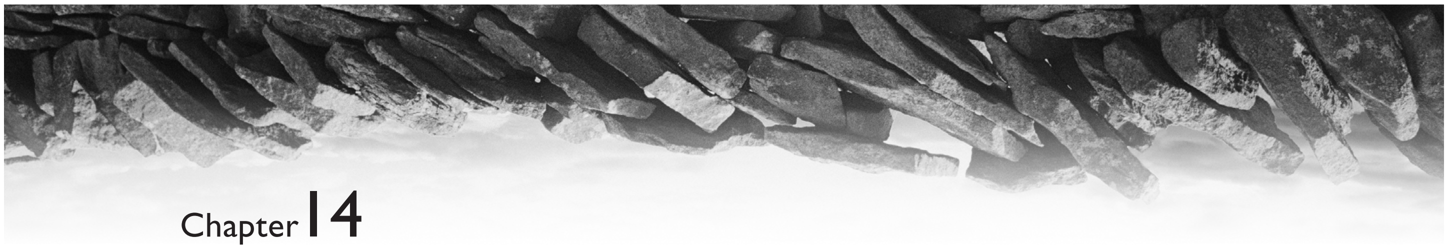


Chapter 14

Graphical User Interfaces

Introduction

In this chapter we will explore the creation of graphical user interfaces (GUIs). Although console programs like the ones we have written in the preceding chapters are still very important, the majority of modern desktop applications have graphical user interfaces. Supplement 3G introduced a `DrawingPanel` class that allowed you to draw two-dimensional graphics on the screen. This class is useful for certain applications, but writing a GUI is not the same as drawing shapes and lines onto a canvas. A real graphical user interface includes window frames which you create that contain buttons, text input fields, and other onscreen components.

A major part of creating a graphical user interface in Java is figuring out how to position and lay out the components of the user interface to match the appearance you desire. Once you have chosen and laid out these components, you must make the events interactive by making them respond to various user events such as button clicks or mouse movements. There are many predefined components, but you can also define components that draw custom two-dimensional graphics, including animations. At the end of this chapter, we will reimplement a basic version of the `DrawingPanel` class from Supplement 3G.

14.1 GUI Basics

VideoNote

GUIs are potentially very complex entities because they involve a large number of interacting objects and classes. Each onscreen component and window is represented by an object, so a programmer starting out with GUIs must learn many new class, method, and package names. In addition, if the GUI is to perform sophisticated tasks, the objects must interact with each other and call each other's methods, which raises tricky communication and scoping issues.

Another factor that makes writing GUIs challenging is that the path of code execution becomes nondeterministic. When a GUI program is running, the user can click any of the buttons and interact with any of the other onscreen components in any order. Because the program's execution is driven by the series of events that occur, we say that programs with GUIs are *event-driven.* In this chapter you'll learn how to handle user events so that your event-driven graphical programs will respond appropriately to user interaction.

Graphical Input and Output with Option Panes

The simplest way to create a graphical window in Java is to have an *option pane* pop up. An option pane is a simple message box that appears on the screen and presents a message or a request for input to the user.

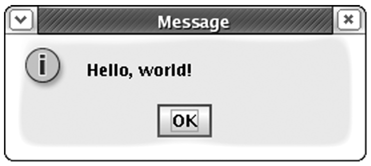
The Java class used to show option panes is called `JOptionPane`. `JOptionPane` belongs to the `javax.swing` package, so you'll need to import this package to use it. ("Swing" is the name of one of Java's GUI libraries.) Note that the package name starts with `javax` this time, not `java`. The `x` is because, in Java's early days, Swing was an extension to Java's feature set.

```
import javax.swing.*; // for GUI components
```

`JOptionPane` can be thought of as a rough graphical equivalent of `System.out.println` output and `Scanner` console input. The following program creates a "Hello, world!" message on the screen with the use of `JOptionPane`:

```
// A graphical equivalent of the classic "Hello world" program.

import javax.swing.*; // for GUI components

public class HelloWorld {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Hello, world!");
    }
}
```

The program produces the following graphical "output" (we'll show screenshots for the output of the programs in this chapter):



Hello, world!

The window may look slightly different in different operating systems, but the message will be the same.

The preceding program uses a static method in the `JOptionPane` class called `showMessageDialog`. This method accepts two parameters: a parent window and a message string to display. We don't have a parent window in this case, so we passed `null`.

`JOptionPane` can be used in three major ways: to display a message (as just demonstrated), to present a list of choices to the user, and to ask the user to type input. The three methods that implement these three behaviors are called `showMessageDialog`, `showConfirmDialog`, and `showInputDialog`, respectively. These methods are detailed in Table 14.1.

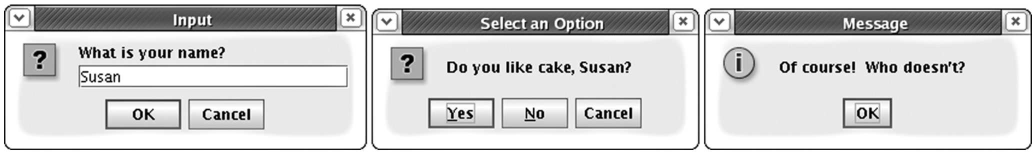
Table 14.1 Useful Methods of the `JOptionPane` Class

Method	Description
`showConfirmDialog(parent, message)`	Shows a Yes/No/Cancel message box containing the given message on the screen and returns the choice as an `int` with one of the following constant values: • `JOptionPane.YES_OPTION` (user clicked "Yes") • `JOptionPane.NO_OPTION` (user clicked "No") • `JOptionPane.CANCEL_OPTION` (user clicked "Cancel")
`showInputDialog(parent, message)`	Shows an input box containing the given message on the screen and returns the user's input value as a `String`
`showMessageDialog(parent, message)`	Shows the given message string in a message box on the screen

The following program briefly demonstrates all three types of option panes:

```java
// Shows several JOptionPane windows on the screen.

import javax.swing.*; // for GUI components

public class UseOptionPanes {
    public static void main(String[] args) {
        // read the user's name graphically
        String name = JOptionPane.showInputDialog(null,
                "What is your name?");

        // ask the user a yes/no question
        int choice = JOptionPane.showConfirmDialog(null,
                "Do you like cake, " + name + "?");

        // show different response depending on answer
        if (choice == JOptionPane.YES_OPTION) {
            JOptionPane.showMessageDialog(null,
                    "Of course! Who doesn't?");
        } else {  // choice == NO_OPTION or CANCEL_OPTION
            JOptionPane.showMessageDialog(null,
                    "We'll have to agree to disagree.");
        }
    }
}
```

The graphical input and output of this program is a series of windows, which pop up one at a time:



One limitation of `JOptionPane` is that its `showConfirmDialog` method always returns the user's input as a `String`. If you'd like to graphically request user input that is a number instead, your program must convert the `String` using the `Integer.parseInt` or `Double.parseDouble` method. These static methods accept a `String` as a parameter and return an `int` or `double` value, respectively.

The following program demonstrates the use of `JOptionPane` to read numbers from the user:

```
// Uses JOptionPane windows for numeric input.

import javax.swing.*; // for GUI components

public class UseOptionPanes2 {
    public static void main(String[] args) {
        String ageText = JOptionPane.showInputDialog(null,
                "How old are you?");
        int age = Integer.parseInt(ageText);

        String moneyText = JOptionPane.showInputDialog(null,
                "How much money do you have?");
        double money = Double.parseDouble(moneyText);

        JOptionPane.showMessageDialog(null,
                "If you can double your money each year,\n" +
                "You'll have " + (money * 32) +
                "dollars at age " + (age + 5) + "!");
    }
}
```

The `Integer.parseInt` and `Double.parseDouble` methods throw exceptions of type `NumberFormatException` if you pass them `String`s that cannot be converted into valid numbers, such as `"abc"`, `"five"`, or `"2×2"`. To make your code robust against such invalid input, you can enclose the code in a `try/catch` statement such as the following:

```
int age;
try {
    age = Integer.parseInt(ageText);
} catch (NumberFormatException nfe) {
    JOptionPane.showMessageDialog(null, "Invalid integer.");
}
```

Table 14.2 summarizes the static methods in the wrapper classes that can be used to convert `String`s.

Working with Frames

`JOptionPane` is useful, but on its own it is not flexible or powerful enough to create rich graphical user interfaces. To do that, you'll need to learn about the various types of widgets, or components, that can be placed on the screen in Java.

An onscreen window is called a *frame*.

Frame
A graphical window on the screen.

The graphical widgets inside a frame, such as buttons or text input fields, are collectively called *components*.

Component
A widget, such as a button or text field, that resides inside a graphical window.

Table 14.2 Useful Methods of Wrapper Classes

Method	Description
`Integer.parseInt(str)`	Returns the integer represented by the given `String` as an `int`
`Double.parseDouble(str)`	Returns the real number represented by the given `String` as a `double`
`Boolean.parseBoolean(str)`	Returns the boolean value represented by the given `String` (if the text is `"true"`, returns `true`; otherwise, returns `false`).

Technically, a frame is also a component, but we will treat frames differently from other components because frames form the physical windows that are seen onscreen. Frames also have a large number of methods not found in other components.

Figure 14.1 illustrates some of Java's more commonly used graphical components, listed by class name. A more complete pictorial reference of the available graphical components can be found in Sun's Java Tutorial at http://java.sun.com/docs/books/tutorial/uiswing/components/index.html.

Frames are represented by objects of the `JFrame` class. Any complex graphical program must construct a `JFrame` object to represent its main graphical window. Once you've constructed a `JFrame` object, you can display it on the screen by calling its `setVisible` method and passing it the `boolean` value `true`. Here's a simple program that constructs a frame and places it onscreen:

```java
// Shows an empty window frame on the screen.

import javax.swing.*;

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

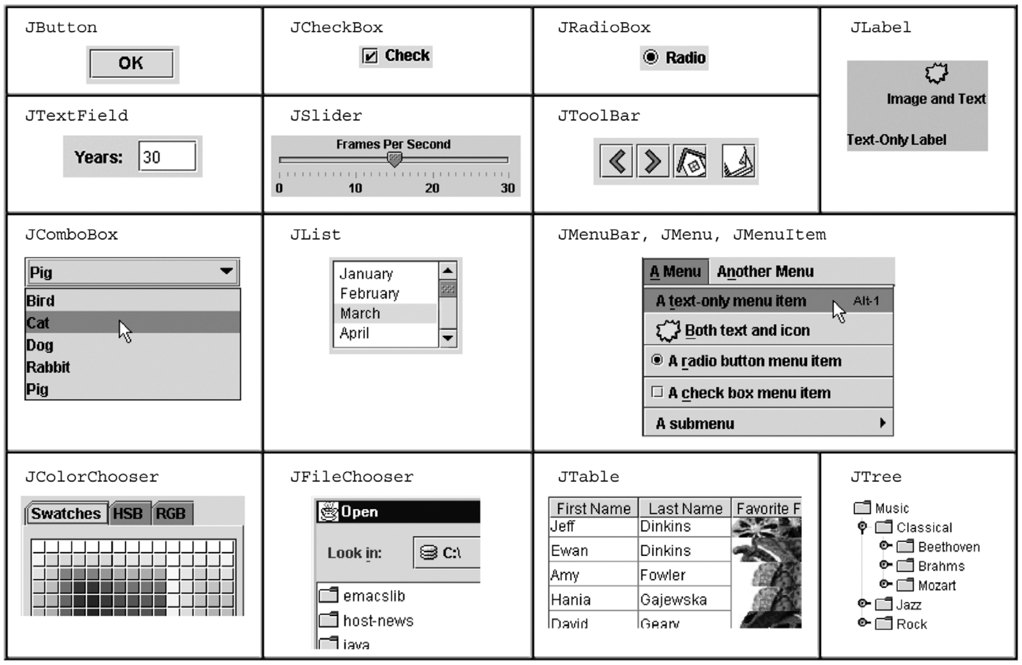
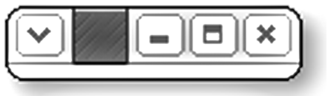


Figure 14.1 Some of Java's graphical components

The program's output is a bit silly—it's just a tiny window:



In fact, there is another problem with the program: Closing the window doesn't actually terminate the Java program. When you display a `JFrame` on the screen, by default Java does not exit the program when the frame is closed. You can tell that the program hasn't exited because a console window will remain on your screen (if you're using certain Java editors) or because your editor does not show its usual message that the program has terminated. If you want the program to exit when the window closes, you have to say so explicitly.

To create a more interesting frame, you'll have to modify some of the properties of `JFrames`. A *property* of a GUI component is a field or attribute it possesses internally that you may wish to examine or change. A frame's properties control features like the size of the window or the text that appears in the title bar. You can set or examine these properties' values by calling methods on the frame.

Table 14.3 lists several useful `JFrame` properties. For example, to set the title text of the frame in the `SimpleFrame` program to "A window frame", you'd write the following line of code:

```
frame.setTitle("A window frame");
```

Table 14.3 Useful Properties That Are Specific to `JFrames`

Property	Type	Description	Methods
default Close operation	`int`	What should happen when the frame is closed; choices include: • `JFrame.DO_NOTHING_ON_CLOSE` (don't do anything) • `JFrame.HIDE_ON_CLOSE` (hide the frame) • `JFrame.DISPOSE_ON_CLOSE` (hide and destroy the frame so that it cannot be shown again) • `JFrame.EXIT_ON_CLOSE` (exit the program)	`getDefaultCloseOperation`, `setDefaultCloseOperation(int)`
icon image	`Image`	The icon that appears in the title bar and Start menu or Dock	`getIconImage`, `setIconImage(Image)`
layout	`LayoutManager`	An object that controls the positions and sizes of the components inside this frame	`getLayout`, `setLayout(LayoutManager)`
resizable	`boolean`	Whether or not the frame allows itself to be resized	`isResizable`, `setResizable(boolean)`
title	`String`	The text that appears in the frame's title bar	`getTitle`, `setTitle(String)`

Table 14.4 Useful Properties of All Components (Including `JFrames`)

Property	Type	Description	Methods
background	`Color`	Background color	`getBackground`, `setBackground(Color)`
enabled	`boolean`	Whether the component can be interacted with	`isEnabled`, `setEnabled(boolean)`
focusable	`boolean`	Whether the keyboard can send input to the component	`isFocusable`, `setFocusable(boolean)`
font	`Font`	Font used to write text	`getFont`, `setFont(Font)`
foreground	`Color`	Foreground color	`getForeground`, `setForeground(Color)`
location	`Point`	(*x*, *y*) coordinate of component's top-left corner	`getLocation`, `setLocation(Point)`
size	`Dimension`	Current width and height of the component	`getSize`, `setSize(Dimension)`
preferred size	`Dimension`	"Preferred" width and height of the component; the size it should be to make it appear naturally on the screen (used with layout managers, seen later)	`getPreferredSize`, `setPreferredSize(Dimension)`
visible	`boolean`	Whether the component can be seen on the screen	`isVisible`, `setVisible(boolean)`

It turns out that all graphical components and frames share a common set of properties, because they exist in a common inheritance hierarchy. The Swing GUI framework is a powerful example of the code sharing of inheritance, since many components share features represented in common superclasses. Table 14.4 lists several useful common properties of frames/components and their respective methods.

Several of the properties in Table 14.4 are objects. The background and foreground properties are `Color` objects, which you may have seen previously in Supplement 3G. The location property is a `Point` object; the `Point` class was discussed in Chapter 8. The font property is a `Font` object; we'll discuss fonts later in this chapter. All of these types are found in the `java.awt` package, so be sure to import it, just as you did when you were drawing graphical programs with `DrawingPanel` in previous chapters:

```
import java.awt.*; // for various graphical objects
```

The size property is an object of type `Dimension`, which simply stores a width and height and is constructed with two integers representing those values. We'll use this property several times in this chapter. We only need to know a bit about the

Table 14.5 Useful Methods of `Dimension` Objects

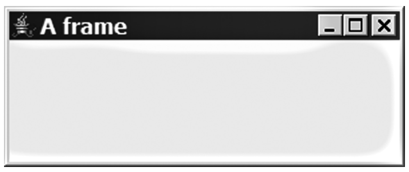
`public Dimension(int width, int height)`	Constructs a `Dimension` representing the given size
`public int getWidth()`	Returns the width represented by this `Dimension`
`public int getHeight()`	Returns the height represented by this `Dimension`

`Dimension` class, such as how to construct it. Table 14.5 lists the methods with which you should be familiar.

The new version of the `SimpleFrame` program that follows creates a frame and sets several of the properties listed in Table 14.4. In this version, we give the frame a color, set its location and size on the screen, and place text in its title bar. We also set the "default close operation" of the frame, telling it to shut down our Java program when the frame is closed:

```java
// Sets several properties of a window frame.

import java.awt.*; // for Dimension
import javax.swing.*; // for GUI components

public class SimpleFrame2 {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setForeground(Color.WHITE);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLocation(new Point(10, 50));
        frame.setSize(new Dimension(300, 120));
        frame.setTitle("A frame");
        frame.setVisible(true);
    }
}
```

When the program is run, the following window appears:

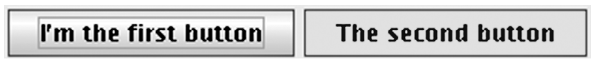


When the window is closed, the program exits.

Buttons, Text Fields, and Labels

Our empty frames are not very interesting. Let's look at some graphical components that can be placed in a frame.

The first component we'll examine is the *button.* You most likely know that a button is an onscreen component that can be clicked to cause an action. Buttons are represented by the `JButton` class. Each `JButton` object you create represents one button on the screen; if you want three buttons, you must create three `JButton` objects and place them in your frame.



You can construct a button either with no parameters (a button with no text), or with a `String` representing the text on the button. Of course, you can always change the button's text by calling the `setText` method on it. You can also set other properties of the button, such as its background color:

```
JButton button1 = new JButton();
button1.setText("I'm the first button");

JButton button2 = new JButton("The second button");
button2.setBackground(Color.YELLOW);
```

A *text field* is a box into which the user can type text strings. A text field is represented by a `JTextField` object, which can be constructed with a parameter specifying the number of characters that should be able to fit in the text field:

```
// creates a field 8 characters wide
JTextField field = new JTextField(8);
```

The character width you specify (`8` in the preceding code) is not enforced by the program; the user can type more than the specified number of characters if he or she wishes. The parameter value just affects the size of the text field on the screen.

A *label* is a string of text that is displayed on the GUI. Labels exist to provide information and are not generally clicked to perform actions. Labels are often placed next to text fields so that the user knows what should be typed into the field. A label is represented by a `JLabel` object, which can be constructed with a parameter specifying the label's text:

```
JLabel label = new JLabel("This is a label");
```

Merely constructing various component objects does not place them onto the screen; you must add them to the frame so it will display them. A frame acts as a *container,* or a region to which you can add graphical components. To add a component to a `JFrame`, call the frame's `add` method and pass the appropriate component as a parameter.

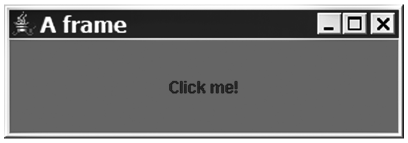
The following program creates a frame and places two buttons inside it:

```
// Creates a frame containing two buttons.

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

public class ComponentsExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(300, 100));
        frame.setTitle("A frame");

        JButton button1 = new JButton();
        button1.setText("I'm a button.");
        button1.setBackground(Color.BLUE);
        frame.add(button1);

        JButton button2 = new JButton();
        button2.setText("Click me!");
        button2.setBackground(Color.RED);
        frame.add(button2);

        frame.setVisible(true);
    }
}
```

You'll notice a pattern in this code. When you create a component, you must do the following things:

- Construct it.
- Set its properties, if necessary.
- Place it on the screen (in this case, by adding it to the frame).

The program produces the following graphical output:



This program's output is probably not what you expected. The first button isn't visible; it's hidden beneath the second button, which has been stretched to fill the entire frame. Let's explore the cause of this problem and learn how to solve it.

Changing a Frame's Layout

The problem in the previous program has to do with the *layout* of the components: the way that they are positioned, sized, and so on. We didn't tell the frame how to lay out the buttons, so it used a default behavior that positions each component in the center of the frame and extends it to fill the entire frame's space. When multiple components are added to the frame, the program places the last component on top and extends it to fill all the onscreen space.

Each button has size and location properties, but setting these in our code won't fix the problem. To fix the window's appearance, we must use an object called a *layout manager.*

Layout Manager

A Java object that determines the positions, sizes, and resizing behavior of the components within a frame or other container on the screen.

The frame has a layout manager object that it uses to position all the components inside it. Even if we set the positions and sizes of the buttons, the frame's layout manager will set them back to its default values. To position the components in a different way, we must set a new layout manager for the frame and use it to position the components.

Java contains many layout manager classes in its `java.awt` package, so make sure to import that package (along with `javax.swing` for the `JFrame` and other component classes):

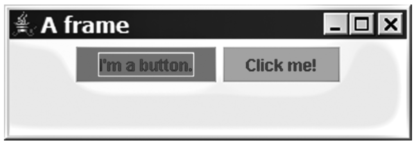
```
import java.awt.*;    // for layout managers
import javax.swing.*; // for GUI components
```

The default type of layout manager for a `JFrame` is called a `BorderLayout`, which we'll explore later in this chapter. If we want the buttons to flow in a left-to-right order instead, we can set the frame to use a different layout manager called

`FlowLayout`. We do this by inserting the following line in the program before adding the buttons:

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

Setting the layout in this manner ensures that both components will be visible in the frame. The program now produces the following graphical output:

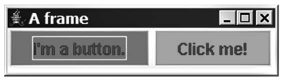


We'll discuss layout managers in detail in Section 14.2.

You may have noticed that the frame isn't sized quite right—there's extra space at the bottom and on the sides around the buttons. `JFrame` objects have a useful method called `pack` that resizes them exactly to fit their contents. If you call `pack`, you don't need to call `setSize` on the frame because `pack` will set its size for you. Calling this method just before you display the frame on the screen will ensure that the components fit snugly within it, without excess whitespace:

```
frame.pack();
frame.setVisible(true);
```

The new, packed frame has the following onscreen appearance:



Handling an Event

The user interfaces we've created so far are not interactive—nothing happens when the user clicks the buttons or types on the components. In order to create useful interactive GUIs, you must learn how to handle Java events. When the user clicks on a component, moves the mouse over it, or otherwise interacts with it, Java's GUI system creates a special kind of object called an *event* to represent this action.

Event

An object that represents a user's interaction with a GUI component and that can be handled by your programs to create interactive components.

By default, if you don't specify how to react to an event, it goes unnoticed by your program. Therefore, nothing happens when the user clicks your buttons or types in your text fields. You can cause the program to respond to a particular event (such as the user clicking a button) by using an object called a *listener.*

Listener

An object that is notified when an event occurs and that executes code to respond to the event.

To handle an event, create a listener object and attach it to the component of interest. The listener object contains the code that you want to run when the appropriate event occurs.

The first kind of event we'll handle in this chapter is called an *action event*. An action event is a fairly general type of event that occurs when the user interacts with many standard components (for example, clicking on a button or entering text into a `JTextField`).

In Java, event listeners are written by implementing particular interfaces. The interface for handling action events in Java is called `ActionListener`. The `ActionListener` interface is located in the `java.awt.event` package, which you'll need to import.

```
import java.awt.event.*; // for action events
```

Even if you've already imported the `java.awt` package, you'll still need to separately import the `java.awt.event` package, because it is not contained within `java.awt`.

The `ActionListener` interface contains only the following method:

```
public void actionPerformed(ActionEvent event)
```

To listen to an event, you write a class that implements the `ActionListener` interface and place into its `actionPerformed` method the code that you want to run when the event occurs. Then you attach an object of your listener class to the appropriate component.

Here is a simple example of an `ActionListener` class that responds to an event by displaying a message box on the screen:

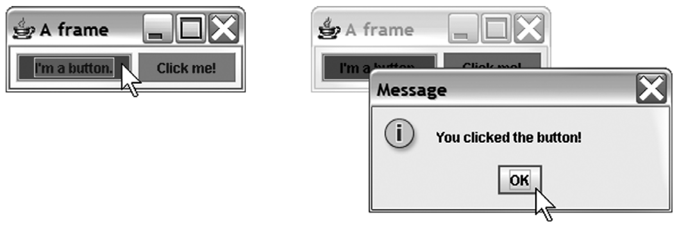
```
// Responds to a button click by displaying a message box.

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

public class MessageListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        JOptionPane.showMessageDialog(null,
                "You clicked the button!");
    }
}
```

Now that we've written this class, we can attach a `MessageListener` object to any button or other component of interest, and it will respond to action events on that component. For example, if we attach a `MessageListener` to a button, an option pane will pop up whenever that button is clicked. To attach the listener, we'll use a method called `addActionListener` that is found in several Swing components and that accepts a parameter of type `ActionListener`.

For example, we can add the following line to the `ComponentsExample` program we developed earlier to attach the listener to the first button:

```
// attach a listener to handle events on this button
button1.addActionListener(new MessageListener());
```

Here is the result when the program is executed and the user clicks on the button:



You clicked the button!

Note that a listener only responds to events on components to which it is added. If the user clicks the "Click me!" button, nothing will happen. If we want the same event to occur when the user clicks this button, we must add a `MessageListener` to it. If we want a different response to occur on a click of this button, we can write a second class that implements `ActionListener` and attach one of its objects to the button.

To summarize, follow these steps to handle an event in Java:

- Write a class that implements `ActionListener`.
- Place the code to handle the event into its `actionPerformed` method.
- Attach an object of your listener class to the component of interest using its `addActionListener` method.

VideoNote

14.2 Laying Out Components

Earlier, we used a `FlowLayout` to position the components in our frame. In this section, we will explore several different layout manager objects that can be used to position components in a variety of ways.

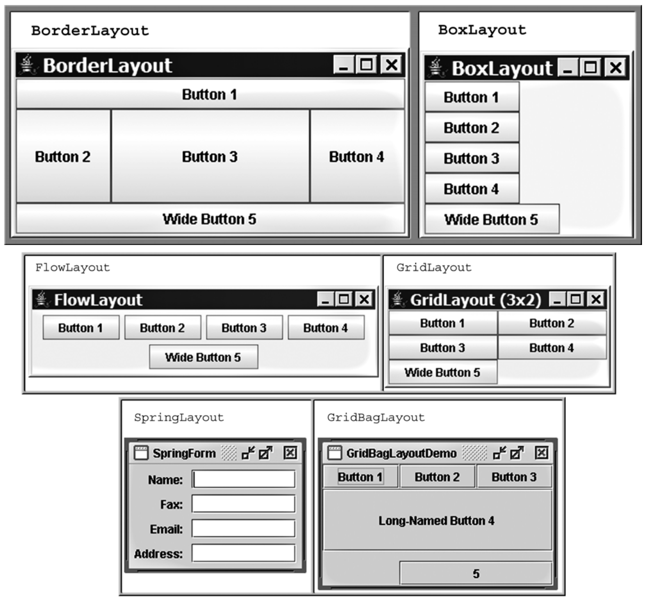


Figure 14.2 Layout managers

Layout Managers

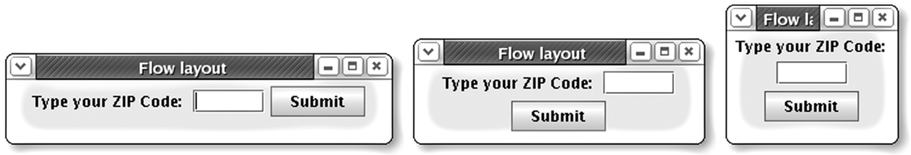
There are many useful layout managers with names that reflect the style in which they position components. Figure 14.2 illustrates some of the most common layout managers.

Perhaps the simplest layout manager is `FlowLayout`. As we discussed previously, a `FlowLayout` positions components in a left-to-right, top-to-bottom flow, similar to the way that words are arranged in a paragraph. If a row of components is too wide to fit in the frame, the row wraps.

Consider the following code, which adds three components to a frame that uses a `FlowLayout`:

```
JLabel label = new JLabel("Type your ZIP code: ");
JTextField field = new JTextField(5);
JButton button = new JButton("Submit");

frame.setLayout(new FlowLayout());
frame.add(label);
frame.add(field);
frame.add(button);
```

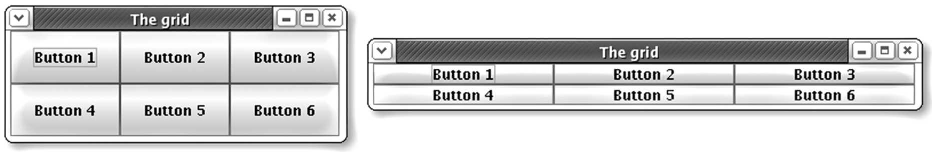
The graphical output that follows is produced when the frame is shown on the screen. Resizing the window illustrates the wrapping behavior:



Another common layout called `GridLayout` lays out components in a grid of rows and columns. You construct a `GridLayout` by passing it the number of rows and the number of columns. When you add components to the frame, they are laid out in equal-sized rectangles. The components are placed in the rectangles in row-major order (left to right, top to bottom). We'll use only buttons in this example, so that you can clearly see each component's size and shape:

```
// 2 rows, 3 columns
frame.setLayout(new GridLayout(2, 3));
for (int i = 1; i <= 6; i++) {
    frame.add(new JButton("Button " + i));
}
```

This code produces the graphical output that follows. As resizing the window demonstrates, `GridLayout` forces every component to be the same size, even though that can result in awkward stretching of the components.



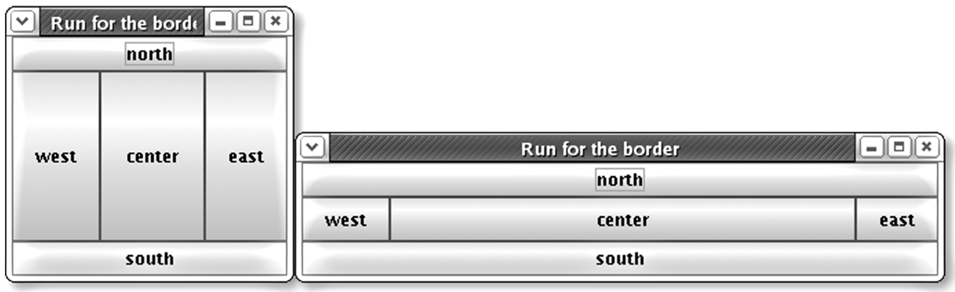
The third and final layout manager that we'll discuss in this section is called `BorderLayout`. A `BorderLayout` divides the frame into five sections: north, south, west, east, and center. When you use a `BorderLayout` to add components to a frame, you pass two parameters: the component to add and a constant representing the region in which to place it. The constants are called `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.WEST`, `BorderLayout.EAST`, and `BorderLayout.CENTER`. If you do not pass a second parameter representing the region, Java will place the component in the center.

The following program demonstrates the addition of several buttons to a `BorderLayout`:

```
// Lays out several components using a BorderLayout.

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

```

```
public class BorderLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(210, 200));
        frame.setTitle("Run for the border");

        frame.setLayout(new BorderLayout());
        frame.add(new JButton("north"), BorderLayout.NORTH);
        frame.add(new JButton("south"), BorderLayout.SOUTH);
        frame.add(new JButton("west"), BorderLayout.WEST);
        frame.add(new JButton("east"), BorderLayout.EAST);
        frame.add(new JButton("center"), BorderLayout.CENTER);

        frame.setVisible(true);
    }
}
```

Here's the graphical output of the program, both before and after resizing:



Notice the stretching behavior: The north and south buttons stretch horizontally but not vertically, the west and east buttons stretch vertically but not horizontally, and the center button gets all the remaining space. You don't have to place components in all five regions; if you leave a region empty, the center consumes its space.

Here is a summary of the stretching and resizing behaviors of the layout managers:

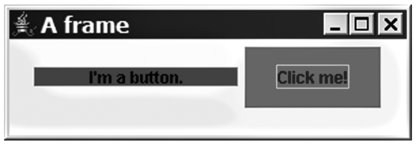
- `FlowLayout`: Does not stretch components. Wraps to next line if necessary.
- `GridLayout`: Stretches all components in both dimensions to make them equal in size at all times.
- `BorderLayout`: Stretches north and south regions horizontally but not vertically, stretches west and east regions vertically but not horizontally, and stretches center region in both dimensions to fill all remaining space not claimed by the other four regions.

Every graphical component has a property for its *preferred size*. By default a component's preferred size is just large enough to fit its contents (such as any text or images inside the component). The `FlowLayout` allows components to appear in the frame at their preferred sizes, while the `GridLayout` disregards its components' preferred sizes.

If you want to change a component's preferred size, you may set its `preferred size` property by calling its `setPreferredSize` method. For example, the `ComponentsExample` program from Section 14.1 created two buttons called `button1` and `button2` and set background colors for them. The following modification assigns different preferred sizes to the two buttons:

```
button1.setPreferredSize(new Dimension(150, 14));
button2.setPreferredSize(new Dimension(100, 45));
```

The frame's `FlowLayout` respects the buttons' preferred sizes, leading to the following onscreen appearance:



Composite Layouts

The individual layout managers are limited, but it is possible to position components in complex ways by using multiple layout managers together. Layout managers can be layered one on top of another to produce combined effects. This is called a *composite layout.*

Composite Layout

A layered layout that uses several layout managers in different nested containers.

A frame acts as a container for components, but it's possible to construct additional containers and use them to enhance the layout of those components. To create a composite layout, you'll need to use another type of container called a *panel*. A panel is an onscreen component that acts solely as a container for other components and generally cannot be seen. A panel has its own layout manager that controls the positions and sizes of any components inside it.

Panels are implemented as objects of the `JPanel` class. Each `JPanel` object has a layout manager, which you can specify either when you construct the panel or by calling its `setLayout` method. By creating several panels with different layout managers, you can control how different groups of components will be laid out. You can then add these panels to the overall frame so that they will appear on the screen.

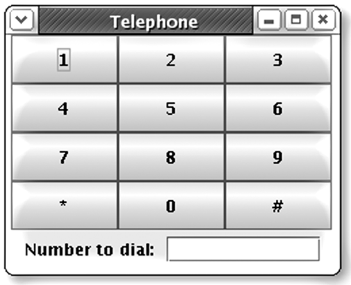
You can construct a `JPanel` object with no parameters, or you can specify the layout manager to use. Once you've constructed the panel, you can add components to it using its `add` method:

```
// creates a JPanel container with a FlowLayout
JPanel panel = new JPanel(new FlowLayout());
panel.add(new JButton("button 1"));
panel.add(new JButton("button 2"));
```

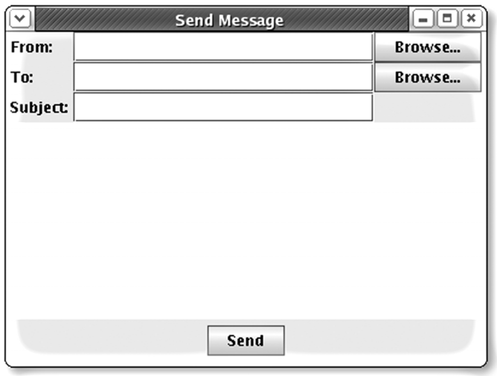
A common strategy is to use a `BorderLayout` for the frame, then add panels to some or all of its five regions. For example, the following code produces a window that looks like a telephone keypad, using a panel with a `GridLayout` in the center region of the frame and a second panel with a `FlowLayout` in the south region of the frame:

```
// A GUI that resembles a telephone keypad.

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

public class Telephone {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(250, 200));
        frame.setTitle("Telephone");

        frame.setLayout(new BorderLayout());

        // main phone buttons
        JPanel centerPanel = new JPanel(new GridLayout(4, 3));
        for (int i = 1; i <= 9; i++) {
            centerPanel.add(new JButton("" + i));
        }
        centerPanel.add(new JButton("*"));
        centerPanel.add(new JButton("0"));
        centerPanel.add(new JButton("#"));
        frame.add(centerPanel, BorderLayout.CENTER);

        // south status panel
        JPanel southPanel = new JPanel(new FlowLayout());
        southPanel.add(new JLabel("Number to dial: "));
        southPanel.add(new JTextField(10));
        frame.add(southPanel, BorderLayout.SOUTH);

        frame.setVisible(true);
    }
}
```

The preceding program uses a `for` loop to set the text of the first nine buttons to the integers 1 through 9, but it runs into the problem that the constructor for a `JButton` accepts a `String` as its parameter. To get around this issue, the `for` loop uses a trick to convert the integer value stored in `i` into a `String` by concatenating it with the empty string, `""`. The program's graphical output is the following:



It can be tricky to get a composite layout to look just right. You may want to practice with lots of examples and carefully consider the stretching behaviors of the various layouts. For example, how might you create a window that looks like this?

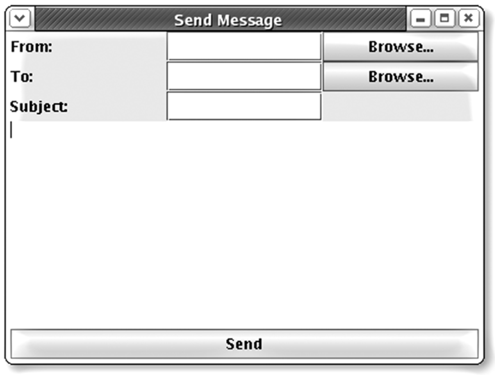


`BorderLayout` is the right choice for the frame, but how would you lay out the individual components? There are three rows of components at the top of the window, so your first thought might be to place those components in a panel with a 3×3 grid layout. You'd then add this layout to the north region of the frame, along with the central text area and a Send button in the south region:

```
// this code does not produce the right layout
frame.setLayout(new BorderLayout());
```

```
JPanel north = new JPanel(new GridLayout(3, 3));
north.add(new JLabel("From: "));
north.add(new JTextField());
north.add(new JButton("Browse..."));
north.add(new JLabel("To: "));
north.add(new JTextField());
north.add(new JButton("Browse..."));
north.add(new JLabel("Subject: "));
north.add(new JTextField());

frame.add(north, BorderLayout.NORTH);
frame.add(new JTextArea(), BorderLayout.CENTER);
frame.add(new JButton("Send"), BorderLayout.SOUTH);
```

This code produces the following graphical output, which is not correct:



There are a few problems with this output. First of all, because a grid layout forces every component inside it to take the same size, the labels, text fields, and buttons are not the proper sizes. You want all the labels to be the same size and all the text fields and buttons to be the same size, but you don't want the different kinds of components to all be the same size.

The simplest way to resolve this size problem is to create three separate panels with grid layout in the north region of the frame, each with three rows and only one column. You can then put the labels into the first grid, the text fields into the second, and the buttons into the third. (Even though there are only two buttons, make the layout 3×1 to leave a blank space to match the expected output.)

To position the three grids next to one another, add another layer by creating a master north panel with a border layout to store all three grids. The labels will occupy the panel's west region, the text fields will occupy the center, and the buttons will occupy the east region. Place this master north panel in the north region of the frame.

The other problem with the previous output is that the Send button is stretched. Because it was placed directly in the south region of a `BorderLayout`, Java stretches the button horizontally to fill the frame. To avoid this problem, put the Send button into a panel with a `FlowLayout`. `FlowLayout` doesn't stretch the components inside it, so the button won't grow to such an odd size. Then add the panel to the south region of the frame, rather than adding the Send button directly.

Here's a complete version of the program that contains these corrections and produces the proper graphical output:

```java
// Creates a GUI that resembles an email Compose Message window.

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

public class EmailMessage {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(400, 300));
        frame.setTitle("Send Message");
        frame.setLayout(new BorderLayout());

        JPanel northWest = new JPanel(new GridLayout(3, 1));
        northWest.add(new JLabel("From: "));
        northWest.add(new JLabel("To: "));
        northWest.add(new JLabel("Subject: "));

        JPanel northCenter = new JPanel(new GridLayout(3, 1));
        northCenter.add(new JTextField());
        northCenter.add(new JTextField());
        northCenter.add(new JTextField());

        JPanel northEast = new JPanel(new GridLayout(3, 1));
        northEast.add(new JButton("Browse..."));
        northEast.add(new JButton("Browse..."));

        JPanel north = new JPanel(new BorderLayout());
        north.add(northWest, BorderLayout.WEST);
        north.add(northCenter, BorderLayout.CENTER);
        north.add(northEast, BorderLayout.EAST);

        JPanel south = new JPanel(new FlowLayout());
        south.add(new JButton("Send"));

```

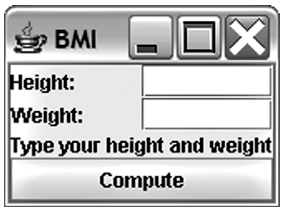
```
            frame.add(north, BorderLayout.NORTH);
            frame.add(new JTextArea(), BorderLayout.CENTER);
            frame.add(south, BorderLayout.SOUTH);

            frame.setVisible(true);
        }
    }
```

14.3 Interaction Between Components

In this section we'll write some larger and more complex graphical programs. These programs will raise new issues about communication between components and event listeners.

Example 1: BMI GUI

Consider the task of writing a graphical program to compute a person's body mass index (BMI). The program should have a way for the user to type in a height and a weight and should use these values to compute the person's BMI. A reasonable appearance for the GUI would be the following window, which uses text fields for input and a button to trigger a computation of the BMI:



Let's figure out how to create a GUI with the proper components and layout first and worry about event handling later. Since we want the program to display text fields with labels next to them, aligned in a row/column format, a 2×2 `GridLayout` is a good choice. But we'll want to use a composite layout, because we don't want the central label and the Compute button to have the same size and position as the grid squares. We'll use a `BorderLayout` on our frame and add the central "Type your height and weight" label and Compute button directly to it. We'll also create a panel with a 2×2 `GridLayout` and place it in the north region of the frame.

The code that follows implements the initial BMI user interface. The program sets the frame's title by passing it as a parameter to its constructor:

```
// A GUI to compute a person's body mass index (BMI).
// Initial version without event handling.

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

public class BmiGui1 {
    public static void main(String[] args) {
        // set up components
        JTextField heightField = new JTextField(5);
        JTextField weightField = new JTextField(5);
        JLabel bmiLabel = new JLabel(
                "Type your height and weight");
        JButton computeButton = new JButton("Compute");

        // layout
        JPanel north = new JPanel(new GridLayout(2, 2));
        north.add(new JLabel("Height: "));
        north.add(heightField);
        north.add(new JLabel("Weight: "));
        north.add(weightField);

        // overall frame
        JFrame frame = new JFrame("BMI");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout());
        frame.add(north, BorderLayout.NORTH);
        frame.add(bmiLabel, BorderLayout.CENTER);
        frame.add(computeButton, BorderLayout.SOUTH);
        frame.pack();
        frame.setVisible(true);
    }
}
```

The program currently does nothing when the user presses the Compute button.

Object-Oriented GUIs

Let's think about how to make our BMI GUI respond to clicks on the Compute button. Clicking a button causes an action event, so we might try putting an `ActionListener` on the button. But the code for the listener's `actionPerformed` method would need to read the text from the height and weight text fields and use that information to compute the BMI and display it as the text of the central label. This is a problem because the GUIs we've written so far were not built to allow so much interaction between components.

To enable listeners to access each of the GUI components, we need to make our GUI itself into an object. We'll declare the onscreen components as fields inside that object and initialize them in the GUI's constructor. In our `BmiGui1` example, we can convert much of the code that is currently in the `main` method into the constructor for the GUI.

The following class, which still has no event handling, implements the new object-oriented version of the GUI:

```
// A GUI to compute a person's body mass index (BMI).
// Object-oriented version without event handling.

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

public class BmiGui2 {
    // onscreen components stored as fields
    private JFrame frame;
    private JTextField heightField;
    private JTextField weightField;
    private JLabel bmiLabel;
    private JButton computeButton;

    public BmiGui2() {
        // set up components
        heightField = new JTextField(5);
        weightField = new JTextField(5);
        bmiLabel = new JLabel(
            "Type your height and weight");
        computeButton = new JButton("Compute");

        // layout
        JPanel north = new JPanel(new GridLayout(2, 2));
        north.add(new JLabel("Height: "));
        north.add(heightField);
        north.add(new JLabel("Weight: "));
        north.add(weightField);

        // overall frame
        frame = new JFrame("BMI");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout());
        frame.add(north, BorderLayout.NORTH);
        frame.add(bmiLabel, BorderLayout.CENTER);
        frame.add(computeButton, BorderLayout.SOUTH);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Now that the GUI is an object, we can write a separate client class to hold the `main` method to construct it. The following short class does the job:

```
// Shows a BMI GUI on the screen.

public class RunBmiGui2 {
    public static void main(String[] args) {
        BmiGui2 gui = new BmiGui2(); // construct/show GUI
    }
}
```

The advantage of an object-oriented GUI is that we can make it into an event listener. We want to add a listener to the Compute button to compute the user's BMI. This listener will need to access the height and weight text fields and the central BMI label. These components are fields within the `BmiGui2` object, so if the `BmiGui2` class itself implements `ActionListener`, it will have access to all the information it needs.

Let's write a third version of our BMI GUI that handles action events. We'll change our class name to `BmiGui3` and change our class header to implement the `ActionListener` interface:

```
public class BmiGui3 implements ActionListener {
```

To implement `ActionListener`, we must write an `actionPerformed` method. Our `actionPerformed` code will read the two text fields' values, convert them into type `double`, compute the BMI using the standard BMI formula of weight / height^2 * 703, and set the text of the central BMI label to show the BMI result. The following code implements the listener:

```
// Handles clicks on Compute button by computing the BMI.
public void actionPerformed(ActionEvent event) {
    // read height and weight info from text fields
    String heightText = heightField.getText();
    double height = Double.parseDouble(heightText);
    String weightText = weightField.getText();
    double weight = Double.parseDouble(weightText);

    // compute BMI and display it onscreen
    double bmi = weight / (height * height) * 703;
    bmiLabel.setText("BMI: " + bmi);
}
```

Now we have to attach the action listener to the Compute button. Since the GUI is the listener, the parameter we pass to the button's `addActionListener` method is the GUI itself. Because this code is inside the GUI object's constructor, we pass the GUI as a parameter using the keyword `this`:

```
// attach GUI as event listener to Compute button
computeButton.addActionListener(this);
```

The second version of our BMI GUI used a separate class called `RunBmiGui2` as a client to run the GUI. However, using a second class for such a minimal client program is a bit of a waste. It is actually legal to place the `main` method in the BMI GUI class itself and not use the `RunBmiGui2` class. If we do this, the class becomes its own client, and we can just compile and run the GUI class to execute the program. (Static methods like `main` are generally placed above any fields, constructors, and instance methods in the same file.)

After we implement the listener code and incorporate the `main` method, the BMI GUI program is complete:

```
// A GUI to compute a person's body mass index (BMI).
// Final version with event handling.

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

public class BmiGui3 implements ActionListener {
    // BmiGui3 is its own runnable client program
    public static void main(String[] args) {
        BmiGui3 gui = new BmiGui3();
    }

    // onscreen components stored as fields
    private JFrame frame;
    private JTextField heightField;
    private JTextField weightField;
    private JLabel bmiLabel;
    private JButton computeButton;

    public BmiGui3() {
        // set up components
        heightField = new JTextField(5);
        weightField = new JTextField(5);
        bmiLabel = new JLabel("Type your height and weight");
        computeButton = new JButton("Compute");

        // attach GUI as event listener to Compute button
        computeButton.addActionListener(this);

        // layout
        JPanel north = new JPanel(new GridLayout(2, 2));
        north.add(new JLabel("Height: "));
        north.add(heightField);
        north.add(new JLabel("Weight: "));
        north.add(weightField);

        // overall frame
        frame = new JFrame("BMI");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout());
        frame.add(north, BorderLayout.NORTH);
        frame.add(bmiLabel, BorderLayout.CENTER);
        frame.add(computeButton, BorderLayout.SOUTH);
        frame.pack();
        frame.setVisible(true);
    }

    // Handles clicks on Compute button by computing the BMI.
    public void actionPerformed(ActionEvent event) {
        // read height/weight info from text fields
        String heightText = heightField.getText();
        double height = Double.parseDouble(heightText);
        String weightText = weightField.getText();
        double weight = Double.parseDouble(weightText);

        // compute BMI and display it onscreen
        double bmi = weight / (height * height) * 703;
        bmiLabel.setText("BMI: " + bmi);
    }
}
```

Example 2: Credit Card GUI

Credit card numbers contain several pieces of information for performing validity tests. For example, Visa card numbers always begin with 4, and a valid Visa card number always passes a digit-sum test known as the Luhn checksum algorithm. Luhn's algorithm states that if you add the digits of any valid credit card number in a certain way, the sum will be a multiple of 10. Systems that accept credit cards perform a Luhn test before they contact the credit card company for final verification of the number. This test allows them to filter out fake or mistyped credit card numbers.

The algorithm for adding the digits can be described as follows: Consider each digit of the credit card number to have a zero-based index: The first is at index 0, and the last is at index 15. Start from the rightmost digit and process each digit one at a time. For each digit at an odd-numbered index (the 15th digit, 13th digit, etc.), simply add that digit to the cumulative sum. For each digit at an even-numbered index (the 14th, 12th, etc.), double the digit's value. If that doubled value is less than 10, add it to the sum; if the doubled value is 10 or greater, add each of its digits separately into the sum.

The following pseudocode describes the Luhn algorithm to add the digits:

```
sum = 0.
for (each digit of credit card number, starting from right) {
    if (digit's index is odd) {
        add digit to sum.
    } else {
        double the digit's value.
        if (doubled value < 10) {
           add doubled value to sum.
        } else {
           split doubled value into its two digits.
           add first digit to sum.
           add second digit to sum.
        }
    }
}
```

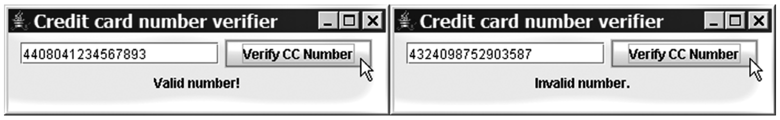
4111111111111111 and 4408041274369853 are sample credit card numbers that pass the Luhn algorithm. Figure 14.3 shows how the algorithm sums the latter number in detail. Notice how digits at even indexes are doubled and split into two digits if their new values are 10 or higher. For example, the number 7 at index 8 is doubled to 14, which is then split to make 1 + 4.

```
CC #     4408 0412 7436 9853

         4    4    0    8    0    4    1    2    7     4    3    6    9     8    5     3
Scale   ×2        ×2        ×2        ×2        ×2         ×2        ×2          ×2
        ---------------------------------------------------------------------------------
         8    4    0    8    0    4    2    2    14    4    6    6    18    8    10    3

Sum    = 8 +  4 +  0 +  8 +  0 +  4 +  2 +  2 + 1+4 + 4 +  6 +  6 + 1+8 + 8 + 1+0 + 3
       = 70
```

`70` is divisible by `10`, therefore, this card number is valid.

Figure 14.3 Example checksum using the Luhn algorithm

Let's write a GUI that allows the user to type in a credit card number, press a button to verify the number, and receive a message stating whether the number was valid. The GUI will have the following appearance:



To validate the credit card number, we'll place a listener on the Verify CC Number button. The code for the listener's `actionPerformed` method will need to read the text from the text field, decide whether the number is valid, and use that information to set the text of the label. Since the listener involves interaction between components, we'll make it object-oriented and make it act as its own listener.

As usual, let's deal with components and layout first before we worry about event handling. The frame can use a `FlowLayout` that wraps the text label to a second line.

The initial version of the program does not respond to events. As with the previous `BmiGui3` example, we'll make the GUI act as its own client program by incorporating a `main` method. Here is the initial version of the program:

```java
// Presents a GUI to verify credit card numbers.
// Initial version without event handling.

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

public class CreditCardGUI1 {
    public static void main(String[] args) {
        CreditCardGUI1 gui = new CreditCardGUI1();
    }

    // fields
    private JFrame frame;
    private JTextField numberField;
    private JLabel validLabel;
    private JButton verifyButton;

    // creates components, does layout, shows window onscreen
    public CreditCardGUI1() {
        numberField = new JTextField(16);
        validLabel = new JLabel("not yet verified");
        verifyButton = new JButton("Verify CC Number");

        frame = new JFrame("Credit card number verifier");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(350, 100));
        frame.setLayout(new FlowLayout());
        frame.add(numberField);
        frame.add(verifyButton);
        frame.add(validLabel);
        frame.setVisible(true);
    }
}
```

Now let's write a second version of the GUI that listens for action events on the Verify CC Number button. The code to validate credit card numbers is complex enough to be its own method. The code will loop over each digit, starting from the extreme right, doubling the digits at even indexes and splitting doubled numbers of 10 or larger into separate digits before adding them to the total. We can achieve this algorithm by adding the `digit / 10` and the `digit % 10` to the sum. We can actually add these same two values to the sum even if the number does not exceed 10, because for single-digit numbers, `digit / 10` is `0` and `digit % 10` is the digit itself. Here is the code that implements the validation:

```
// returns whether the given string can be a valid Visa
// card number according to the Luhn checksum algorithm
public boolean isValidCC(String text) {
    int sum = 0;
    for (int i = text.length() – 1; i >= 0; i--) {
        int digit = Integer.parseInt(text.substring(i, i + 1));
        if (i % 2 == 0) { // double even digits
            digit *= 2;
        }
        sum += (digit / 10) + (digit % 10);
    }
    // valid numbers add up to a multiple of 10
    return sum % 10 == 0 && text.startsWith("4");
}
```

Now that we have a method to tell us whether a given string represents a valid Visa card number, we can write the code to listen for events. First, we'll modify our class header:

```
public class CreditCardGUI2 implements ActionListener {
```

Next, we need to write the `actionPerformed` method. The method is short and simple, because the `isValidCC` method does the bulk of the work:

```
// Sets label's text to show whether CC number is valid.
public void actionPerformed(ActionEvent event) {
    String text = numberField.getText();
    if (isValidCC(text)) {
        validLabel.setText("Valid number!");
    } else {
        validLabel.setText("Invalid number.");
    }
}
```

Here is the complete version of the program, which includes all the components we just developed:

```
// Presents a GUI to verify credit card numbers.
// Final version with event handling.

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

public class CreditCardGUI2 implements ActionListener {
    public static void main(String[] args) {
        CreditCardGUI2 gui = new CreditCardGUI2();
    }

    // fields
    private JFrame frame;
    private JTextField numberField;
    private JLabel validLabel;
    private JButton verifyButton;

    // creates components, does layout, shows window onscreen
    public CreditCardGUI2() {
        numberField = new JTextField(16);
        validLabel = new JLabel("not yet verified");
        verifyButton = new JButton("Verify CC Number");

        // event listeners
        verifyButton.addActionListener(this);

        frame = new JFrame("Credit card number verifier");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(350, 100));
        frame.setLayout(new FlowLayout());
        frame.add(numberField);
        frame.add(verifyButton);
        frame.add(validLabel);
        frame.setVisible(true);
    }

    // Returns whether the given string is a valid Visa
    // card number according to the Luhn checksum algorithm.
    public boolean isValidCC(String text) {
        int sum = 0;
        for (int i = text.length() – 1; i >= 0; i--) {
            int digit = Integer.parseInt(
                    text.substring(i, i + 1));
            if (i % 2 == 0) { // double even digits
                digit *= 2;
            }
            sum += (digit / 10) + (digit % 10);
        }

        // valid numbers add up to a multiple of 10
        return sum % 10 == 0 && text.startsWith("4");
    }

    // Sets label's text to show whether CC number is valid.
    public void actionPerformed(ActionEvent event) {
        String text = numberField.getText();
        if (isValidCC(text)) {
            validLabel.setText("Valid number!");
        } else {
            validLabel.setText("Invalid number.");
        }
    }
}
```

14.4 Additional Components and Events

VideoNote

The GUIs we have developed so far use only a few components (buttons, text fields, and labels) and can respond to only one kind of event (action events). In this section, we'll look at some other useful components and events that GUI programs can use and to which they can respond.

Text Areas, Scrollbars, and Fonts

Text fields are useful for single-line text input, but they don't work well when the user wants to type a larger or more complex message. Fortunately, there is another kind of component called a *text area* that represents a text input box that can accept multiple lines. Text areas are represented by `JTextArea` objects. You can construct a `JTextArea` by passing the number of rows and columns (i.e., the number of lines and the number of letters in each line):

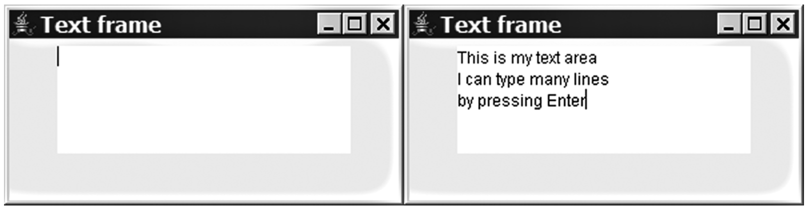
```
JTextArea area = new JTextArea(5, 20);
```

The preceding line of code creates a text area that is 5 lines tall and 20 letters wide:

The following program constructs a frame and adds a text area to it:

```
// Demonstrates the JTextArea component.

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

public class TextFrame {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new FlowLayout());
        frame.setSize(new Dimension(300, 150));
        frame.setTitle("Text frame");

        JTextArea area = new JTextArea(5, 20);
        frame.add(area);

        frame.setVisible(true);
    }
}
```

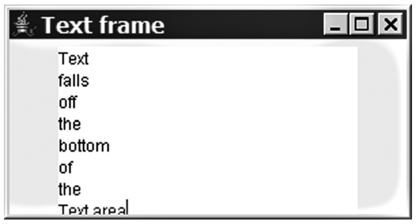
The program produces the following graphical output (shown both before and after the user types some text in the text area):



This is my text area
I can type many lines
by pressing Enter

In a larger GUI example, an event listener might examine the text that is written in a text area by calling its `getText` method or set new text in the text area by calling its `setText` method.

Currently, when the user types too much text to fit in the text area, the text disappears off the bottom of the text box:

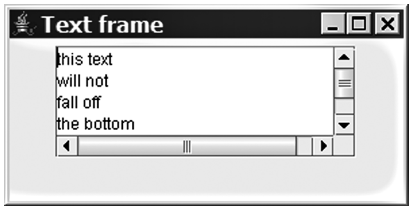


Text
falls
off
the
bottom
of
the
Text area

To fix this problem, we can make the text area scrollable by adding familiar navigation components called *scrollbars* to it. Scrollbars are represented by instances of a special container component called a `JScrollPane`. To make a component scrollable, create a `JScrollPane`, add the component to the scroll pane, and then add the scroll pane to the overall frame. You construct a `JScrollPane` object by passing the relevant component as a parameter:

```
// use scrollbars on this text area
frame.add(new JScrollPane(area));
```

We add the `JScrollPane` instead of the `JTextArea` to the frame, and the scroll pane creates scrollbars along the right and bottom edges of the text area when the text that the user enters is too large to display:

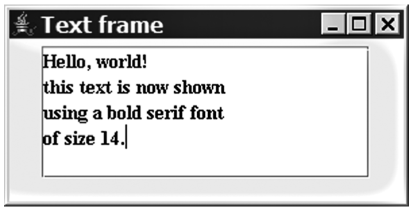


this text
will not
fall off
the bottom

The appearance of onscreen text is determined by *fonts,* which are sets of descriptions for how to display text characters. If you don't like the default font of the text area, remember that every component has a font property that you can set. As we discussed previously in Supplement 3G, fonts are represented by `Font` objects. A `Font` object can be constructed by passing its name, style (such as bold or italic), and size in pixels. For example, the following line of code tells the text area to use a size 14 serifed font:

```
area.setFont(new Font("Serif", Font.BOLD, 14));
```

The text area now has the following appearance:



Hello, world!
this text is now shown
using a bold serif font
of size 14.

See Supplement 3G for a more lengthy discussion of `Font` objects.

Icons

Many Java components, including buttons and labels, have an icon property that can be set to place an image on the component. This property is of type `Icon`.

The `getIcon` and `setIcon` methods use values that implement the `Icon` interface. The easiest way to get an object that implements `Icon` is to create an object of type `ImageIcon`. The `ImageIcon` constructor accepts a `String` parameter that represents the image file to load. The image is a file on your hard disk, such as a GIF, JPG, or PNG image. Place your image files in the same folder as your program files.

The following code places an icon on a button. The icon image comes from a file called `smiley.jpg` that has already been saved to the same folder as the code:

```
// create a smiley face icon for this button
JButton button = new JButton("Have a nice day");
button.setIcon(new ImageIcon("smiley.jpg"));
```

The screenshot that follows shows the appearance of the button when it is placed in a frame. It is enlarged to accommodate both its text and its new icon:

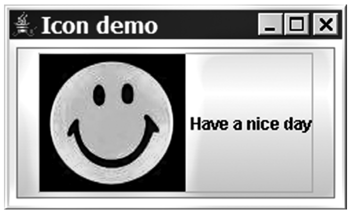


Table 14.6 Useful Methods of `BufferedImage` Objects

`public BufferedImage(int width, int height, int type)` Constructs an image buffer of the given size and type. Valid types are • `BufferedImage.TYPE_INT_ARGB`: An image with a transparent background. • `BufferedImage.TYPE_INT_RGB`: An image with a solid black background.
`public Graphics getGraphics()` Returns the pen for drawing on this image buffer.

Another way to create an icon besides loading a file containing the icon is to draw an icon yourself. You can create a blank image buffer by using a `BufferedImage` object and draw on it by using a `Graphics` pen, as we did with the `DrawingPanel` in Supplement 3G. (In fact, the `DrawingPanel` is implemented with an internal `BufferedImage`.) `BufferedImage` is a class in the `java.awt.image` package, so if you want to use it, you must import the package:

```
import java.awt.image.*; // for BufferedImage
```

The constructor and some useful methods of the `BufferedImage` class are shown in Table 14.6.

To use a `BufferedImage`, construct one of a particular size and type (we recommend `BufferedImage.TYPE_INT_ARGB`), then get the `Graphics` object from it using the `getGraphics` method. You can then issue standard drawing commands such as `drawRect` or `fillOval`. Once you're finished drawing on the `BufferedImage`, create a new `ImageIcon` object and pass the `BufferedImage` to the `ImageIcon` constructor. You can set this `ImageIcon` as the icon for an onscreen component by calling `setIcon` on that component and passing the `ImageIcon` as the parameter, as demonstrated in the following code:

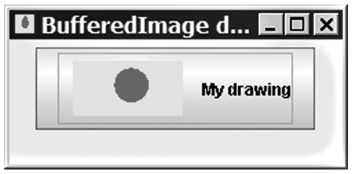
```
JButton button = new JButton();
button.setText("My drawing");

// create a shape image icon for this button
BufferedImage image = new BufferedImage(100, 100,
        BufferedImage.TYPE_INT_ARGB);
Graphics g = image.getGraphics();
g.setColor(Color.YELLOW);
g.fillRect(10, 20, 80, 70);
g.setColor(Color.RED);
g.fillOval(40, 50, 25, 25);

ImageIcon icon = new ImageIcon(image);
button.setIcon(icon);
```

You can also use a `BufferedImage` as the icon image for a frame by calling its `setIconImage` method:

```
frame.setIconImage(image);
```

Java's designers chose confusing names here, because the `setIconImage` method doesn't accept an `ImageIcon` as its parameter.

A screenshot of the button's appearance when it is placed into a frame follows. Notice that a smaller version of the icon also appears in the top-left corner of the window, because of the `setIconImage` call:



Mouse Events

So far, we've worked exclusively with `ActionListener` objects. When we want to listen to mouse clicks or movements, we use another type of listener called a `MouseInputListener`. The `MouseInputListener` interface resides in the `javax.swing.event` package, which you'll need to import:

```
import javax.swing.event.*; // for mouse events
```

The `MouseInputListener` interface methods for handling mouse input are listed in Table 14.7. There are quite a few methods, and you probably won't want to implement them all. Many programs handle only button presses or cursor movements. In these cases, you have to write empty versions of all the other `MouseInputListener` methods, because otherwise the class doesn't implement the interface properly and won't compile.

Table 14.7 The Methods of the `MouseInputListener` Interface

`public void mouseClicked(MouseEvent event)` Invoked when the mouse button has been clicked (pressed and released) on a component.
`public void mouseDragged(MouseEvent event)` Invoked when a mouse button is pressed on a component and then dragged.
`public void mouseEntered(MouseEvent event)` Invoked when the mouse enters a component.
`public void mouseExited(MouseEvent event)` Invoked when the mouse exits a component.
`public void mouseMoved(MouseEvent event)` Invoked when the mouse has been moved onto a component but no buttons have been pushed.
`public void mousePressed(MouseEvent event)` Invoked when a mouse button has been pressed on a component.
`public void mouseReleased(MouseEvent event)` Invoked when a mouse button has been released on a component.

To avoid the annoyance of writing empty bodies for the MouseInputListener methods that you don't want to use, you can use a class named MouseInputAdapter that implements default empty versions of all the methods from the MouseInputListener interface. You can extend MouseInputAdapter and override only the methods that correspond to the mouse event types you want to handle. We'll make all the mouse listeners we write subclasses of MouseInputAdapter so that we don't have to write any unwanted methods.

For example, let's write a mouse listener class that responds only when the mouse cursor moves over a component. To do this, we'll extend the MouseInputAdapter class and override only the mouseEntered method.

```
// Responds to a mouse event by showing a message dialog.

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

public class MovementListener extends MouseInputAdapter {
    public void mouseEntered(MouseEvent event) {
        JOptionPane.showMessageDialog(null, "Mouse entered!");
    }
}
```

Unfortunately, attaching this new listener to a component isn't so straightforward. The designers of the GUI component classes decided to separate the various types of mouse actions into two categories, mouse button clicks and mouse movements, and created two separate interfaces (MouseListener and MouseMotionListener) to represent these types of actions. Later, the MouseInputListener and MouseInputAdapter methods were added to merge the two listeners, but GUI components still need mouse listeners to be attached in two separate ways for them to work.

If we wish to hear about mouse enter, exit, press, release, and click events, we must call the addMouseListener method on the component. If we wish to hear about mouse move and drag events, we must call the addMouseMotionListener method on the component. If we want to hear about all the events, we can add the mouse input adapter in both ways. (This is what we'll do in the examples in this section.)

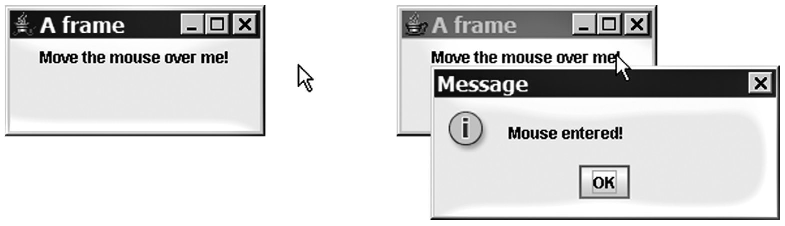
Table 14.8 Useful Methods of Components

`public void addMouseListener(MouseListener listener)` Attaches a listener to hear mouse enter, exit, press, release, and click events.
`public void addMouseMotionListener(MouseMotionListener listener)` Attaches a listener to hear mouse move and drag events.

Here is an entire program that uses our `MovementListener` to respond when the user moves the mouse over a label:

```java
// A GUI that listens to mouse movements over a label.

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

public class MouseGUI {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new FlowLayout());
        frame.setSize(new Dimension(200, 100));
        frame.setTitle("A frame");

        JLabel label = new JLabel();
        label.setText("Move the mouse over me!");
        frame.add(label);

        MovementListener mListener = new MovementListener();
        label.addMouseListener(mListener);
        label.addMouseMotionListener(mListener);

        frame.setVisible(true);
    }
}
```

The program produces the following graphical output, shown both before and after the user moves the mouse onto the `JLabel`:



When the `actionPerformed` method is dealing with action listeners, it accepts a parameter of type `ActionEvent` that represents the action that occurs. We didn't use this object for anything in our programs. However, the corresponding parameter in mouse listeners, which is of type `MouseEvent`, is very useful. Several handy pieces of information are stored in the `MouseEvent` parameter. Table 14.9 lists some of the methods of this parameter.

Table 14.9 Useful Methods of `MouseEvent` Objects

`public int getButton()`
Returns the number of the mouse button that was pressed or released (1 for the left button, 2 for the right button, and so on).

`public int getClickCount()`
Returns the number of times the user clicked the button. This method is useful for detecting double-clicks.

`public Point getPoint()`
Returns the (x, y) point where the mouse event occurred.

`public int getX()`
Returns the x-coordinate where the mouse event occurred.

`public int getY()`
Returns the y-coordinate where the mouse event occurred.

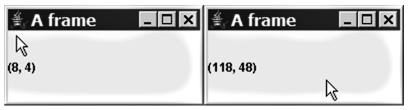
For example, the mouse listener that follows could be attached to any component, even the frame itself. It would make a message appear any time the user pressed the mouse button:

```java
// Responds to a mouse click by showing a message
// that indicates where the user clicked

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

public class ClickListener extends MouseInputAdapter {
    public void mousePressed(MouseEvent event) {
        JOptionPane.showMessageDialog(null,
                "Mouse pressed at position ("
                + event.getX() + ", " + event.getY() + ")");
    }
}
```

As another example, the program that follows uses a mouse listener to set a label's text to show the mouse's position as it moves over the label. Like the GUIs in our previous examples, this GUI is object-oriented and serves as its own listener:

```java
// A GUI that displays the position of the mouse over a label.

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

 public class MousePointGUI extends MouseInputAdapter {
    public static void main(String[] args) {
        MousePointGUI gui = new MousePointGUI();
    }

    // fields
    private JFrame frame;
    private JLabel label;

    // sets up the GUI, components, and events
    public MousePointGUI() {
        label = new JLabel();
        label.setText("Move the mouse over me!");

        // listen for mouse events
        label.addMouseListener(this);
        label.addMotionListener(this);

        frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(200, 100));
        frame.setTitle("A frame");
        frame.add(label);
        frame.setVisible(true);
    }

    // responds to mouse movement events
    public void mouseMoved(MouseEvent event) {
        label.setText("(" + event.getX() + ", " +
                          event.getY() + ")");
    }
}
```

The program produces the following graphical output, shown after the user moves the mouse onto a few different points on the label:



14.5 Two-Dimensional Graphics

In Supplement 3G, we introduced a graphical class called `DrawingPanel`. The `DrawingPanel` was simple enough that you did not need to learn a lot of details about graphical user interfaces. Now that we're starting to uncover those details, we can examine how to draw our own two-dimensional graphics manually. This will help you eventually understand the code for `DrawingPanel` and even reimplement it yourself.

Drawing onto Panels

Earlier in this chapter we discussed the `JPanel` component, which we used as an invisible container for laying out other components. Panels have one other important function: They serve as surfaces onto which we can draw. The `JPanel` object has a method called `paintComponent` that draws the panel on the screen. By default this method draws nothing, so the panel is transparent.

We can use inheritance to change the drawing behavior for a panel. If we want to make a panel onto which we can draw shapes, we can extend `JPanel` and override the `paintComponent` method. Here is the header of the `paintComponent` method we must override:

```
public void paintComponent(Graphics g)
```

Its parameter, `Graphics g`, should look familiar to you if you read Supplement 3G. `Graphics` is a class in Java's `java.awt` package that has methods for drawing shapes, lines, and images onto a surface. Think of the `Graphics` object as a pen and the panel as a sheet of paper.

There is one quirk to keep in mind when you are writing a `paintComponent` method. Remember that when you override a method, you replace the superclass method's previous functionality. We don't want to lose the behavior of `paintComponent` from `JPanel`, though, because it does important things for the panel; we just want to add additional behavior to it. Therefore, the first thing you should do when you override `paintComponent` is to call `super.paintComponent` and pass it your `Graphics` object `g` as its parameter. (If you don't, you can get strange ghosty afterimages when you drag or resize your window.)

Here's a small class that represents a panel with two rectangles drawn on it:

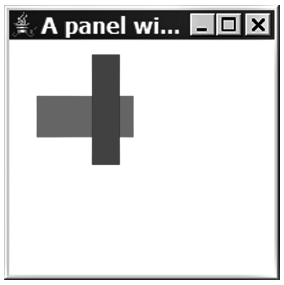
```
// A panel that draws two rectangles on its surface.

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

public class RectPanel extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g); // call JPanel's version
```

```

            g.setColor(Color.RED);
            g.fillRect(20, 40, 70, 30);
            g.setColor(Color.BLUE);
            g.fillRect(60, 10, 20, 80);
        }
}
```

We can now write a separate class for a GUI that incorporates this panel. In the GUI class, we create a `RectPanel` object and add it to the frame. The rectangle will appear on the screen with the two shapes drawn on top of it. Here's the example client code that uses our `RectPanel`:

```
// Demonstrates the RectPanel class by placing one into a GUI.

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

public class UseRectPanel {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 200);
        frame.setTitle("A panel with rectangles");

        RectPanel panel = new RectPanel();
        panel.setBackground(Color.WHITE);
        frame.add(panel);

        frame.setVisible(true);
    }
}
```

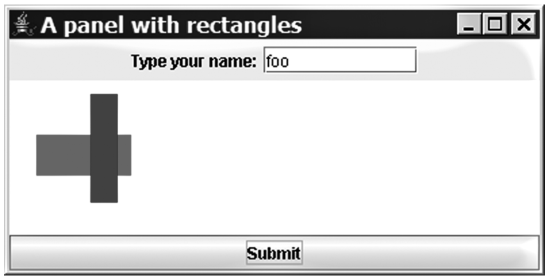
The program produces the following graphical output:



You might wonder why you can't just use a `DrawingPanel` rather than going to all this trouble. There are several reasons. One is that `DrawingPanel` isn't actually a standard part of Java, so you can't rely on it outside a classroom setting. Also, although it's a nice tool for simple drawing, you can't make a `DrawingPanel` part of a larger GUI with other components. The following modified code for the previous client program achieves a mixture of components that would be impossible to replicate with a `DrawingPanel`:

```
public static void main(String[] args) {
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(400, 200);
    frame.setTitle("A panel with rectangles");
    frame.setLayout(new BorderLayout());
    JPanel north = new JPanel(new FlowLayout());
    north.add(new JLabel("Type your name:"));
    north.add(new JTextField(10));
    frame.add(north, BorderLayout.NORTH);

    frame.add(new JButton("Submit"), BorderLayout.SOUTH);

    RectPanel panel = new RectPanel();
    panel.setBackground(Color.WHITE);
    frame.add(panel, BorderLayout.CENTER);

    frame.setVisible(true);
}
```

The program produces the following graphical output:



For a more complete description of two-dimensional graphical methods and objects, see Supplement 3G's discussion of the `Graphics` object.

Animation with Timers

A panel doesn't have to show a simple static drawing. You can use an object called a *timer,* implemented by an object of the `Timer` class, to animate a panel to show a moving picture. A timer is an object that, once started, fires an action event at regular intervals. You supply the timer with an `ActionListener` to use and a delay time in milliseconds between action events, and it does the rest. You need to include two objects to create an animation:

- An `ActionListener` object that updates the way the panel draws itself
- A `Timer` object to invoke the `ActionListener` object at regular intervals, causing the panel to animate

Let's modify our `RectPanel` class from the previous section to move its rectangles on the panel. To make them move, we must store their positions as fields in the `JPanel` object. We'll repeatedly change the fields' values and redraw the rectangles as the program is running, which will make it look like they're moving. We'll store the desired change in `x` and change in `y` as fields called `dx` and `dy`, respectively. Here's the code for our animated `RectPanel`:

```
// A panel that draws animated rectangles on its surface.
// Initial version—will change in the following section.

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

public class AnimatedRectPanel1 extends JPanel {
    private Point p1; // location of first rectangle
    private Point p2; // location of second rectangle
    private int dx; // amount by which to move horizontally
    private int dy; // amount by which to move vertically

    public AnimatedRectPanel1() {
        p1 = new Point(20, 40);
        p2 = new Point(60, 10);
        dx = 5;
        dy = 5;
    }

    // draws the panel on the screen
    public void paintComponent(Graphics g) {
        super.paintComponent(g); // call JPanel's version
        g.setColor(Color.RED);
        g.fillRect(p1.x, p1.y, 70, 30);
        g.setColor(Color.BLUE);
        g.fillRect(p2.x, p2.y, 20, 80);
    }
}
```

The next step is to create an action listener that updates the panel each time it is invoked. We'll make the `AnimatedRectPanel` its own listener. You might be tempted to put a `for` loop in the `actionPerformed` method, but this isn't the way to do animation. Instead, `actionPerformed` moves the rectangles by a small amount, and the timer invokes the listener repeatedly at regular intervals to redraw and animate the panel.

The first rectangle will move horizontally, bouncing off the panel's left and right edges. The second rectangle will move vertically, bouncing off the panel's top and bottom edges. The most straightforward thing to do in our `ActionListener` is to adjust the two points' coordinates:

```
// initial version—doesn't turn around
public void actionPerformed(ActionEvent event) {
    p1.x += dx;
    p2.y += dy;
}
```

In this version, the rectangles don't bounce off the panel's edges. To add that feature, we need to check to see whether a rectangle has hit either edge. We'll know that the first rectangle has hit the left edge when its *x*-coordinate drops to `0`, and we'll know that it has hit the right edge when the value of its *x*-coordinate plus its width exceeds the panel's width. The calculation is similar for the second rectangle, except that we'll look at the *y*-coordinate and the heights of the rectangle and the panel.

There's one last thing to mention about animating `JPanels`. When we change the way the panel should be drawn, we have to call a method called `repaint` to instruct the panel to draw itself again. If we don't call `repaint`, we won't see any change on the screen. (Forgetting to call `repaint` is a common bug.) Here's the code that implements the bouncing behavior and redraws the panel:

```
// called at regular intervals by timer to redraw the panel
public void actionPerformed(ActionEvent event) {
    p1.x += dx;
    p2.y += dy;
    if (p1.x <= 0 || p1.x + 70 >= getWidth()) {
        dx = -dx; // rectangle 1 has hit left/right edge
    }
    if (p2.y <= 0 || p2.y + 80 >= getHeight()) {
        dy = -dy; // rectangle 2 has hit top/bottom edge
    }
    repaint(); // instruct the panel to redraw itself
}
```

Now that our `ActionListener` code is complete, we have to create and start the timer that invokes the listener. Let's add code for the timer to the constructor for the `AnimatedRectPanel`:

```
// set up timer to animate rectangles every 100 ms
Timer time = new Timer(100, this);
time.start();
```

Here's the complete second version of the `AnimatedRectPanel` program that incorporates all of these features:

```
// A panel that draws animated rectangles on its surface.

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

public class AnimatedRectPanel2 extends JPanel
        implements ActionListener {
    private Point p1; // location of first rectangle
    private Point p2; // location of second rectangle
    private int dx; // amount by which to move horizontally
    private int dy; // amount by which to move vertically

    public AnimatedRectPanel2() {
        p1 = new Point(20, 40);
        p2 = new Point(60, 10);
        dx = 5;
        dy = 5;

        // set up timer to animate rectangles every 100 ms
        Timer time = new Timer(100, this);
        time.start();
    }

    // draws two rectangles on this panel on the screen
    public void paintComponent(Graphics g) {
        super.paintComponent(g); // call JPanel's version
        g.setColor(Color.RED);
        g.fillRect(p1.x, p1.y, 70, 30);
        g.setColor(Color.BLUE);
        g.fillRect(p2.x, p2.y, 20, 80);
    }

    // called at regular intervals by timer to redraw the panel
    public void actionPerformed(ActionEvent event) {
        p1.x += dx;
        p2.y += dy;
        if (p1.x <= 0 || p1.x + 70 >= getWidth()) {
            dx = -dx; // rectangle 1 has hit left/right edge
        }
        if (p2.y <= 0 || p2.y + 80 >= getHeight()) {
            dy = -dy; // rectangle 2 has hit top/bottom edge
        }

        repaint(); // instruct the panel to redraw itself
    }
}
```

Here's the graphical window that is produced as output, which animates the rectangles as we intended:

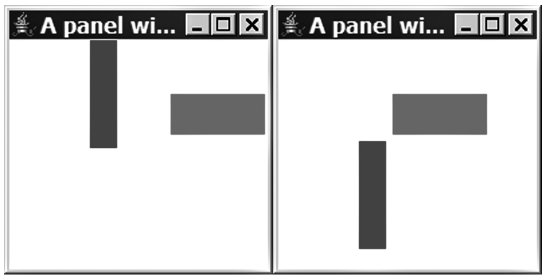


Table 14.10 lists several methods of the `Timer` class.

Table 14.10 Useful Methods of `Timer` Objects

`public Timer(int msDelay, ActionListener listener)` Creates a `Timer` that activates the given `ActionListener` every `msDelay` milliseconds.
`public void start()` Tells the timer to start ticking and to activate its `ActionListener`.
`public void stop()` Tells the timer to stop ticking and not to activate its `ActionListener`.

14.6 Case Study: Implementing `DrawingPanel`

In previous chapters, we used the `DrawingPanel` class to create simple two-dimensional graphics. In this final section, we'll analyze `DrawingPanel` to determine how it works and implement a basic version of the class.

Initial Version without Events

When we created the `DrawingPanel` class, we had two major design goals. The first was to provide an easy way for client code to access a `Graphics` object. The second was to provide a simple interface so that the client did not need to know about calling `paintComponent`, `repaint`, and other complex methods.

The graphics drawn by the client onto a `DrawingPanel` are actually drawn onto a `BufferedImage` object stored inside it. The `DrawingPanel` declares a `BufferedImage` as a field and initializes it in its constructor. When the client calls `getGraphics` on the `DrawingPanel`, the program returns a reference to the buffered image's graphics pen. To place the buffered image onto the screen, we set it as the icon for a `JLabel`.

The start of the `DrawingPanel` class looks a lot like the code for the GUIs that we developed in this chapter. It begins by declaring various graphical components as fields. The fields are the overall window frame, the `Graphics` object for the onscreen buffered image, and a panel to hold the image:

```
public class DrawingPanel {
    private JFrame frame;  // overall window frame
    private JPanel panel;  // drawing surface
    private Graphics g;    // drawing pen
    ...
}
```

The constructor of the `DrawingPanel` accepts two parameters that represent the panel's width and height. It initializes the fields and constructs the `BufferedImage` to serve as the persistent buffer where shapes and lines can be drawn. The class also adds a few methods for the client, such as `getGraphics`, `setBackground`, and `setVisible`. The following lines of code form an initial version of its complete code:

```
// A simple interface for drawing persistent images.
// Initial version without events.

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

public class DrawingPanel {
```

```java
    private JFrame frame; // overall window frame
    private JPanel panel; // drawing surface
    private Graphics g;   // drawing pen

    // constructs a drawing panel of given size
    public DrawingPanel(int width, int height) {
        // sets up the empty image onto which we will draw
        BufferedImage image = new BufferedImage(width, height,
                BufferedImage.TYPE_INT_ARGB);
        g = image.getGraphics();
        g.setColor(Color.BLACK);

        // enclose the image in a label inside a panel
        JLabel label = new JLabel();
        label.setIcon(new ImageIcon(image));
        panel = new JPanel(new FlowLayout());
        panel.setBackground(Color.WHITE);
        panel.setPreferredSize(new Dimension(width, height));
        panel.add(label);

        // set up the JFrame
        frame = new JFrame("Drawing Panel");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setResizable(false);
        frame.add(panel);
        frame.pack();
        frame.setVisible(true);
    }

    // obtains the Graphics object to draw on the panel
    public Graphics getGraphics() {
        return g;
    }

    // sets the background color of the drawing panel
    public void setBackground(Color c) {
        panel.setBackground(c);
    }

    // shows or hides the drawing panel on the screen
    public void setVisible(boolean visible) {
        frame.setVisible(visible);
    }
}
```

The code places the buffered image onto a label as its icon, then stores the buffered image into a panel that is placed inside the frame. We use this intermediate panel to ensure that the background colors are set properly.

Second Version with Events

A more sophisticated version of `DrawingPanel` will also act as an event listener for mouse events, so that as the user drags the mouse over the panel, the program will display the cursor's (*x, y*) position on a status bar at the bottom for debugging. To achieve this, the `DrawingPanel` can be modified to act as a mouse listener and attach itself to listen to its central panel. Whenever the mouse moves, the mouse listener will receive the event and will update the status bar text to show the mouse position.

To implement these capabilities, we'll begin by changing our class header to the following header:

```
public class DrawingPanel extends MouseInputAdapter {
```

We'll then add a new field called `statusBar`, of type `JLabel`, to the program and place it in the south region of the frame. Initially the status bar's text will be empty, but if the mouse moves, its listener will change the text to reflect the cursor's position. The following code implements the mouse listener's `mouseMoved` method:

```
// draws status bar text when mouse moves
public void mouseMoved(MouseEvent e) {
    statusBar.setText("(" + e.getX() + ", " + e.getY() + ")");
}
```

We'll also add the following code to the constructor to attach the `DrawingPanel` as a listener to its inner panel:

```
// attach listener to observe mouse movement
panel.addMouseListener(this);
panel.addMouseMotionListener(this);
```

There's another kind of event that we should handle in the `DrawingPanel`. When the client code using `DrawingPanel` takes a long time (such as when it uses console input to guide what shapes will be drawn), we may need to repaint the `DrawingPanel` periodically to reflect any new shapes that the client has drawn. To keep things simple, we don't want to force the client to call `repaint` itself.

To force the `DrawingPanel` to refresh at a given interval, we'll use a `Timer` to periodically call `repaint`. Since a `Timer` requires an action listener as a parameter, we'll make the `DrawingPanel` implement the `ActionListener` interface. Its increasingly long header will now look like this:

```
public class DrawingPanel extends MouseInputAdapter
        implements ActionListener {
```

The `actionPerformed` method simply needs to call `repaint` on the main onscreen panel:

```
// used for timer that repeatedly repaints screen
public void actionPerformed(ActionEvent e) {
    panel.repaint();
}
```

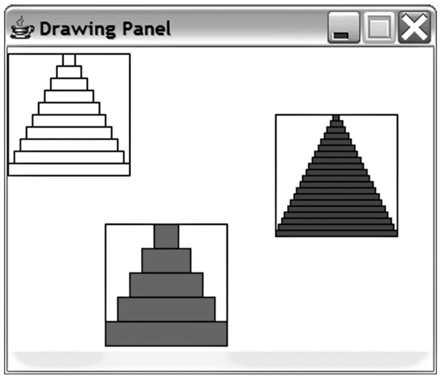
Lastly, we'll add the following lines to the `DrawingPanel`'s constructor to create and start the timer:

```
// start a repaint timer to refresh the screen
Timer timer = new Timer(250, this);
timer.start();
```

Here is the complete code for the second version of the `DrawingPanel` class, which includes all the components we developed:

```
// A simple interface for drawing persistent images.
// Final version with events.

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

public class DrawingPanel extends MouseInputAdapter
        implements ActionListener {
    private JFrame frame; // overall window frame
    private JPanel panel; // drawing surface
    private JLabel statusBar; // status bar
    private Graphics g; // drawing pen

    // constructs a drawing panel of given size
    public DrawingPanel(int width, int height) {
        // set up the empty image onto which we will draw
        BufferedImage image = new BufferedImage(width, height,
                BufferedImage.TYPE_INT_ARGB);
        g = image.getGraphics();
        g.setColor(Color.BLACK);

        // encloses the image in a label inside a panel
        JLabel label = new JLabel();
        label.setIcon(new ImageIcon(image));
        panel = new JPanel(new FlowLayout());
        panel.setBackground(Color.WHITE);
        panel.setPreferredSize(new Dimension(width, height));
        panel.add(label);

        // the status bar that shows the mouse position
        statusBar = new JLabel(" ");

        // attaches listener to observe mouse movement
        panel.addMouseListener(this);
        panel.addMouseMotionListener(this);

        // sets up the JFrame
        frame = new JFrame("Drawing Panel");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setResizable(false);
        frame.setLayout(new BorderLayout());
        frame.add(panel, BorderLayout.CENTER);
        frame.add(statusBar, BorderLayout.SOUTH);
        frame.pack();
        frame.setVisible(true);

        // starts a repaint timer to refresh the screen
        Timer timer = new Timer(250, this);
        timer.start();
    }

    // obtains the Graphics object to draw on the panel
    public Graphics getGraphics() {
        return g;
    }

    // sets the background color of the drawing panel
    public void setBackground(Color c) {
        panel.setBackground(c);
    }

    // shows or hides the drawing panel on the screen
    public void setVisible(boolean visible) {
        frame.setVisible(visible);
    }

    // used for timer that repeatedly repaints screen
    public void actionPerformed(ActionEvent e) {
        panel.repaint();
    }

    // draws status bar text when mouse moves
    public void mouseMoved(MouseEvent e) {
        statusBar.setText("(" + e.getX() + ", " +
                                e.getY() + ")");
    }
}
```

This version of the code can be used to draw any of the example programs from Supplement 3G. For example, the following screenshot shows the output when the Pyramids case study from Supplement 3G is run using the `DrawingPanel` we have just implemented:



Chapter Summary

A graphical user interface (GUI) has a window frame that contains buttons, text input fields, and other onscreen components.

The `JOptionPane` class is a simple GUI class that can be used to display messages and prompt a user to input values, enabling graphical output and input.

Some of the most common graphical components are buttons (`JButton` objects), text input fields (`JTextField` objects), and text labels (`JLabel` objects).

All graphical components in Java belong to a common inheritance hierarchy, so they share a common set of methods that can be used to get and set properties such as background color, size, and font.

Components are positioned in a frame or container by objects called layout managers, such as `BorderLayout`, `FlowLayout`, and `GridLayout`. The features of the layout managers can be combined by nesting them in different containers to form a composite layout.

Java generates special objects called events when the user interacts with onscreen graphical components. To write an interactive GUI, you must write responses to these events.

The most common type of event is an `ActionEvent`, which you can handle by writing a class that implements the `ActionListener` interface. You can also respond to `MouseEvents` by writing a class that extends the `MouseInputAdapter` class.

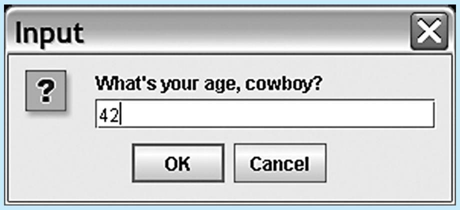
To draw lines and shapes, you must write a class that extends `JPanel` and write a method called `paintComponent`.

GUIs can be animated using objects called timers that cause events to fire on an action listener at regular intervals.

Self-Check Problems

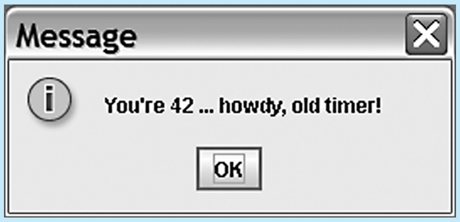
Section 14.1: GUI Basics

1. What are the three new packages introduced in this section that you must import when you write graphical programs? Write the `import` statements that are necessary in a program that uses these classes.
2. Write Java code to pop up an option pane asking the user for his or her age. If the user types a number less than 40, respond with a message box saying that he or she is young. Otherwise, tease the user for being old. Here is an example:


Input

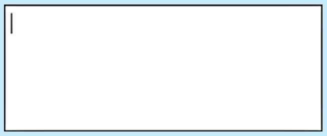
What's your age, cowboy?

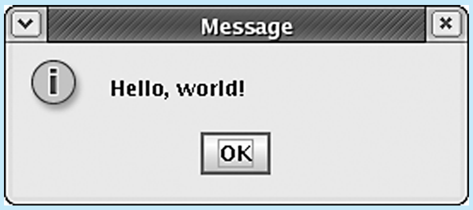
42

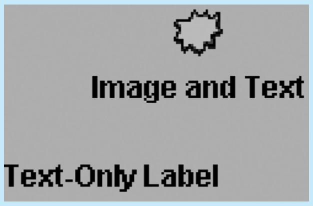

Message

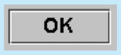
You're 42 ... howdy, old timer!

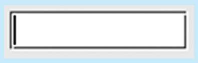
3. What is a component? How is a frame different from other components?
4. Name two properties of frames. Give an example piece of code that creates a new frame and sets these two properties to have new values of your choice.
5. Identify the Java class used to represent each of the following graphical components.

a.


b. 

c. 

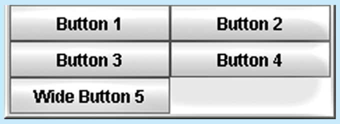
d. 

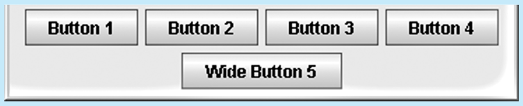
e. 

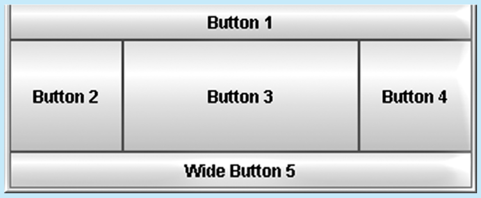
6. Write a piece of Java code that creates two buttons, one with a green background and the text "Click me" and the other with a yellow background and the text "Do not touch!"

Section 14.2: Laying Out Components

7. Identify the layout manager that would produce each of the following onscreen appearances:

a. 

b. 

c. 

8. Write the code that would produce each of the layouts pictured in problem 7, given the following variable declarations:

```
JButton b1 = new JButton("Button 1");
JButton b2 = new JButton("Button 2");
JButton b3 = new JButton("Button 3");
JButton b4 = new JButton("Button 4");
JButton b5 = new JButton("Wide Button 5");
```

Section 14.3: Interaction between Components

9. What is an event? What is a listener? What interface does Java use to handle standard action events?

10. Describe the code that you must write for your program to handle an event. What class(es) and method(s) do you have to write? What messages must be sent to the component in question (such as the button to be clicked)?

11. Write an `ActionListener` to be attached to a button so that whenever the user clicks that button, an option pane will pop up that says, "Greetings, Earthling!"

12. Why should complex GUIs that handle events be written in an object-oriented way?

Section 14.4: Additional Components and Events

13. What classes and interfaces does Java use to implement mouse listeners? In what ways does the process of handling mouse events differ from that of handling action events?

14. Modify the `MousePointGUI` program's mouse listener so that it sets the frame's background color to blue if the mouse pointer is in the upper half of the frame and red if it is in the lower half of the frame. (*Hint*: Use the label's `getHeight` and `setForeground` methods in your solution.)

Section 14.5: Two-dimensional Graphics

15. What class should be extended in order to draw two-dimensional graphics? What method should be overwritten to do the drawing?

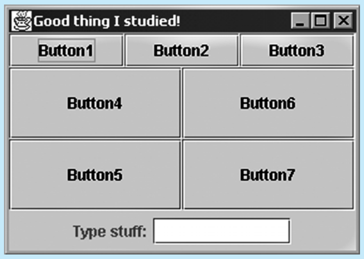
16. Write a panel class that paints a red circle on itself when it is drawn on the screen.

17. What is a timer? How are timers used with panels in programs that draw graphics?

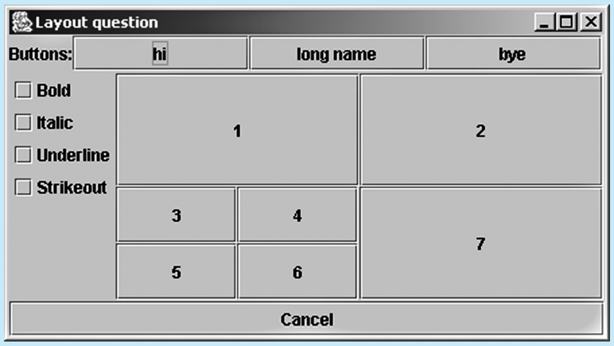
18. Modify your red circle panel from problem 16 so that the color changes to blue and back again, alternating every second. Use a timer to "animate" the color changes of the panel.

Exercises

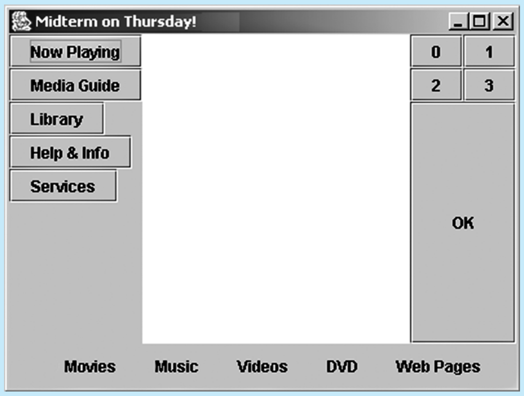
1. Write a complete Java program that creates a window titled "Good thing I studied!" that is 285 by 200 pixels.



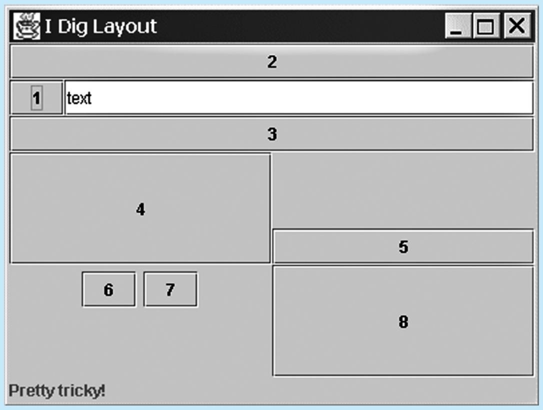
2. Write a complete Java program that creates a window titled `"Layout question"` that is 420 by 250 pixels.



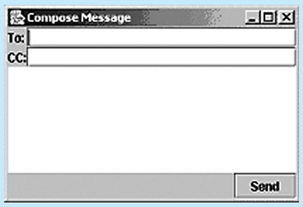
3. Write a complete Java program that creates a window titled “Midterm on Thursday!” that is 400 by 300 pixels.



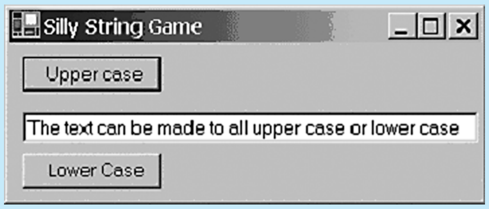
4. Write a complete Java program that creates a window titled “I Dig Layout” that is 400 by 300 pixels.



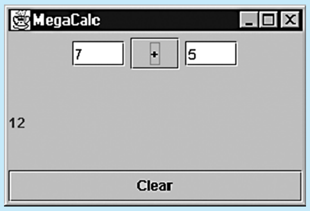
5. Write a complete Java program that creates a window titled "Compose Message" that is 300 by 200 pixels.



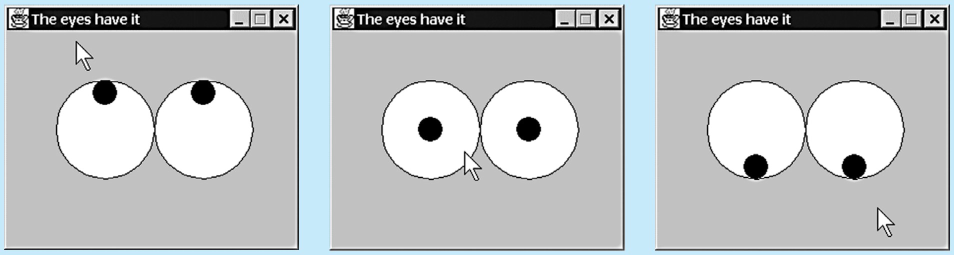
6. Write a complete Java program that creates a window titled "Silly String Game" that is 300 by 100 pixels. The user can type text into the text field. When the user presses a button labeled "Upper case", the text is capitalized. When the user presses a button labeled "Lower Case", the text is made lowercase.



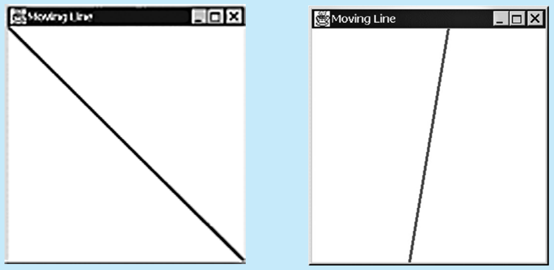
7. Write a complete Java program that creates a window titled "MegaCalc" that is 400 by 300 pixels. In the window, there are two text fields in which the user types the operands to a binary + operation. When the user presses the + button, the program treats the text in the two fields as two integers, adds them together, and displays the resulting integer on the screen as the text of a result label.



8. Write a panel class called `EyePanel` (and any other classes you need, such as event listeners) to implement a panel that draws two eyes that look up, to the center, or down depending on whether the mouse cursor is above, inside, or below the eyes. The following three screenshots show the eyes in each of the three positions:

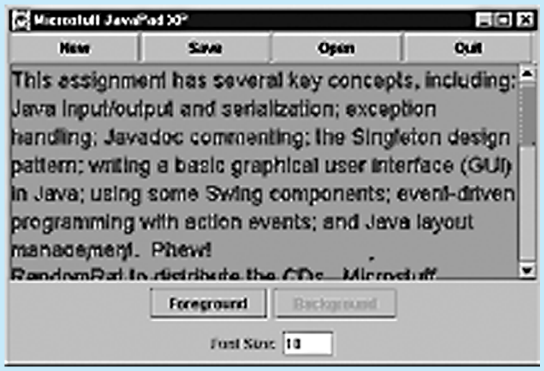


9. Write a complete Java program that animates a moving line, as depicted in the following screenshots:



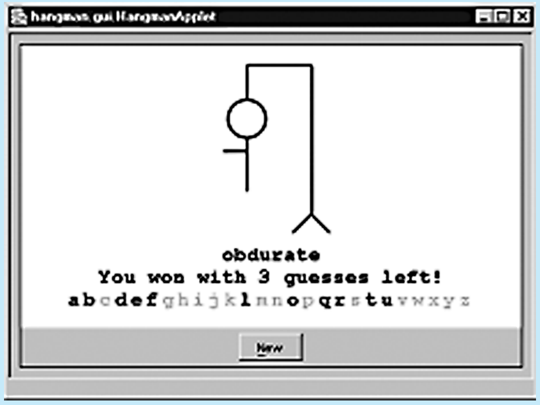
Programming Projects

1. Write a GUI program that represents a simple Notepad clone. The program should support a New feature that clears the text buffer, a Save feature that writes to a file using a `PrintStream`, a Load feature that reads from a file using a `Scanner`, and a Quit feature that exits the program. You can also add support for changing the font size and background/foreground colors. Your GUI might look something like the following.



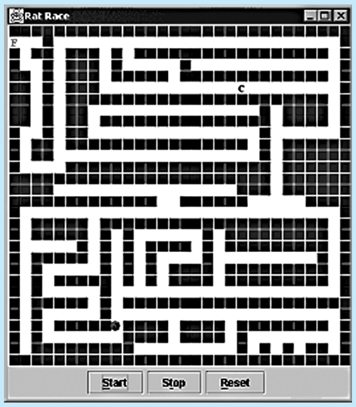
This assignment has several key concepts, including: Java input/output and serialization; exception handling; Javadoc commenting; the Singleton design pattern; writing a basic graphical user interface (GUI) in Java; using some Swing components; event-driven programming with action events; and Java layout management. Phew!
RandomPal to distribute the CDs - Microstuff

2. Write a graphical Hangman game program. The program should draw the stick-man step by step, drawing a new part each time the user guesses a wrong letter. Display the letters that have and have not yet been guessed. Pick the game word that the user is trying to guess from a list you create yourself. Your GUI might look like the following:



```
obdurate
You won with 3 guesses left!
abcdefghijklmnopqrstuvwxyz
```

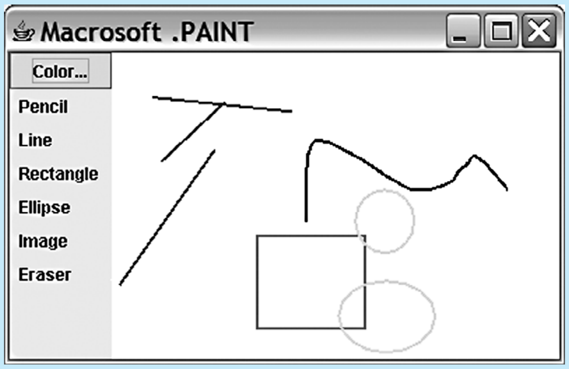
3. Write a program that draws a graphical maze and an inheritance hierarchy of rats that escape the maze in different ways. (For example, your program might include a `RandomRat` that moves in a random direction, a `JumpingRat` that moves straight but can jump over walls, or a `WallHuggerRat` that sticks close to the wall on its right until it finds the exit.) Consider adding to the maze a cat that can eat any rat it touches. Your maze might look something like this:



M14_REGE1813_02_SE_C14.qxd 2/10/10 3:44 PM Page 885

EQA

4. Write a painting program that allows the user to paint shapes and lines in many different colors. Here's a sample display of the program:


Macrosoft .PAINT
Color...
Pencil
Line
Rectangle
Ellipse
Image
Eraser