11. Your first version of this program might use the following definition of the method `actionPerformed`:

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

This version of the method `actionPerformed` does nothing, but your program will run and will display a window on the screen, just as shown in Display 16.11. If you click any of the buttons, nothing will happen, but you can use this version of your GUI to adjust details, such as the order and location of buttons.

After you get the GUI to look the way you want it to look, you can define the action parts of the GUI, typically the method `actionPerformed`.

If you include the phrase `implements ActionListener` at the start of your `JFrame` definition, then you must include some definition of the method `actionPerformed`. A method definition, such as

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

which does nothing (or does very little) is called a **stub**. Using stubs is a good programming technique in many contexts, not just in Swing programs.

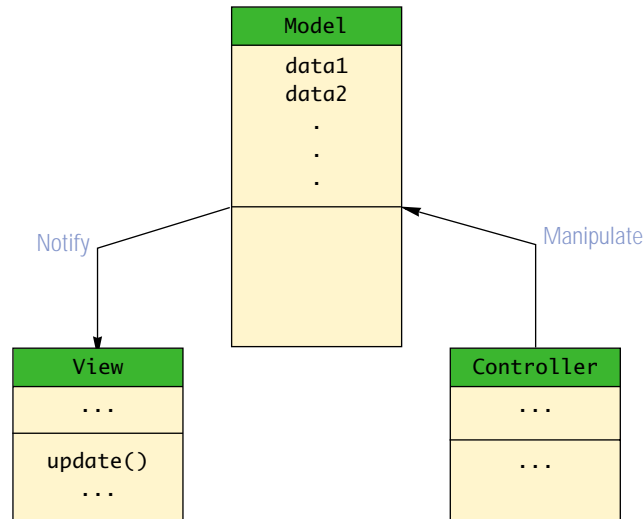
Alternatively, when writing your first version of a Swing GUI like the one in Display 16.11, you could omit the definition of the method `actionPerformed` completely, *provided you also omit the phrase `implements ActionListener` and omit the invocations of `addActionListener`.*

stub

THE MODEL-VIEW-CONTROLLER PATTERN ❖

The technique we advocated in the previous Programming Tip is an example of a general technique known as the **Model-View-Controller** pattern. Display 16.13 gives a diagram of

Model-View-
Controller

Display 16.13 The Model-View-Controller Pattern

this pattern. The Model part of the pattern performs the heart of the application. The View part is the output part; it displays a picture of the Model's state. The Controller is the input part; it relays commands from the user to the Model. Each of the three interacting parts is realized as an object with responsibility for its own tasks. In a simple task such as the `JFrame` in Display 16.11, you can have a single object with different methods to realize each of the roles Model, View, and Controller.

To simplify the discussion, we have presented the Model-View-Controller pattern as if the user interacts directly with the Controller. The Controller need not be under the direct control of the user, but could be controlled by some other software or hardware component. In a Swing GUI, the View and Controller parts might be separate classes or separate methods combined into one larger class that displays a single window for all user interactions.

Self-Test Exercises

26. Suppose you omitted the method `actionPerformed` from the class in Display 16.11 and made no other changes. Would the class compile? If it compiles, will it run with no error messages?
27. Suppose you omitted the method `actionPerformed` and the phrase `implements ActionListener` from the class in Display 16.11 and made no other changes. Would the class compile? If it compiles, will it run with no error messages?

16.4

Menus and Buttons

For hours and location press 1.

For a recorded message describing services press 2.

For instructions on using our website press 3.

To use our automated information system press 4.

To speak to an operator between 8 am and noon Monday through Thursdays press 7.

Phone answering machine

In this section we describe the basics of Swing menus. Swing menu items (menu choices) behave essentially the same as Swing buttons. They generate action events that are handled by action listeners, just as buttons do.

Example

A GUI WITH A MENU

Display 16.14 contains a program that is essentially the same as the GUI in Display 16.11 except that this GUI uses a menu instead of buttons. This GUI has a menu bar at the top of the window. The menu bar lists the names of all the pull-down menus. This GUI has only one pull-down menu, which is named "Add Colors". However, there could be more pull-down menus in the same menu bar.

The user can pull down a menu by clicking its name in the menu bar. Display 16.14 contains three pictures of the GUI. The first is what you see when the GUI first appears. In that picture, the menu name "Add Colors" can be seen in the menu bar, but you cannot see the menu. If you click the words "Add Colors" with your mouse, the menu drops down, as shown in the second picture of the GUI. If you click "Red", "White", or "Blue" on the menu, then a vertical strip of the named color appears in the GUI.

In the next subsection, we go over the details of the program in Display 16.14.

MENU BARS, MENUS, AND MENU ITEMS

When adding menus as we did in Display 16.14, you use the three Swing classes `JMenuItem`, `JMenu`, and `JMenuBar`. Entries on a menu are objects of the class `JMenuItem`. These `JMenuItems` are placed in `JMenus`, and then the `JMenus` are typically placed in a `JMenuBar`. Let's look at the details.

A **menu** is an object of the class `JMenu`. A choice on a menu is called a **menu item** and is an object of the class `JMenuItem`. A menu item is identified by the string that labels it, such as "Red", "White", or "Blue" in the menu in Display 16.14. You can add as many `JMenuItems` as you wish to a menu. The menu lists the items in the order in

menu
menu item

Display 16.14. A GUI with a Menu (Part 1 of 3)

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import java.awt.Container;
4  import java.awt.GridLayout;
5  import java.awt.Color;
6  import javax.swing.JMenu;
7  import javax.swing.JMenuItem;
8  import javax.swing.JMenuBar;
9  import java.awt.event.ActionListener;
10 import java.awt.event.ActionEvent;

11 public class MenuDemo extends JFrame implements ActionListener
12 {
13     public static final int WIDTH = 300;
14     public static final int HEIGHT = 200;

15     private JPanel redPanel;
16     private JPanel whitePanel;
17     private JPanel bluePanel;

18     public static void main(String[] args)
19     {
20         MenuDemo gui = new MenuDemo();
21         gui.setVisible(true);
22     }

23     public MenuDemo()
24     {
25         super("Menu Demonstration");
26         setSize(WIDTH, HEIGHT);
27         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28         Container contentPane = getContentPane();
29         contentPane.setLayout(new GridLayout(1, 3));

30         redPanel = new JPanel();
31         redPanel.setBackground(Color.LIGHT_GRAY);
32         contentPane.add(redPanel);

33         whitePanel = new JPanel();
34         whitePanel.setBackground(Color.LIGHT_GRAY);
35         contentPane.add(whitePanel);
```

Display 16.14 A GUI with a Menu (Part 2 of 3)

```
36     bluePanel = new JPanel();
37     bluePanel.setBackground(Color.LIGHT_GRAY);
38     contentPane.add(bluePanel);

39     JMenu colorMenu = new JMenu("Add Colors");

40     JMenuItem redChoice = new JMenuItem("Red");
41     redChoice.addActionListener(this);
42     colorMenu.add(redChoice);

43     JMenuItem whiteChoice = new JMenuItem("White");
44     whiteChoice.addActionListener(this);
45     colorMenu.add(whiteChoice);

46     JMenuItem blueChoice = new JMenuItem("Blue");
47     blueChoice.addActionListener(this);
48     colorMenu.add(blueChoice);

49     JMenuBar bar = new JMenuBar();
50     bar.add(colorMenu);
51     setJMenuBar(bar);
52 }
```

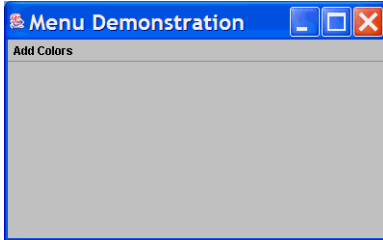
The definition of `actionPerformed` is identical to the definition given in Display 16.11 for a similar GUI using buttons instead of menu items.

```
53     public void actionPerformed(ActionEvent e)
54     {
55         String buttonString = e.getActionCommand();

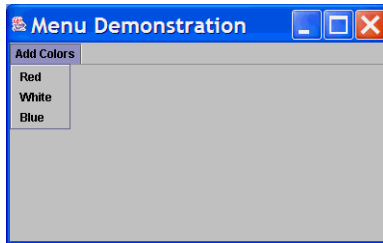
56         if (buttonString.equals("Red"))
57             redPanel.setBackground(Color.RED);
58         else if (buttonString.equals("White"))
59             whitePanel.setBackground(Color.WHITE);
60         else if (buttonString.equals("Blue"))
61             bluePanel.setBackground(Color.BLUE);
62         else
63             System.out.println("Unexpected error.");
64     }
65 }
```

Display 16.14 A GUI with a Menu (Part 3 of 3)

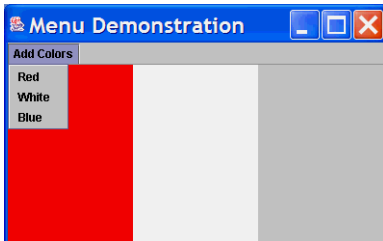
RESULTING GUI



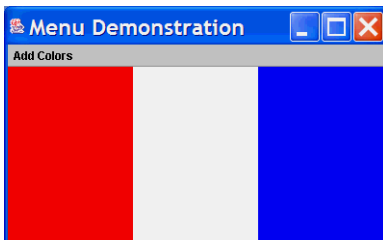
RESULTING GUI (after clicking Add Colors in the menu bar)



RESULTING GUI (after choosing Red and White on the menu)



RESULTING GUI (after choosing all the colors on the menu)



which they are added. The following code, taken from the constructor in Display 16.14, creates a new `JMenu` object named `colorMenu` and then adds a `JMenuItem` labeled "Red". Other menu items are added in a similar way.

```
JMenu colorMenu = new JMenu("Add Colors");

JMenuItem redChoice = new JMenuItem("Red");
redChoice.addActionListener(this);
colorMenu.add(redChoice);
```

Note that, just as we did for buttons in Display 16.11, in Display 16.14 we have registered the `this` parameter as an action listener for each menu item. Defining action listeners and registering listeners for menu items are done in the exact same way as for buttons. In fact, the syntax is even the same. If you compare Display 16.14 and Display 16.11, you will see that the definition of the method `actionPerformed` is the same in both classes.

listeners

You add a `JMenuItem` to an object of the class `JMenu` using the method `add` in exactly the same way that you add a component, such as a button, to a container object. Moreover, if you look at the preceding code, you will see that you specify a string for a `JMenuItem` in the same way that you specify a string to appear on a button.

A **menu bar** is a container for menus, typically placed near the top of a windowing interface. You add a menu to a menu bar using the method `add` in the same way that you add menu items to a menu. The following code from the constructor in Display 16.14 creates a new menu bar named `bar` and then adds the menu named `colorMenu` to this menu bar:

menu bar

```
JMenuBar bar = new JMenuBar();
bar.add(colorMenu);
```

There are two different ways to add a menu bar to a `JFrame`. You can use the method `setJMenuBar`, as shown in the following code from the constructor in Display 16.14:

```
setJMenuBar(bar);
```

This sets an instance variable of type `JMenuBar` so that it names the menu bar named `bar`. Saying it less formally, this adds the menu bar named `bar` to the `JFrame` and places the menu bar at the top of the `JFrame`.

MENUS

A menu is an object of the class `JMenu`. A choice on a menu is an object of the class `JMenuItem`. Menus are collected together in a menu bar (or menu bars). A menu bar is an object of the class `JMenuBar`.

Events and listeners for menu items are handled in exactly the same way as they are for buttons.

Alternatively, you can use the `add` method to add a menu bar to the content pane of a `JFrame` (or to any other container). You do so in the same way that you add any other component, such as a label or a button. An example of using `add` to add a `JMenuBar` to the content pane of a `JFrame` is given in the file `MenuAdd.java` on the accompanying CD.

extra code on CD

ADDING MENUS TO A JFrame

In the following, we assume that all code is inside a constructor for a (derived class of a) `JFrame`. To see the following examples put together to produce a complete GUI, see the constructor in Display 16.14.

CREATING MENU ITEMS

A menu item is an object of the class `JMenuItem`. You create a new menu item in the usual way, as illustrated by the following example. The string in the argument position is the displayed text for the menu item.

```
JMenuItem redChoice = new JMenuItem("Red");
```

ADDING MENU ITEM LISTENERS

Events and listeners for menu items are handled in the exact same way as they are for buttons: Menu items fire action events that are received by objects of the class `ActionListener`.

SYNTAX:

```
JMenu_Item_Name.addActionListener(Action_Listener);
```

EXAMPLE:

```
redChoice.addActionListener(this);
```

CREATING A MENU

A menu is an object of the class `JMenu`. You create a new menu in the usual way, as illustrated by the following example. The string argument is the displayed text that identifies the menu.

```
JMenu colorMenu = new JMenu("Add Colors");
```

ADDING MENU ITEMS TO A MENU

You use the method `add` to add menu items to a menu.

SYNTAX:

```
JMenu_Name.add(JMenuItem);
```

EXAMPLE (colorMenu IS AN OBJECT OF THE CLASS JMenuItem):

```
colorMenu.add(redChoice);
```

CREATING A MENU BAR

A menu bar is an object of the class `JMenuBar`. You create a new menu bar in the usual way, as illustrated by the following example:

```
JMenuBar bar = new JMenuBar();
```

ADDING A MENU TO A MENU BAR

You add a menu to a menu bar using the method `add` as follows:

SYNTAX:

```
JMenu_Bar_Name.add(JMenu_Name);
```

EXAMPLE (bar IS AN OBJECT OF THE CLASS `JMenuBar`):

```
bar.add(colorMenu);
```

ADDING A MENU BAR TO A FRAME

There are two different ways to add a menu bar to a `JFrame`. You can use the method `add` to add the menu bar to the content pane of the `JFrame` (or to any other container). Another common way of adding a menu bar to a `JFrame` is to use the method `setJMenuBar` as follows:

SYNTAX:

```
setJMenuBar(JMenu_Bar_Name);
```

EXAMPLE:

```
setJMenuBar(bar);
```

■ NESTED MENUS ❖

As shown in Display 16.12, the class `JMenu` is a descendent of the `JMenuItem` class. So, every `JMenu` object is also a `JMenuItem` object. Thus, a `JMenu` can be a menu item in another menu. This means that you can nest menus. For example, the outer menu might give you a list of menus. You can display one of the menus on that list by clicking the name of the desired menu. You can then choose an item from that menu by using your mouse again. There is nothing new you need to know to create these nested menus. You simply add menus to menus just as you add other menu items. There is an example of nested menus in the file `NestedMenus.java` on the accompanying CD.

[extra code on CD](#)

■ THE AbstractButton CLASS

As shown in Display 16.12, the classes `JButton` and `JMenuItem` are derived classes of the abstract class named `AbstractButton`. All of the basic properties and methods of the classes `JButton` and `JMenuItem` are inherited from the class `AbstractButton`. That is why objects of the class `JButton` and objects of the class `JMenuItem` are so similar. Some of the methods for the class `AbstractButton` are listed in Display 16.15. All these methods are inherited by both the class `JButton` and the class `JMenuItem`. (Some of these methods were inherited by the class `AbstractButton` from the class `JComponent`, so you may sometimes see some of the methods listed as “inherited from `JComponent`.”)

Display 16.15 Some Methods in the Class `AbstractButton` (Part 1 of 2)

The abstract class `AbstractButton` is in the `javax.swing` package.

All of these methods are inherited by both of the classes `JButton` and `JMenuItem`.

```
public void setBackground(Color theColor)
```

Sets the background color of this component.

```
public void addActionListener(ActionListener listener)
```

Adds an `ActionListener`.

```
public void removeActionListener(ActionListener listener)
```

Removes an `ActionListener`.

```
public void setActionCommand(String actionCommand)
```

Sets the action command.

```
public String getActionCommand()
```

Returns the action command for this component.

```
public void setText(String text)
```

Makes `text` the only text on this component.

```
public String getText()
```

Returns the text written on the component, such as the text on a button or the string for a menu item.

```
public void setPreferredSize(Dimension preferredSize)
```

Sets the preferred size of the button or label. Note that this is only a suggestion to the layout manager. The layout manager is not required to use the preferred size. The following special case will work for most simple situations. The `int` values give the width and height in pixels.

```
public void setPreferredSize(  
    new Dimension(int width, int height))
```

Display 16.15 Some Methods in the Class `AbstractButton` (Part 2 of 2)

```
public void setMaximumSize(Dimension maximumSize)
```

Sets the maximum size of the button or label. Note that this is only a suggestion to the layout manager. The layout manager is not required to respect this maximum size. The following special case will work for most simple situations. The `int` values give the width and height in pixels.

```
public void setMaximumSize(  
    new Dimension(int width, int height))
```

```
public void setMinimumSize(Dimension minimumSize)
```

Sets the minimum size of the button or label. Note that this is only a suggestion to the layout manager. The layout manager is not required to respect this minimum size.

Although we do not discuss the `Dimension` class, the following special case is intuitively clear and will work for most simple situations. The `int` values give the width and height in pixels.

```
public void setMinimumSize(  
    new Dimension(int width, int height))
```

THE `Dimension` CLASS

Objects of the class `Dimension` are used with buttons, menu items, and other objects to specify a size. The `Dimension` class is in the package `java.awt`. The parameters in the following constructor are pixels.

CONSTRUCTOR:

```
Dimension(int width, int height)
```

EXAMPLE:

```
aButton.setPreferredSize(new Dimension(30, 50));
```

THE `setActionCommand` METHOD

When the user clicks a button or menu item, that fires an action event that normally goes to one or more action listeners where it becomes an argument to an `actionPerformed` method. This action event includes a `String` instance variable that is known as the **action command** for the button or menu item and that is retrieved with the accessor method `getActionCommand`. The action event in the event is copied from an instance variable in the button or menu item object. If you do nothing to change it, the action command is the string written on the button or the menu item. The method `setActionCommand` given in Display 16.15 for the class `AbstractButton` can be used with any `JButton` or `JMenuItem` to change the action command for that component.

action command

Among other things, this will allow you to have different action commands for two buttons, two menu items, or a button and menu item even though they have the same string written on them.

setAction-
Command

The method `setActionCommand` takes a `String` argument that becomes the new action command for the calling button or menu item. For example, consider the following code:

```
 JButton nextButton = new JButton("Next");
 nextButton.setActionCommand("Next Button");
 JMenuItem chooseNext = new JMenuItem("Next");
 chooseNext.setActionCommand("Next Menu Item");
```

If we had not used `setActionCommand` in the preceding code, then the button `nextButton` and the menu item `chooseNext` would both have the action command "Next" and so we would have no way to tell which of the two components `nextButton` and `chooseNext` an action event "Next" came from. However, using the method `setActionCommand`, we can give them the different action commands "Next Button" and "Next Menu Item".

The action command for a `JButton` or `JMenuItem` is kept as the value of a private instance variable for the `JButton` or `JMenuItem`. The method `setActionCommand` is simply an ordinary mutator method that changes the value of this instance variable.

setActionCommand AND getActionCommand

Every button and every menu item has a string associated with it that is known as the **action command** for that button or menu item. When the button or menu item is clicked, it fires an action event `e`. The following invocation returns the action command for the button or menu item that fired `e`:

```
e.getActionCommand()
```

The method `actionPerformed` typically uses this action command string to decide which button or menu item was clicked.

The default action command for a button or menu item is the string written on it, but if you want, you can change the action command with an invocation of the method `setActionCommand`. For example, the menu item `chooseNext` created by the following code will display the string "Next" when it is a menu choice, but will have the string "Next Menu Item" as its action command.

EXAMPLE:

```
JMenuItem chooseNext = new JMenuItem("Next");
chooseNext.setActionCommand("Next Menu Item");
```

An alternate approach to defining action listeners is given in the next subsection. That technique is, among other things, another way to deal with multiple buttons or menu items that have the same thing written on them.

■ LISTENERS AS INNER CLASSES ❖

In all of our previous examples, our GUIs had only one action listener object to deal with all action events from all buttons and menus in the GUI. The opposite extreme also has much to recommend it. If you have a separate `ActionListener` class for each button or menu item, then each button or menu item can have its own unique action listener. There is then no need for a multiway `if-else` statement. The listener knows which button or menu items was clicked because it listens to only one button or menu item.

The approach outlined in the previous paragraph does have one down side: You typically need to give a lot of definitions of `ActionListener` classes. Rather than putting each of these classes in a separate file, it is much cleaner to make them private inner classes. This has the added advantage of allowing the `ActionListener` classes to have access to private instance variables and methods of the outer class.

In Display 16.16 we have redone the GUI in Display 16.14 using the techniques of this subsection.

Self-Test Exercises

28. What type of event is fired when you click a `JMenuItem`? How does it differ from the type of event fired when you click a `JButton` ?
29. Write code to create a `JButton` with "Hello" written on it but with "Bye" as its action command.
30. Write code to create a `JMenuItem` with "Hello" as its displayed text (when it is a choice in a menu) but with "Bye" as its action command.
31. If you want to change the action command for a `JButton` , you use the method `setActionCommand` . What method do you use to change the action command for a `JMenuItem` ?
32. Is the following legal in Java?

```
JMenu aMenu = new JMenu();
...
JMenu aSubMenu = new JMenu();
...
aMenu.add(aSubMenu);
```

33. How many `JMenuBar` objects can you have in a `JFrame` ?
34. A `JFrame` has a private instance variable of type `JMenuBar` . What is the name of the mutator method to change the value of this instance variable?

Display 16.16 Listeners as Inner Classes (Part 1 of 2)

<Import statements are the same as in Display 16.14.>

```
1 public class InnerListenersDemo extends JFrame
2 {
3     public static final int WIDTH = 300;
4     public static final int HEIGHT = 200;
5
6     private JPanel redPanel;
7     private JPanel whitePanel;
8     private JPanel bluePanel;
9
10    private class RedListener implements ActionListener
11    {
12        public void actionPerformed(ActionEvent e)
13        {
14            redPanel.setBackground(Color.RED);
15        }
16    } //End of RedListener inner class
17
18    private class WhiteListener implements ActionListener
19    {
20        public void actionPerformed(ActionEvent e)
21        {
22            whitePanel.setBackground(Color.WHITE);
23        }
24    } //End of WhiteListener inner class
25
26    private class BlueListener implements ActionListener
27    {
28        public void actionPerformed(ActionEvent e)
29        {
30            bluePanel.setBackground(Color.BLUE);
31        }
32    } //End of BlueListener inner class
33
34    public static void main(String[] args)
35    {
36        InnerListenersDemo gui = new InnerListenersDemo();
37        gui.setVisible(true);
38    }
39 }
```

Display 16.16 Listeners as Inner Classes (Part 2 of 2)

The resulting GUI is the same as in Display 16.14.

```
34     public InnerListenersDemo()
35     {
36         super("Menu Demonstration");
37         setSize(WIDTH, HEIGHT);
38         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
39         Container contentPane = getContentPane();
40         contentPane.setLayout(new GridLayout(1, 3));
41
42         redPanel = new JPanel();
43         redPanel.setBackground(Color.LIGHT_GRAY);
44         contentPane.add(redPanel);
45
46         whitePanel = new JPanel();
47         whitePanel.setBackground(Color.LIGHT_GRAY);
48         contentPane.add(whitePanel);
49
50         bluePanel = new JPanel();
51         bluePanel.setBackground(Color.LIGHT_GRAY);
52         contentPane.add(bluePanel);
53
54         JMenu colorMenu = new JMenu("Add Colors");
55
56         JMenuItem redChoice = new JMenuItem("Red");
57         redChoice.addActionListener(new RedListener());
58         colorMenu.add(redChoice);
59
60         JMenuItem whiteChoice = new JMenuItem("White");
61         whiteChoice.addActionListener(new WhiteListener());
62         colorMenu.add(whiteChoice);
63
64         JMenuItem blueChoice = new JMenuItem("Blue");
65         blueChoice.addActionListener(new BlueListener());
66         colorMenu.add(blueChoice);
67
68         JMenuBar bar = new JMenuBar();
69         bar.add(colorMenu);
70         setJMenuBar(bar);
71     }
72 }
```

35. Write code to create a new menu item named `aChoice` that has the label "Exit".
36. Suppose you are defining a class called `MenuGUI` that is a derived class of the class `JFrame`. Write code to add the menu item `mItem` to the menu `m`. Then add `m` to the menu bar `mBar`, and then add the menu bar to the `JFrame` `MenuGUI`. Assume that this all takes place inside a constructor for `MenuGUI`. Also assume that everything has already been constructed with `new`, and that all necessary listeners are registered. You just need to do the adding of things.
37. ❖ How can you modify the program in Display 16.16 so that when the `Blue` menu item is clicked all three colors are shown? The `Red` and `White` choices remain the same. (Remember the menu items may be clicked in any order, so the `Blue` menu item can be the first or second item clicked.)
38. ❖ Rewrite the Swing GUI in Display 16.16 so that there is only one action listener inner class. The inner class constructor will have two parameters, one for a panel and one for a color.

16.5

Text Fields and Text Areas

Write your answers in the spaces provided.

Common instruction for an examination

You have undoubtedly interacted with windowing systems that provide spaces for you to enter text information such as your name, address, and credit card number. In this section, we show you how to add these fields for text input and text output to your Swing GUIs.

TEXT AREAS AND TEXT FIELDS

text field
`JTextField`

A **text field** is an object of the class `JTextField` and is displayed as a field that allows the user to enter a single line of text. In Display 16.17 the following creates a text field named `name` in which the user will be asked to enter his or her name:

```
private JTextField name;
...
name = new JTextField(NUMBER_OF_CHAR);
```

In Display 16.17 the variable `name` is a private instance variable. The creation of the `JTextField` in the last of the previous lines takes place inside the class constructor. The number `NUMBER_OF_CHAR` that is given as an argument to the `JTextField` constructor specifies that the text field will have room for at least `NUMBER_OF_CHAR` characters to be visible. The defined constant `NUMBER_OF_CHAR` is 30, so the text field is guaranteed to have room for at least 30 characters. You can type any number of characters into a text

Display 16.17 A Text Field (Part 1 of 3)

```
1  import javax.swing.JFrame;
2  import javax.swing.JTextField;
3  import javax.swing.JPanel;
4  import javax.swing.JLabel;
5  import javax.swing.JButton;
6  import java.awt.Container;
7  import java.awt.GridLayout;
8  import java.awt.BorderLayout;
9  import java.awt.FlowLayout;
10 import java.awt.Color;
11 import java.awt.event.ActionListener;
12 import java.awt.event.ActionEvent;

13 public class TextFieldDemo extends JFrame
14     implements ActionListener
15 {
16     public static final int WIDTH = 400;
17     public static final int HEIGHT = 200;
18     public static final int NUMBER_OF_CHAR = 30;

19     private JTextField name;

20     public static void main(String[] args)
21     {
22         TextFieldDemo gui = new TextFieldDemo();
23         gui.setVisible(true);
24     }

25     public TextFieldDemo()
26     {
27         super("Text Field Demo");
28         setSize(WIDTH, HEIGHT);
29         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30         Container content = getContentPane();
31         content.setLayout(new GridLayout(2, 1));

32         JPanel namePanel = new JPanel();
33         namePanel.setLayout(new BorderLayout());
34         namePanel.setBackground(Color.WHITE);

35         name = new JTextField(NUMBER_OF_CHAR);
```

Display 16.17 A Text Field (Part 2 of 3)

```

36     namePanel.add(name, BorderLayout.SOUTH);
37     JLabel nameLabel = new JLabel("Enter your name here:");
38     namePanel.add(nameLabel, BorderLayout.CENTER);

39     content.add(namePanel);

40     JPanel buttonPanel = new JPanel();
41     buttonPanel.setLayout(new FlowLayout());
42     buttonPanel.setBackground(Color.PINK);
43     JButton actionButton = new JButton("Click me");
44     actionButton.addActionListener(this);
45     buttonPanel.add(actionButton);

46     JButton clearButton = new JButton("Clear");
47     clearButton.addActionListener(this);
48     buttonPanel.add(clearButton);

49     content.add(buttonPanel);
50 }

51 public void actionPerformed(ActionEvent e)
52 {
53     String actionCommand = e.getActionCommand();

54     if (actionCommand.equals("Click me"))
55         name.setText("Hello " + name.getText());
56     else if (actionCommand.equals("Clear"))
57         name.setText("");
58     else
59         name.setText("Unexpected error.");
60 }

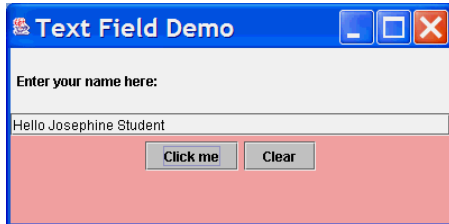
61 }

```

This sets the text field equal to the empty string, which makes it blank.

RESULTING GUI (When program is started and a name entered)



Display 16.17 A Text Field (Part 3 of 3)**RESULTING GUI** (After clicking the "Click me" button)

field but only a limited number will be visible; in this case, you know that at least 30 characters will be visible.

A Swing GUI can read the text in a text field and so receive text input, and if that is desired, can produce output by causing text to appear in the text field. The method `getText` returns the text written in the text field. For example, the following will set a variable named `inputString` to whatever string is in the text field `name` at the time that the `getText` method is invoked:

```
String inputString = name.getText();
```

The method `getText` is an input method, and the method `setText` is an output method. The method `setText` can be used to display a new text string in a text field. For example, the following will cause the text field `name` to change the text it displays to the string "This is some output":

```
name.setText("This is some output");
```

The following line from the method `actionPerformed` in Display 16.17 uses both `getText` and `setText`:

```
name.setText("Hello " + name.getText());
```

This line changes the string in the text field `name` to "Hello " followed by the old string value in the text field. The net effect is to insert the string "Hello " in front of the string displayed in the text field.

A **text area** is an object of the class `JTextArea`. A text area is the same as a text field except that it allows multiple lines. Two parameters to the constructor for `JTextArea` specify the minimum number of lines and the minimum number of characters per line that are guaranteed to be visible. You can enter any amount of text in a text area, but only a limited number of lines and a limited number of characters per line will be visible. For example, the following creates a `JTextArea` named `theText` that will have at least 5 lines and at least 20 characters per line visible:

```
JTextArea theText = new JTextArea(5, 20);
```

text area
`JTextArea`

getText AND setText

The classes `JTextField` and `JTextArea` both contain methods called `getText` and `setText`. The method `getText` can be used to retrieve the text written in the text field or text area. The method `setText` can be used to change the text written in the text field or text area.

SYNTAX:

`Name_of_Text_Component.getText()` returns the text currently displayed in the text field or text area.

`Name_of_Text_Component.setText(New_String_To_Display);`

EXAMPLES:

```
String inputString = ioComponent.getText();
ioComponent.setText("Hello out there!");
```

`ioComponent` may be an instance of either of the classes `JTextField` or `JTextArea`.

There is also a constructor with one additional `String` parameter for the string initially displayed in the text area. For example:

```
JTextArea theText = new JTextArea("Enter\ntext here.", 5, 20);
```

Note that a string value can be multiple lines because it can contain the new-line character `'\n'`.

A `JTextField` has a similar constructor with a `String` parameter, as in the following example:

```
JTextField ioField =
    new JTextField("Enter numbers here.", 30);
```

If you look at Display 16.12, you will see that both `JTextField` and `JTextArea` are derived classes of the abstract class `JTextComponent`. Most of the methods for `JTextField` and `JTextArea` are inherited from `JTextComponent` and so `JTextField` and `JTextArea` have mostly the same methods with the same meanings except for minor redefinitions to account for having just one line or multiple lines. Display 16.18 describes some methods in the class `JTextComponent`. All of these methods are inherited and have the described meaning in both `JTextField` and `JTextArea`.

You can set the line-wrapping policy for a `JTextArea` using the method `setLineWrap`. The method takes one argument of type `boolean`. If the argument is `true`, then at the end of a line, any additional characters for that line will appear on the following line of the text area. If the argument is `false`, the extra characters will be on the same line and will not be visible. For example, the following sets the line wrap policy for the

`setLineWrap`

Display 16.18 Some Methods in the Class JTextComponent

All these methods are inherited by the classes `JTextField` and `JTextArea`.

The abstract class `JTextComponent` is in the package `javax.swing.text`. The classes `JTextField` and `JTextArea` are in the package `javax.swing`.

```
public String getText()
```

Returns the text that is displayed by this text component.

```
public boolean isEditable()
```

Returns `true` if the user can write in this text component. Returns `false` if the user is not allowed to write in this text component.

```
public void setBackground(Color theColor)
```

Sets the background color of this text component.

```
public void setEditable(boolean argument)
```

If `argument` is `true`, then the user is allowed to write in the text component. If `argument` is `false`, then the user is not allowed to write in the text component.

```
public void setText(String text)
```

Sets the text that is displayed by this text component to be the specified text.

`JTextArea` object named `theText` so that at the end of a line, any additional characters for that line will appear on the following line:

```
theText.setLineWrap(true);
```

You can specify that a `JTextField` or `JTextArea` cannot be written in by the user. To do so, use the method `setEditable`, which is a method in both the `JTextField` and `JTextArea` classes. If `theText` names an object in either of the classes `JTextField` or `JTextArea`, then the following

```
theText.setEditable(false);
```

will set `theText` so that only your GUI program can change the text in the text component `theText`; the user cannot change the text. After this invocation of `setEditable`, if the user clicks the mouse in the text component named `theText` and then types at the keyboard, the text in the text component will not change.

To reverse things and make `theText` so that the user can change the text in the text component, use `true` in place of `false`, as follows:

```
theText.setEditable(true);
```

If no invocation of `setEditable` is made, then the default state allows the user to change the text in the text component.

output-only
`setEditable`

THE CLASSES JTextField AND JTextArea

The classes `JTextField` and `JTextArea` can be used to add areas for changeable text to a GUI. An object of the class `JTextField` has one line that displays some specified number of characters. An object of the class `JTextArea` has a size consisting of a specified number of lines and a specified number of characters per line. More text can be typed into a `JTextField` or `JTextArea` than is specified in its size, but the extra text may not be visible.

The number of characters per line and the number of lines are a guaranteed minimum. More lines and especially more characters per line may be visible. (The space per line is actually guaranteed to be *Characters_Per_Line* times the space for one uppercase letter M.)

SYNTAX:

```
JTextField Name_of_Text_Field = new JTextField(Characters_Per_Line);
JTextArea Name_of_Text_Area =
    new JTextArea(Number_of_Lines, Characters_Per_Line);
```

EXAMPLES:

```
JTextField name = new JTextField(30);
JTextArea someText = new JTextArea(10, 30);
```

There are also constructors that take an additional `String` argument that specifies an initial string to display in the text component.

SYNTAX:

```
JTextField Name_of_Text_Field =
    new JTextField(Initial_String, Characters_Per_Line);
JTextArea Name_of_Text_Area =
    new JTextArea(Initial_String, Number_of_Lines, Characters_Per_Line);
```

EXAMPLES:

```
JTextField name = new JTextField("Enter name here.", 30);
JTextArea someText =
    new JTextArea("Enter story here.\nClick button.", 10, 30);
```

NUMBER OF CHARACTERS PER LINE

The number of characters per line (given as an argument to constructors for `JTextField` or `JTextArea`) is not the number of just any characters. The number gives the number of em spaces in the line. An **em space** is the space needed to hold one uppercase letter M, which is the widest letter in the alphabet. So a line that is specified to hold 20 characters will always be able to hold at least 20 characters and will almost always hold more than 20 characters.

SCROLL BARS

Scroll bars for text areas and text fields are discussed in Chapter 18. They are a nice touch, but until you reach Chapter 18, your GUI programs will work fine without them.

Tip

LABELING A TEXT FIELD

Sometimes you want to label a text field. For example, suppose the GUI asks for a name and a credit card number and expects the user to enter these in two text fields. In this case, the GUI needs to label the two text fields so that the user knows in which field to type the name and in which field to type the credit card number. You can use an object of the class `JLabel` to label a text field or any other component in a Swing GUI. Simply place the label and text field in a `JPanel` and treat the `JPanel` as a single component. For example, we did this with the text field name in Display 16.17.

Self-Test Exercises

39. What is the difference between an object of the class `JTextArea` and an object of the class `JTextField`?
40. What would happen if when running the GUI in Display 16.17 you were to enter your name and click the "Click me" button three times?
41. Rewrite the program in Display 16.17 so that it uses a text area in place of a text field. Change the label "Enter your name here:" to "Enter your story here:". When the user clicks the "Click me" button, your GUI should change the string displayed in the text area to "Your story is " + `lineCount` + " lines long.". The variable `lineCount` is a variable of type `int` that your program sets equal to the number of lines currently displayed in the text area. Use a `BorderLayout` manager for the content pane, and place your text area in the region `BorderLayout.CENTER` so that there is room for it. You can assume the user enters at least one line before clicking the "Click me" button. The last line in the text area will have no `'\n'` and so you may need to add one if you are counting the number of occurrences of `'\n'`. Blank lines are counted.

Tip

INPUTTING AND OUTPUTTING NUMBERS

Just as was true for text input with `BufferedReader` and `JOptionPane`, when you want to input numbers using any Swing GUI, your GUI must convert input text to numbers. For example, when you input the number 42 in a `JTextField`, your program will receive the string "42", not

the number 42. Your program must convert the input string value "42" to the integer value 42. When you want to output numbers using a GUI constructed with Swing, you must convert the numbers to a string and then output that string. For example, if you want to output the number 40, your program would convert the integer value 40 to the string value "40". With Swing, all input typed by the user is string input and all displayed output is string output. The techniques for converting back and forth between strings and numbers were given in Chapter 2.

■ A SWING CALCULATOR

Designing a realistic Swing calculator is Programming Project 1. In this programming example we will develop a simplified calculator to get you started on that Programming Project. Display 16.19 contains a GUI for a calculator that keeps a running total of numbers. The user enters a number in the text field and then clicks either the + or – button. The number in the text field is then added into or subtracted from a running total that is kept in the instance variable `result`, and then the new total, the new value of `result`, is given in the text field. If the user clicks the "Reset" button, then the running total, the value of `result`, is set to zero. When the GUI is first run, the running total, the value of `result`, is set to zero.

Most of the details are similar to things you have already seen, but one new element is the use of exception handling. If the user enters a number in an incorrect format, such as placing a comma in a number, then one of the methods throws a `NumberFormatException`. If the user enters a number in an incorrect format, such as 2,000 with a comma instead of 2000, the method `assumingCorrectNumberFormats` invokes the method `stringToDouble` with the alleged number string "2,000" as an argument. Then `stringToDouble` calls `Double.parseDouble`, but `Double.parseDouble` throws a `NumberFormatException` because no Java number string can contain a comma. Since the invocation of `Double.parseDouble` takes place within an invocation of the method `stringToDouble`, `stringToDouble` in turn throws a `NumberFormatException`. The invocation of `stringToDouble` takes place inside the invocation of `assumingCorrectNumberFormats`, so `assumingCorrectNumberFormats` throws the `NumberFormatException` that it received from the invocation of `stringToDouble`. However, the invocation of `assumingCorrectNumberFormats` is inside a `try` block. The exception is caught in the following `catch` block. At that point, the `JTextField` (named `ioField`) is set to the error message "Error: Reenter Number."

Notice that if a `NumberFormatException` is thrown, the value of the instance variable `result` is not changed. A `NumberFormatException` can be thrown by an invocation of `stringToDouble` in either of the following lines of code from the method `assumingCorrectNumberFormats`:

```
result = result + stringToDouble(ioField.getText());
```

or

```
result = result - stringToDouble(ioField.getText());
```

Display 16.19 A Simple Calculator (Part 1 of 4)

```
1  import javax.swing.JFrame;
2  import javax.swing.JTextField;
3  import javax.swing.JPanel;
4  import javax.swing.JLabel;
5  import javax.swing.JButton;
6  import java.awt.Container;
7  import java.awt.BorderLayout;
8  import java.awt.FlowLayout;
9  import java.awt.Color;
10 import java.awt.event.ActionListener;
11 import java.awt.event.ActionEvent;

12 /**
13  A simplified calculator.
14  The only operations are addition and subtraction.
15  */
16 public class Calculator extends JFrame
17     implements ActionListener
18 {
19     public static final int WIDTH = 400;
20     public static final int HEIGHT = 200;
21     public static final int NUMBER_OF_DIGITS = 30;

22     private JTextField ioField;
23     private double result = 0.0;

24     public static void main(String[] args)
25     {
26         Calculator aCalculator = new Calculator();
27         aCalculator.setVisible(true);
28     }

29     public Calculator()
30     {
31         setTitle("Simplified Calculator");
32         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
33         setSize(WIDTH, HEIGHT);
34         Container contentPane = getContentPane();
35         contentPane.setLayout(new BorderLayout());

36         JPanel textPanel = new JPanel();
37         textPanel.setLayout(new FlowLayout());
38         ioField =
39             new JTextField("Enter numbers here.", NUMBER_OF_DIGITS);
40         ioField.setBackground(Color.WHITE);
41         textPanel.add(ioField);
42         contentPane.add(textPanel, BorderLayout.NORTH);
```

Display 16.19 A Simple Calculator (Part 2 of 4)

```
43     JPanel buttonPanel = new JPanel();
44     buttonPanel.setBackground(Color.BLUE);
45     buttonPanel.setLayout(new FlowLayout());

46     JButton addButton = new JButton("+");
47     addButton.addActionListener(this);
48     buttonPanel.add(addButton);
49     JButton subtractButton = new JButton("-");
50     subtractButton.addActionListener(this);
51     buttonPanel.add(subtractButton);
52     JButton resetButton = new JButton("Reset");
53     resetButton.addActionListener(this);
54     buttonPanel.add(resetButton);

55     contentPane.add(buttonPanel, BorderLayout.CENTER);
56 }
```

```
57 public void actionPerformed(ActionEvent e)
58 {
59     try
60     {
61         assumingCorrectNumberFormats(e);
62     }
63     catch (NumberFormatException e2)
64     {
65         ioField.setText("Error: Reenter Number.");
66     }
67 }
```

A `NumberFormatException` does not need to be declared or caught in a catch block.

```
68 //Throws NumberFormatException.
69 public void assumingCorrectNumberFormats(ActionEvent e)
70 {
71     String actionCommand = e.getActionCommand();

72     if (actionCommand.equals("+"))
73     {
74         result = result + stringToDouble(ioField.getText());
75         ioField.setText(Double.toString(result));
76     }
77     else if (actionCommand.equals("-"))
78     {
79         result = result - stringToDouble(ioField.getText());
80         ioField.setText(Double.toString(result));

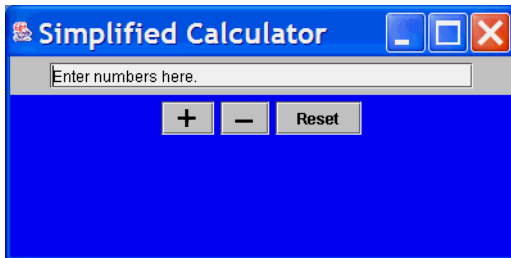
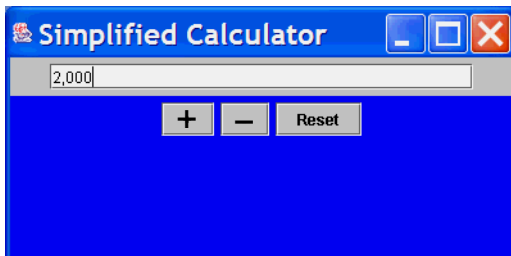
```

Display 16.19 A Simple Calculator (Part 3 of 4)

```
81     }
82     else if (actionCommand.equals("Reset"))
83     {
84         result = 0.0;
85         ioField.setText("0.0");
86     }
87     else
88         ioField.setText("Unexpected error.");
89 }

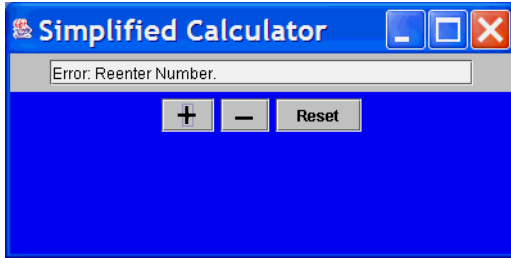
90 //Throws NumberFormatException.
91 private static double stringToDouble(String stringObject)
92 {
93     return Double.parseDouble(stringObject.trim());
94 }

95 }
```

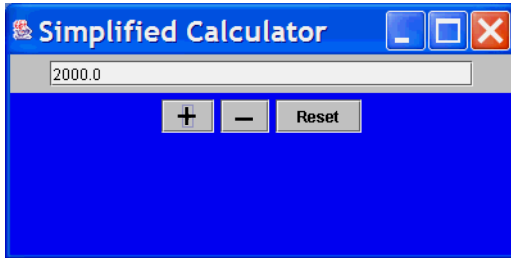
RESULTING GUI (When started)**RESULTING GUI** (After entering 2,000)

Display 16.19 A Simple Calculator (Part 4 of 4)

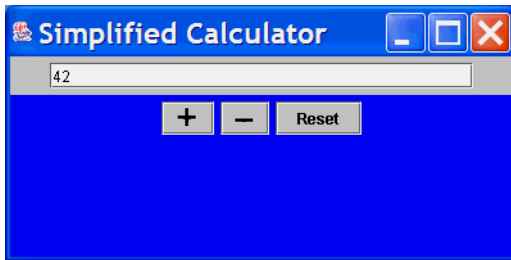
RESULTING GUI (After clicking +)



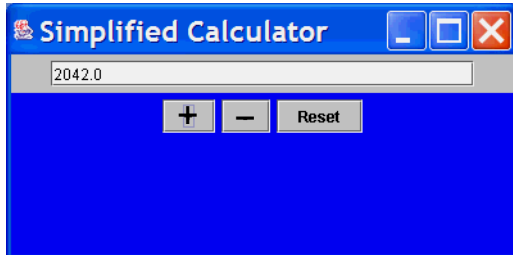
RESULTING GUI (After entering 2000 and clicking +)



RESULTING GUI (After entering 42)



RESULTING GUI (After clicking +)



If the exception is thrown, execution of the method `stringToDouble` ends immediately and control passes to the `catch` block. Thus, control passes to the `catch` block before the previous addition or subtraction is performed. So `result` is unchanged, and the user can reenter the last number and proceed with the GUI as if that incorrect number were never entered.

UNCAUGHT EXCEPTIONS

In a Swing program, throwing an uncaught exception does not end the GUI, but it may leave it in an unpredictable state. It is best to always catch any exception that is thrown even if all that the `catch` block does is output an instruction to redo something, such as reentering some input, or just outputs an error message.

Self-Test Exercises

42. In the GUI in Display 16.19, why did we make the text field `ioField` an instance variable but did not make instance variables of any of the buttons `addButton`, `subtractButton`, or `resetButton`?
43. What would happen if the user running the GUI in Display 16.19 were to run the GUI and simply click the addition button without typing anything into the text field?
44. What would happen if the user running the GUI in Display 16.19 were to type the number 10 into the text field and then click the addition button three times? Explain your answer.
45. Suppose you change the `main` method in Display 16.19 to the following:

```
public static void main(String[] args)
{
    Calculator calculator1 = new Calculator();
    calculator1.setVisible(true);

    Calculator calculator2 = new Calculator();
    calculator2.setVisible(true);
}
```

This will cause two calculator windows to be displayed. (If one is on top of the other, you can use your mouse to move the top one.) If you add numbers in one of these calculators, will anything change in the other calculator?

46. Suppose you change the `main` method in Display 16.19 as we described in exercise 45. This will cause two calculator windows to be displayed. If you click the close-window button in one of the windows, will one window go away or will both windows go away?
 - Swing GUIs (graphical user interfaces) are programmed using event-driven programming. In event-driven programming, a user action, like a button click, gener-

Chapter Summary

ates an event, and that event is automatically passed to an event-handling method that performs the appropriate action.

- There are two main techniques for designing a Swing GUI class. You can use inheritance to create a derived class of one of the library classes such as `JFrame` or you can build a GUI by adding components to a container class. You normally use both of these techniques when defining a Swing GUI class.
- A windowing GUI is usually defined as a derived class of the class `JFrame`.
- A button is an object of the class `JButton`. Clicking a button fires an action event that is handled by an action listener. An action listener is any class that implements the `ActionListener` interface.
- A label is an object of the class `JLabel`. You can use a label to add text to a GUI.
- When adding components to an object of a container class, such as adding a button to a panel or `JFrame`, you use the method `add`. The components in a container are arranged by an object called a layout manager.
- For an object of the class `JFrame`, you do not use the method `add` directly with the object. Instead, you use the method `getContentPane` to obtain the content pane of the `JFrame` object and you then use the `add` method with the content pane.
- A panel is a container object that is used to group components inside of a larger container. Panels are objects in the class `JPanel`.
- A menu item is a choice on a menu. A menu item is realized in your code as an object of the class `JMenuItem`. A menu is an object of the class `JMenu`. A menu item is added to a `JMenu` with the method `add`. A menu bar is an object of the class `JMenuBar`. A menu is added to a `JMenuBar` with the method `add`.
- A `JMenuBar` can be added to a `JFrame` with the method `setJMenuBar`. It can also be added using the method `add`, just as any other component can be added.
- Both buttons and menu items fire action events and so normally have an action listener registered with them to respond to the events.

ANSWERS TO SELF-TEST EXERCISES

1. The `JFrame` class.
2. Sizes in Swing are measured in pixels.
3.

```
someWindow.setDefaultCloseOperation(  
    JFrame.DO_NOTHING_ON_CLOSE);
```
4.

```
someWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```
5. When you click the minimizing button, the `JFrame` is reduced to an icon, usually at the bottom of your monitor screen.

6. `someWindow.setVisible(n > 0);`

The following also works but is not good style:

```
if (n > 0)
    someWindow.setVisible(true);
else
    someWindow.setVisible(false);
```

7. An action event.

8. `public void actionPerformed(ActionEvent e)`

9. Change

```
JFrame firstWindow = new JFrame();
```

to

```
JFrame firstWindow = new JFrame("My First Window");
```

Alternatively, you can add the following:

```
firstWindow.setTitle("My First Window");
```

10. Delete

```
setTitle("First Window Class");
```

and replace

```
super();
```

with

```
super("First Window Class");
```

11. Change

```
setDefaultCloseOperation(
    JFrame.DO_NOTHING_ON_CLOSE);
```

to

```
setDefaultCloseOperation(
    JFrame.EXIT_ON_CLOSE);
```

12. Change the following line in the no-argument constructor in Display 16.6 from

```
this(Color.WHITE);
```

to

```
this(Color.MAGENTA);
```

13. `Container contentPane = myFrame.getContentPane();`
14. `import java.awt.Container;`
15. `contentPane.add(new JLabel("Close-window button works."));`
16. Yes, it is legal. It is okay to reuse a variable name such as `aLabel`.
17. You need to change the add statements, as in the following rewritten section of code:

```
JLabel label1 = new JLabel("First label");
contentPane.add(label1, BorderLayout.NORTH);

JLabel label2 = new JLabel("Second label");
contentPane.add(label2, BorderLayout.CENTER);

JLabel label3 = new JLabel("Third label");
contentPane.add(label3, BorderLayout.SOUTH);
```

18. You need to change the add statements, as in the following rewritten section of code:

```
JLabel label1 = new JLabel("First label");
contentPane.add(label1, BorderLayout.NORTH);

JLabel label2 = new JLabel("Second label");
contentPane.add(label2, BorderLayout.EAST);

JLabel label3 = new JLabel("Third label");
contentPane.add(label3, BorderLayout.SOUTH);
```

19. The argument should be `new GridLayout(1, 3)`. So, the entire method invocation is

```
contentPane.setLayout(new GridLayout(1, 3));
```

Alternatively, you could use `new GridLayout(1, 0)`. It is also possible to do something similar with a `BorderLayout` manager or a `FlowLayout` manager, but a `GridLayout` manager will work nicer here.

20. The argument should be `new GridLayout(0, 1)`. So, the entire method invocation is

```
contentPane.setLayout(new GridLayout(0, 1));
```

Alternatively, you could use `new GridLayout(3, 1)`, if you know there will be at most three components added, but if more than three components are added, then a second column will be added. It is also possible to do something similar with a `BorderLayout` manager, but a `GridLayout` manager will work nicer here.

21. You need to use `getContentPane` when adding components to a `JFrame`. You do not use `getContentPane` when adding components to a `JPanel`.

22. `java.awt`

23. An object of the class `JPanel` is both a container class and a component class.
24. To make it look as though you have an empty grid element, add an empty panel to the grid element.
25.

```
import javax.swing.JPanel;
import java.awt.Color;
```

```
public class PinkJPanel extends JPanel
{
    public PinkJPanel()
    {
        setBackground(Color.PINK);
    }
}
```

The class `PinkJPanel` is on the CD that accompanies this text.

[extra code on CD](#)

26. It will not compile, but will give a compiler error message saying that `actionPerformed` is not defined (since it claims to implement the `ActionListener` interface).
27. It will not compile, but will give compiler error messages saying that, in effect, the invocations of `addActionListener` such as
- ```
redButton.addActionListener(this);
```
- have arguments of an incorrect type.
28. Clicking a `JMenuItem` fires an action event (that is, an object of the class `ActionEvent`). This is the same as with a `JButton`.
29. 

```
JButton b = new JButton("Hello");
b.setActionCommand("Bye");
```
30. 

```
JMenuItem m = new JMenuItem("Hello");
m.setActionCommand("Bye");
```
31. To change the action command for a `JMenuItem`, you use the method `setActionCommand`, just as you would for a `JButton`.
32. Yes, it is legal.
33. As many as you want. Only one can be added with the method `setJMenuBar`, but any number of others can be added to the content pane using the `add` method.
34. `setJMenuBar`
35. 

```
JMenuItem aChoice = new JMenuItem("Exit");
```
36. 

```
m.add(mItem);
mBar.add(m);
```

```
setJMenuBar(mBar);
```

You could use the following instead of using `setJMenuBar`:

```
getContentPane().add(mBar);
```

This will all take place inside a constructor named `MenuGUI`.

37. Register all three types of listeners with `blueChoice`, as follows:

```
blueChoice.addActionListener(new RedListener());
blueChoice.addActionListener(new WhiteListener());
blueChoice.addActionListener(new BlueListener());
```

38. Replace the three inner classes with the following inner class:

```
private class ColorListener implements ActionListener
{
 private JPanel thePanel;
 private Color theColor;

 public ColorListener(Color c, JPanel p)
 {
 theColor = c;
 thePanel = p;
 }

 public void actionPerformed(ActionEvent e)
 {
 thePanel.setBackground(theColor);
 }
} //End of ColorListener inner class
```

Replace

```
redChoice.addActionListener(new RedListener());
```

with

```
redChoice.addActionListener(
 new ColorListener(Color.RED, redPanel));
```

Also make similar changes to the menu items `whiteChoice` and `blueChoice`, with the obvious changes to colors and panels.

extra code on CD

This is not really preferable to what we did in Display 16.16, but it is a good exercise. The complete program done this way is on the accompanying CD in the file named `InnerListenersDemo2.java`.

39. A `JTextField` object displays only a single line. A `JTextArea` object can display more than one line of text.

40. The contents of the text field would change to "Hello Hello Hello " followed by your name.
41. This program is on the CD that accompanies this text.

[extra code on CD](#)

```
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JButton;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Color;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class TextAreaDemo extends JFrame
 implements ActionListener
{
 public static final int WIDTH = 400;
 public static final int HEIGHT = 200;
 public static final int NUMBER_OF_LINES = 10;
 public static final int NUMBER_OF_CHAR = 30;

 private JTextArea story;

 public static void main(String[] args)
 {
 TextAreaDemo gui = new TextAreaDemo();
 gui.setVisible(true);
 }

 public TextAreaDemo()
 {
 setTitle("Text Area Demo");
 setSize(WIDTH, HEIGHT);
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 Container content = getContentPane();
 content.setLayout(new GridLayout(2, 1));
 JPanel storyPanel = new JPanel();
 storyPanel.setLayout(new BorderLayout());
 storyPanel.setBackground(Color.WHITE);

 story = new JTextArea(NUMBER_OF_LINES, NUMBER_OF_CHAR);
```

```
 storyPanel.add(story, BorderLayout.CENTER);
 JLabel storyLabel = new JLabel("Enter your story here:");
 storyPanel.add(storyLabel, BorderLayout.NORTH);

 content.add(storyPanel);

 JPanel buttonPanel = new JPanel();
 buttonPanel.setLayout(new FlowLayout());
 buttonPanel.setBackground(Color.PINK);
 JButton actionButton = new JButton("Click me");
 actionButton.addActionListener(this);
 buttonPanel.add(actionButton);

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

 content.add(buttonPanel);
 }

 public void actionPerformed(ActionEvent e)
 {
 String actionCommand = e.getActionCommand();

 if (actionCommand.equals("Click me"))
 {
 int lineCount = getLineCount();
 story.setText("Your story is "
 + lineCount + " lines long.");
 }
 else if (actionCommand.equals("Clear"))
 story.setText("");
 else
 story.setText("Unexpected error.");
 }

 private int getLineCount()
 {
 String storyString = story.getText();
 int count = 0;

 for (int i = 0; i < storyString.length(); i++)
 if (storyString.charAt(i) == '\n')
```

```

 count++;

 return count + 1; //The last line has no '\n'.
 }
}

```

42. We made the text field an instance variable because we needed to refer to it in the definition of the method `actionPerformed`. On the other hand, the only direct reference we had to the buttons was in the constructor. So, we need names for the buttons only in the constructor definition.
43. The GUI would try to add the string "Enter numbers here." as if it were a string for a number. This will cause a `NumberFormatException` to be thrown and the string "Error: Reenter Number." would be displayed in the text field.
44. Every time the user clicks the addition button, the following assignment is executed:

```
result = result + stringToDouble(ioField.getText());
```

So, the number in the text field is added to the total as many times as the user clicks the addition button. But, the value in the text field is the running total, so the running total is added to itself. Thus, the running total is added to the total as many times as the user clicks the addition button.

Let's say that the user starts the GUI, types in 10, and clicks the addition button. That adds 10 to `result`, so the value of `result` is then 0 plus 10, which is 10, and 10 is displayed. Now the user clicks the addition button a second time. That adds 10 to `result` again, so the value of `result` is 10 plus 10, which is 20, and 20 is displayed. Next the user clicks the addition button a third time. This time, 20 is in the text field, and so it is added to `result`, which is also 20. Thus, the value of `result` is now 40, and 40 is displayed. Note that it is always the number in the text field that is added in.

45. The two calculator windows are completely independent. Each has its own instance variable `result`, which has no effect on the other's instance variable `result`.
46. If you click the close-window button in either calculator window, the entire program ends because that causes an invocation of `System.exit`. There is no invocation of `System.exit` in Display 16.19, but the following ensures that a `System.exit` that is in some library class will be invoked:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

## PROGRAMMING PROJECTS

1. Design and code a Swing GUI calculator. You can use Display 16.19 as a starting point, but your calculator will be more sophisticated. Your calculator will have two text fields that the user cannot change: One labeled "Result" will contain the result of performing the operation, and the other labeled "Operand" will be for the user to enter a number to be added, subtracted, and so forth from the result. The user enters the number for the "Operand" text field by clicking buttons labeled with the digits 0 through 9 and a decimal point, just as in a real calculator. Allow the operations of addition, subtraction, multiplication, and division. Use a `GridLayout` manager to produce a button pad that looks similar to the keyboard on a real calculator.

When the user clicks a button for an operation: the operation is performed, the "Result" text field is updated, and the "Operand" text field is cleared. Include a button labeled "Reset" that resets the "Result" to 0.0. Also include a button labeled "Clear" that resets the "Operand" text field so it is blank.

Define an exception class named `DivisonByZeroException`. Have your code throw and catch a `DivisonByZeroException` if the user attempts to "divide by zero." Your code will catch the `DivisonByZeroException` and output a suitable message to the "Operand" text field. The user may then enter a new substitute number in the "Operand" text field. Since values of type `double` are, in effect, approximate values, it makes no sense to test for equality with 0.0. Consider an operand to be "equal to zero" if it is in the range  $-1.0e-10$  to  $+1.0e-10$ .

2. (This is Programming Project 5 in Chapter 10, but done with a windowing interface.) Write a program that has a windowing interface and that gives and takes advice on program writing. The program starts by writing a piece of advice to the screen and asking the user to type in a different piece of advice. The program then ends. The next person to run the program receives the advice given by the person who last ran the program. The advice is kept in a text file and the content of the file changes after each run of the program. You can use your editor to enter the initial piece of advice in the file so that the first person who runs the program receives some advice. Allow the user to type in advice of any length so that it can be any number of lines long. The user is told to end his or her advice by pressing the return key two times. Your program can then test to see that it has reached the end of the input by checking to see when it reads two consecutive occurrences of the character `'\n'`. Alternatively, your program can simply test for an empty line marking the end of the file.
3. (This is Programming Project 1 in Chapter 10, but done with a windowing interface.) Write a program that has a windowing interface and that will search a text file of strings representing numbers of type `int` and output the largest and the smallest numbers. The file contains nothing but strings for numbers of type `int`, one per line. The windowing interface has a text field for entering the file name and two additional text fields that do not let the user write in them and that display the two numbers output. Your program should check to see that the named file exists and is readable, but need not do any other checks on the file.



4. (The Swing part of this project is pretty easy, but to do this programming project you need to know how to convert numbers from one base to another.) Write a program that converts integers from base ten (ordinary decimal) notation to base two notation. Use Swing to perform input and output via a window interface. The user enters a base ten numeral in one text field and clicks a button with "Convert" written on it; the equivalent base two numeral then appears in another text field. Be sure to label the two text fields. Include a "Clear" button that clears both text fields when clicked. (Hint: Include a private method that converts the string for a base ten numeral to the string for the equivalent base two numeral.)
5. (The Swing part of this project is pretty easy, but to do this programming project you need to know how to convert numbers from one base to another.) Write a program that converts integers from base two notation to base ten (ordinary decimal) notation. Use Swing to perform input and output via a window interface. The user enters a base two numeral in one text field and clicks a button with "Convert" written on it; the equivalent base ten numeral then appears in another text field. Be sure to label the two text fields. Include a "Clear" button that clears both text fields when clicked. (Hint: Include a private method that converts the string for a base two numeral to an equivalent `int` value.)
6. (It would help to do Programming Projects 4 and 5 before doing this one.) Write a program that converts integers from base two notation to base ten (ordinary decimal) notation and vice versa. Use Swing to perform input and output via a window interface. There are two text fields, one for base two numerals and one for base ten numerals. There are three buttons with the strings "To Base 10", "To Base 2", and "Clear". If the user enters a base two numeral in the base two text field and clicks the "To Base 10" button, the equivalent base ten numeral appears in the base ten text field. Similarly, if the user enters a base ten numeral in the base ten text field and clicks the "To Base 2" button, the equivalent base two numeral appears in the base two text field. Be sure that the text fields are labeled. If the user clicks the "Clear" button, that clears both text fields.