SESSION 6: Programming Languages for Objects

- The Basic GUI Application
- Graphics and Painting
- Mouse Events
- Timers, KeyEvents, and State Machines
- Basic Components
- Basic Layout
- Menus and Dialogs

Introduction to GUI Programming

COMPUTER USERS TODAY EXPECT to interact with their computers using a graphical user interface (GUI). Java can be used to write GUI programs ranging from simple applets which run on a Web page to sophisticated stand-alone applications.

GUI programs differ from traditional "straight-through" programs that you have encountered in the first few chapters of this book. One big difference is that GUI programs are event-driven. That is, user actions such as clicking on a button or pressing a key on the keyboard generate events, and the program must respond to these events as they occur.

Event-driven programming builds on all the skills you have learned in the first five chapters of this text. You need to be able to write the methods that respond to events. Inside those methods, you are doing the kind of programming-in-the-small that was covered in Chapter 2 and Chapter 3. And of course, objects are everywhere in GUI programming. Events are objects. Colors and fonts are objects. GUI components such as buttons and menus are objects. Events are handled by instance methods contained in objects. In Java, GUI programming is object-oriented programming.

This chapter covers the basics of GUI programming

The Basic GUI Application

THE COMMAND-LINE PROGRAMS that you have learned how to program would seem very alien to most computer users. The style of interaction where the user and the computer take turns typing strings of text seems like something out of the early days of computing, although it was only in the mid 1980s that home computers with graphical user interfaces started to become available. Today, most people interact with their computers exclusively through a GUI. A GUI program offers a much richer type of user interface, where the user uses a mouse and keyboard to interact with GUI components such as windows, menus, buttons, check boxes, text input boxes, scroll bars, and so on.

A GUI program still has a `main()` subroutine, but in general, that main routine just creates one or more GUI components and displays them on the computer screen. Once the GUI components have been created, they follow their **own** programming -- programming that tells them how to draw themselves on the screen and how to respond to events such as being clicked on by the user.

A GUI program doesn't have to be immensely complex. We can, for example, write a very simple GUI "Hello World" program that says "Hello" to the user, but does it by opening a window where the greeting is displayed:

```
import javax.swing.JOptionPane;

public class HelloWorldGUI1 {

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

}
```

When this program is run, a window appears on the screen that contains the message "Hello World!". The window also contains an "OK" button for the user to click after reading the message. When the user clicks this button, the window closes and the program ends. This program can be placed in a file named *HelloWorldGUI1.java*, compiled, and run using the `java` command on the command line just like any other Java program.

Now, this program is already doing some pretty fancy stuff. It creates a window, it draws the contents of that window, and it handles the event that is generated when the user clicks the button. The reason the program was so easy to write is that all the work is done by `showMessageDialog()`, a `static` method in the built-in class *JOptionPane*. (Note that the source code "imports" the class `javax.swing.JOptionPane` to make it possible to refer to the *JOptionPane* class using its simple name. See Subsection 4.5.3 for information about importing classes from Java's standard packages.)

If you want to display a message to the user in a GUI program, this is a good way to do it: Just use a standard class that already knows how to do the work! And in fact, *JOptionPane* is regularly used for just this purpose (but as part of a larger program, usually). Of course, if you want to do anything serious in a GUI program, there is a lot more to learn. To give you an idea of the types of things that are involved, we'll look at a short GUI program that does the same things as the previous program -- open a window containing a message and an OK button, and respond to a click on the button by ending the program -- but does it all by hand instead of by using the built-in *JOptionPane* class. Mind you, this is **not** a good way to write the program, but it will illustrate some important aspects of GUI programming in Java.

Here is the source code for the program. You are not expected to understand it yet. I will explain how it works below, but it will take the rest of the chapter before you will really understand completely. In this section, you will just get a brief overview of GUI programming.

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

public class HelloWorldGUI2 {

   private static class HelloWorldDisplay extends JPanel {
      public void paintComponent(Graphics g) {
         super.paintComponent(g);
         g.drawString( "Hello World!", 20, 30 );
      }
   }

   private static class ButtonHandler implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         System.exit(0);
      }
   }

   public static void main(String[] args) {

      HelloWorldDisplay displayPanel = new HelloWorldDisplay();
      JButton okButton = new JButton("OK");
      ButtonHandler listener = new ButtonHandler();
      okButton.addActionListener(listener);

      JPanel content = new JPanel();
      content.setLayout(new BorderLayout());
      content.add(displayPanel, BorderLayout.CENTER);
      content.add(okButton, BorderLayout.SOUTH);

      JFrame window = new JFrame("GUI Test");
      window.setContentPane(content);
      window.setSize(250,100);
      window.setLocation(100,100);
      window.setVisible(true);

   }

}
```

6.1.1 JFrame and JPanel

In a Java GUI program, each GUI component in the interface is represented by an object in the program. One of the most fundamental types of component is the window. Windows have many behaviors. They can be opened and closed. They can be resized. They have "titles" that are displayed in the title bar above the window. And most important, they can contain other GUI components such as buttons and menus.

Java, of course, has a built-in class to represent windows. There are actually several different types of window, but the most common type is represented by the *JFrame* class (which is included in the package `javax.swing`). A *JFrame* is an independent window that can, for

example, act as the main window of an application. One of the most important things to understand is that a *JFrame* object comes with many of the behaviors of windows already programmed in. In particular, it comes with the basic properties shared by all windows, such as a titlebar and the ability to be opened and closed. Since a *JFrame* comes with these behaviors, you don't have to program them yourself! This is, of course, one of the central ideas of object-oriented programming. What a *JFrame* doesn't come with, of course, is content, the stuff that is contained in the window. If you don't add any other content to a *JFrame*, it will just display a blank area -- or, if you don't set its size, it will be so tiny that it will be hard to find on the screen. You can add content either by creating a *JFrame* object and then adding the content to it or by creating a subclass of *JFrame* and adding the content in the constructor of that subclass.

The main program above declares a variable, `window`, of type *JFrame* and sets it to refer to a new window object with the statement:

```
JFrame window = new JFrame("GUI Test");
```

The parameter (the string "GUI test") in the constructor specifies the title that will be displayed in the titlebar of the window. This line creates the window object, but the window itself is not yet visible on the screen. Before making the window visible, some of its properties are set with these statements:

```
window.setContentPane(content);
window.setSize(250,100);
window.setLocation(100,100);
```

The first line here sets the content of the window. (The content itself was created earlier in the main program.) The second line says that the window will be 250 pixels wide and 100 pixels high. The third line says that the upper left corner of the window will be 100 pixels over from the left edge of the screen and 100 pixels down from the top. Once all this has been set up, the window is actually made visible on the screen with the command:

```
window.setVisible(true);
```

It might look as if the program ends at that point, and, in fact, the `main()` routine does end. However, the window is still on the screen and the program as a whole does not end until the user clicks the OK button. Once the window was opened, a new thread was created to manage the graphical user interface, and that thread continues to run even after `main()` has finished.

The content that is displayed in a *JFrame* is called its content pane. (In addition to its content pane, a *JFrame* can also have a menu bar, which is a separate thing that I will talk about later.) A basic *JFrame* already has a blank content pane; you can either add things to that pane or you can replace the basic content pane entirely. In my sample program, the line `window.setContentPane(content)` replaces the original blank content pane with a different component. (Remember that a "component" is just a visual element of a graphical user interface.) In this case, the new content is a component of type *JPanel*.

JPanel is another of the fundamental classes in Swing. The basic *JPanel* is, again, just a blank rectangle. There are two ways to make a useful *JPanel*: The first is to **add other components** to the panel; the second is to **draw something** in the panel. Both of these techniques are illustrated in the sample program. In fact, you will find two *JPanels* in the program: `content`, which is used to contain other components, and `displayPanel`, which is used as a drawing surface.

Let's look more closely at `displayPanel`. This variable is of type *HelloWorldDisplay*, which is a static nested class inside the *HelloWorldGUI2* class. (Nested classes were introduced in Section 5.8.) This class defines just one instance method, `paintComponent()`, which overrides a method of the same name in the *JPanel* class:

```
private static class HelloWorldDisplay extends JPanel {
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      g.drawString( "Hello World!", 20, 30 );
   }
}
```

The `paintComponent()` method is called by the system when a component needs to be painted on the screen. In the *JPanel* class, the `paintComponent` method simply fills the panel with the panel's background color. The `paintComponent()` method in *HelloWorldDisplay* begins by calling `super.paintComponent(g)`. This calls the version of `paintComponent()` that is defined in the superclass, *JPanel*; that is, it fills the panel with the background color. (See Subsection 5.6.2 for a discussion of the special variable `super`.) Then it calls `g.drawString()` to paint the string "Hello World!" onto the panel. The net result is that whenever a *HelloWorldDisplay* is shown on the screen, it displays the string "Hello World!".

We will often use *JPanels* in this way, as drawing surfaces. Usually, when we do this, we will define a class that is a subclass of *JPanel* and we will write a `paintComponent` method in that class to draw the desired content in the panel. The subclass of *JPanel* can be defined either as a separate class in its own file or as a nested class. In simple cases, I will tend to use a nested class for the convenience.

6.1.2 Components and Layout

Another way of using a *JPanel* is as a container to hold other components. Java has many classes that define GUI components. Except for top-level components like windows, components must be added to a container before they can appear on the screen. In the sample program, the variable named `content` refers to a *JPanel* that is used as a container. Two other components are added to that container. This is done in the statements:

```
content.add(displayPanel, BorderLayout.CENTER);
content.add(okButton, BorderLayout.SOUTH);
```

Here, `content` refers to an object of type *JPanel*; later in the program, this panel becomes the content pane of the window. The first component that is added to `content` is `displayPanel` which, as discussed above, displays the message, "Hello World!". The second is `okButton` which represents the button that the user clicks to close the window. The variable `okButton` is of type *JButton*, the Java class that represents push buttons.

The "BorderLayout" stuff in these statements has to do with how the two components are arranged in the container. When components are added to a container, there has to be some way of deciding how those components are arranged inside the container. This is called "laying out" the components in the container, and the most common technique for laying out components is to use a layout manager. A layout manager is an object that implements some policy for how to arrange the components in a container; different types of layout manager implement different policies. One type of layout manager is defined by the *BorderLayout* class. In the program, the statement

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

creates a new *BorderLayout* object and tells the `content` panel to use the new object as its layout manager. Essentially, this line determines how components that are added to the content panel will be arranged inside the panel. We will cover layout managers in much more detail later, but for now all you need to know is that adding `okButton` in the `BorderLayout.SOUTH` position puts the button at the bottom of the panel, and putting `displayPanel` in the `BorderLayout.CENTER` position makes it fill any space that is not taken up by the button.

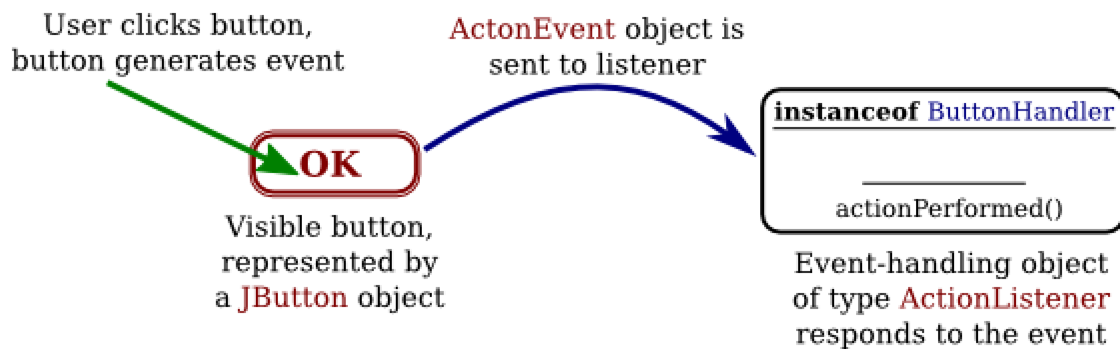
This example shows a general technique for setting up a GUI: Create a container and assign a layout manager to it, create components and add them to the container, and use the container as the content pane of a window. A container is itself a component, so it is possible that some of the components that are added to the top-level container are themselves containers, with their own layout managers and components. This makes it possible to build up complex user interfaces in a hierarchical fashion, with containers inside containers inside containers...

6.1.3 Events and Listeners

The structure of containers and components sets up the physical appearance of a GUI, but it doesn't say anything about how the GUI **behaves**. That is, what can the user do to the GUI and how will it respond? GUIs are largely event-driven; that is, the program waits for events that are generated by the user's actions (or by some other cause). When an event occurs, the program responds by executing an event-handling method. In order to program the behavior of a GUI, you have to write event-handling methods to respond to the events that you are interested in.

The most common technique for handling events in Java is to use event listeners. A listener is an object that includes one or more event-handling methods. When an event is detected by another object, such as a button or menu, the listener object is notified and it responds by running the appropriate event-handling method. An event is detected or generated by an object. Another

object, the listener, has the responsibility of responding to the event. The event itself is actually represented by a third object, which carries information about the type of event, when it occurred, and so on. This division of responsibilities makes it easier to organize large programs.



As an example, consider the OK button in the sample program. When the user clicks the button, an event is generated. This event is represented by an object belonging to the class *ActionEvent*. The event that is generated is associated with the button; we say that the button is the source of the event. The listener object in this case is an object belonging to the class *ButtonHandler*, which is defined as a nested class inside *HelloWorldGUI2*:

```
private static class ButtonHandler implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      System.exit(0);
   }
}
```

This class implements the *ActionListener* interface -- a requirement for listener objects that handle events from buttons. (Interfaces were introduced in Section 5.7.) The event-handling method is named `actionPerformed`, as specified by the *ActionListener* interface. This method contains the code that is executed when the user clicks the button; in this case, the code is simply a call to `System.exit()`, which will terminate the program.

There is one more ingredient that is necessary to get the event from the button to the listener object: The listener object must register itself with the button as an event listener. This is done with the statement:

```
okButton.addActionListener(listener);
```

This statement tells `okButton` that when the user clicks the button, the ActionEvent that is generated should be sent to `listener`. Without this statement, the button has no way of knowing that there is something that would like to listen for events from the button.

This example shows a general technique for programming the behavior of a GUI: Write classes that include event-handling methods. Create objects that belong to these classes and register them as listeners with the objects that will actually detect or generate the events. When an event occurs, the listener is notified, and the code that you wrote in one of its event-handling methods is executed. At first, this might seem like a very roundabout and complicated way to get things

done, but as you gain experience with it, you will find that it is very flexible and that it goes together very well with object oriented programming.

This section has introduced some of the fundamentals of GUI programming. We will spend the rest of the chapter exploring them in more detail.

6.1.4 Some Java GUI History

The original GUI toolkit for Java was the AWT, the "Abstract Windowing Toolkit." It provided a common interface to the GUI components already built into various operating systems. At the very beginning, it used a simpler event model that did not require listener objects, but that model was abandoned in favor of listeners very quickly in Java 1.1.

When Java was first introduced, one of the important applications was applets. An applet is a GUI program that can run on a web page in a web browser. Applets were covered in previous versions of this textbook, but they have become much less widely used and have been dropped from this seventh edition of the book.

The `Swing` GUI toolkit was introduced in Java 1.2 as an improved alternative to the AWT, with a larger variety of sophisticated components and a more logical structure. Although Swing uses some aspects of the AWT, most of its components are written in Java rather than being based on operating system components. Swing has been the standard toolkit for writing GUI programs in Java for over ten years, and it is the toolkit that I cover in this book.

More recently, however, another GUI toolkit called JavaFX has been introduced. It uses many of the same core ideas as Swing, including components, layout, and events, but uses a different structure for its applications and a different set of classes. With Java 8, JavaFX becomes the preferred approach to writing GUI applications. However, I do not cover JavaFX in this book. JavaFX is compatible with Swing and can use Swing components, and Swing will continue to be supported in Java. (Indeed, the AWT is still supported!) And as I've said, JavaFX is built on the same core ideas as Swing.

Graphics and Painting

EVERYTHING YOU SEE ON A COMPUTER SCREEN has to be drawn there, even the text. The Java API includes a range of classes and methods that are devoted to drawing. In this section, I'll look at some of the most basic of these. Some of this material was already covered in preliminary form in Section 3.9.

The physical structure of a GUI is built of components. The term component refers to a visual element in a GUI, including buttons, menus, text-input boxes, scroll bars, check boxes, and so on. In Java, GUI components are represented by objects belonging to subclasses of the class

`java.awt.Component`. Most components in the Swing GUI toolkit -- although not top-level components like JFrame -- belong to subclasses of the class `javax.swing.JComponent`, which is itself a subclass of `java.awt.Component`. Every component is responsible for drawing itself. If you want to use a standard component, you only have to add it to your program. You don't have to worry about painting it on the screen. That will happen automatically, since it already knows how to draw itself.

Sometimes, however, you do want to draw on a component. You will have to do this whenever you want to display something that is not included among the standard, pre-defined component classes. When you want to do this, you have to define your own component class and provide a method in that class for drawing the component. I will always use a subclass of *JPanel* when I need a drawing surface of this kind, as I did for the *HelloWorldDisplay* class in the example *HelloWorldGUI2.java* in the previous section. A JPanel, like any JComponent, draws its content in the method

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

To create a drawing surface, you should define a subclass of `JPanel` and provide a custom `paintComponent()` method. Create an object belonging to this class and use it in your program. When the time comes for your component to be drawn on the screen, the system will call its `paintComponent()` to do the drawing. That is, the code that you put into the `paintComponent()` method will be executed whenever the panel needs to be drawn on the screen; by writing this method, you determine the picture that will be displayed in the panel. Note that you are not likely to call a `paintComponent()` method any more than you are likely to call a `main()` routine. The *system* calls the method. You *write* the method to say what will happen when the system calls it.

Note that the `paintComponent()` method has a parameter of type *Graphics*. The *Graphics* object will be provided by the system when it calls your method. You need this object to do the actual drawing. To do any drawing at all in Java, you need a graphics context. A graphics context is an object belonging to the class `java.awt.Graphics`. Instance methods are provided in this class for drawing shapes, text, and images. Any given *Graphics* object can draw to only one location. In this chapter, that location will always be a GUI component belonging to some subclass of `JPanel`. The *Graphics* class is an abstract class, which means that it is impossible to create a graphics context directly, with a constructor. There are actually two ways to get a graphics context for drawing on a component: First of all, of course, when the `paintComponent()` method of a component is called by the system, the parameter to that method is a graphics context for drawing on the component. Second, every component has an instance method called `getGraphics()`. This method is a function that returns a graphics context that can be used for drawing on the component outside its `paintComponent()` method. The official line is that you should **not** do this, and I will almost always avoid it. But I have found it convenient to use `getGraphics()` in a few examples. (Note that if `g` is a graphics context created with `getGraphics()`, it is good form to call `g.dispose()` when finished using it. This releases any operating system resources that might be held by `g`.)

The `paintComponent()` method in the *JPanel* class simply fills the panel with the panel's background color. When defining a subclass of *JPanel* for use as a drawing surface, you will usually want to fill the panel with the background color before drawing other content onto the panel (although it is not necessary to do this if the drawing commands in the method cover the background of the component completely). This is traditionally done with a call to `super.paintComponent(g)`, so most `paintComponent()` methods that you write will have the form:

```
public void paintComponent(g) {
   super.paintComponent(g);
   . . . // Draw the content of the component.
}
```

In general, a component should do all drawing operations in its `paintComponent()` method. What happens if, in the middle of some other method, you realize that the content of the component needs to be changed? You should **not** call `paintComponent()` directly to make the change. Instead, you have to inform the system that the component needs to be redrawn, and let the system do its job by calling `paintComponent()`. You do this by calling the component's `repaint()` method. The method

```
public void repaint();
```

is defined in the `Component` class, and so can be used with any component. You should call `repaint()` to inform the system that the component needs to be redrawn. It is important to understand that the `repaint()` method returns immediately, without doing any painting itself. The system will call the component's `paintComponent()` method *later*, as soon as it gets a chance to do so, after processing other pending events if there are any. It is even possible that many calls to `repaint()` will all be handled by one call to `paintComponent()`, if the calls to `repaint()` occur in a very short timespan.

Note that the system can also call `paintComponent()` for other reasons. It is called when the component first appears on the screen. It will also be called if the size of the component changes, which can happen when the user resizes the window that contains the component. This means that `paintComponent()` should be capable of redrawing the content of the component on demand. As you will see, however, some of our early examples will not be able to do this correctly.

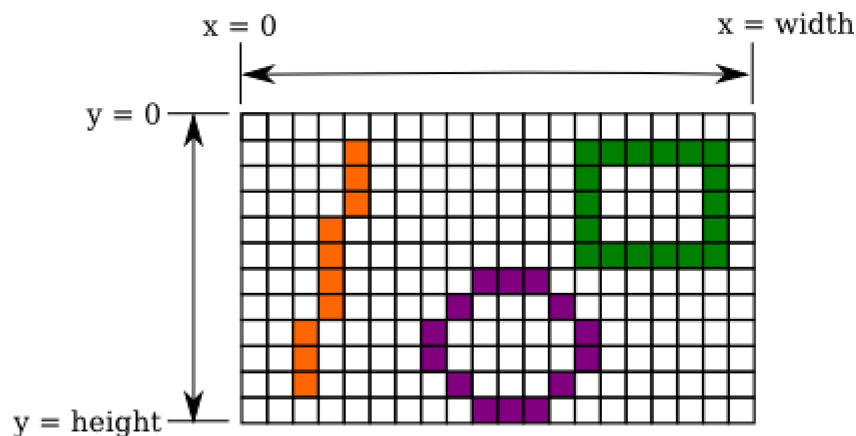
This means that, to work properly, the `paintComponent()` method must be smart enough to correctly redraw the component at any time. To make this possible, a program should store data in its instance variables about the state of the component. These variables should contain all the information necessary to redraw the component completely. The `paintComponent()` method should use the data in these variables to decide what to draw. When the program wants to change the content of the component, it should not simply draw the new content. It should change the values of the relevant variables and call `repaint()`. When the system calls `paintComponent()`, that method will use the new values of the variables and will draw the

component with the desired modifications. This might seem a roundabout way of doing things. Why not just draw the modifications directly? There are at least two reasons. First of all, it really does turn out to be easier to get things right if all drawing is done in one method. Second, even if you could directly draw the modifications, you would still have to save enough information about the modifications to enable `paintComponent()` to **redraw** the component correctly on demand.

You will see how all this works in practice as we work through examples in the rest of this chapter. For now, we will spend the rest of this section looking at how to get some actual drawing done.

6.2.1 Coordinates

The screen of a computer is a grid of little squares called pixels. The color of each pixel can be set individually, and drawing on the screen just means setting the colors of individual pixels.



A graphics context draws in a rectangle made up of pixels. A position in the rectangle is specified by a pair of integer coordinates, `(x,y)`. The upper left corner has coordinates `(0,0)`. The `x` coordinate increases from left to right, and the `y` coordinate increases from top to bottom. The illustration shows a 20-pixel by 12-pixel component (with very large pixels). A small line, rectangle, and oval are shown as they would be drawn by coloring individual pixels. (Note that, properly speaking, the coordinates don't belong to the pixels but to the grid lines between them.)

For any component, you can find out the size of the rectangle that it occupies by calling the instance methods `getWidth()` and `getHeight()`, which return the number of pixels in the horizontal and vertical directions, respectively. In general, it's not a good idea to assume that you know the size of a component, since the size is often set by a layout manager and can even change if the component is in a window and that window is resized by the user. This means that it's good form to check the size of a component before doing any drawing on that component. For example, you can use a `paintComponent()` method that looks like:

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);
   int width =  getWidth();   // Find out the width of this
component.
   int height = getHeight();  // Find out its height.
   . . .   // Draw the content of the component.
}
```

Of course, your drawing commands will have to take the size into account. That is, they will have to use (x,y) coordinates that are calculated based on the actual height and width of the component. (However, if you are sure that you know the size, using constants for the width and height can make the drawing easier.)

6.2.2 Colors

You will probably want to use some color when you draw. Java is designed to work with the RGB color system. An RGB color is specified by three numbers that give the level of red, green, and blue, respectively, in the color. A color in Java is an object of the class, java.awt.Color. You can construct a new color by specifying its red, blue, and green components. For example,

```
Color myColor = new Color(r,g,b);
```

There are two constructors that you can call in this way. In the one that I almost always use, r, g, and b are integers in the range 0 to 255. In the other, they are numbers of type float in the range 0.0F to 1.0F. (Recall that a literal of type float is written with an "F" to distinguish it from a double number.) Often, you can avoid constructing new colors altogether, since the *Color* class defines several named constants representing common colors: Color.WHITE, Color.BLACK, Color.RED, Color.GREEN, Color.BLUE, Color.CYAN, Color.MAGENTA, Color.YELLOW, Color.PINK, Color.ORANGE, Color.LIGHT_GRAY, Color.GRAY, and Color.DARK_GRAY. (There are older, alternative names for these constants that use lower case rather than upper case constants, such as Color.red instead of Color.RED, but the upper case versions are preferred because they follow the convention that constant names should be upper case.)

An alternative to RGB is the HSB color system. In the HSB system, a color is specified by three numbers called the hue, the saturation, and the brightness. The hue is the basic color, ranging from red through orange through all the other colors of the rainbow. The brightness is pretty much what it sounds like. A fully saturated color is a pure color tone. Decreasing the saturation is like mixing white or gray paint into the pure color. In Java, the hue, saturation and brightness are always specified by values of type float in the range from 0.0F to 1.0F. The *Color* class has a static member function named getHSBColor for creating HSB colors. To create the color with HSB values given by h, s, and b, you can say:

```
Color myColor = Color.getHSBColor(h,s,b);
```

For example, to make a color with a random hue that is as bright and as saturated as possible, you could use:

```
Color randomColor = Color.getHSBColor( (float)Math.random(), 1.0F,
1.0F );
```

The type cast is necessary because the value returned by `Math.random()` is of type double, and `Color.getHSBColor()` requires values of type float. (By the way, you might ask why RGB colors are created using a constructor while HSB colors are created using a static member function. The problem is that we would need two different constructors, both of them with three parameters of type float. Unfortunately, this is impossible. You can have two constructors only if the number of parameters or the parameter types differ.)

The RGB system and the HSB system are just different ways of describing the same set of colors. It is possible to translate between one system and the other. The best way to understand the color systems is to experiment with them. (The sample program *SimpleColorChooser.java* lets you do that. You won't understand the source code at this time, but you can run it to play with color selection or to find the RGB or HSB values for the color that want.)

One of the properties of a *Graphics* object is the current drawing color, which is used for all drawing of shapes and text. If `g` is a graphics context, you can change the current drawing color for `g` using the method `g.setColor(c)`, where `c` is a *Color*. For example, if you want to draw in green, you would just say `g.setColor(Color.GREEN)` before doing the drawing. The graphics context continues to use the color until you explicitly change it with another `setColor()` command. If you want to know what the current drawing color is, you can call the function `g.getColor()`, which returns an object of type *Color*. This can be useful if you want to change to another drawing color temporarily and then restore the previous drawing color.

Every component has an associated foreground color and background color. Generally, the component is filled with the background color before anything else is drawn (although some components are "transparent," meaning that the background color is ignored). When a new graphics context is created for a component, the current drawing color is set to the foreground color. Note that the foreground color and background color are properties of the component, not of a graphics context.

The foreground and background colors of a component can be set by calling instance methods `component.setForeground(color)` and `component.setBackground(color)`, which are defined in the *Component* class and therefore are available for use with any component. This can be useful even for standard components, if you want them to use colors that are different from the defaults.

6.2.3 Fonts

A font represents a particular size and style of text. The same character will appear different in different fonts. In Java, a font is characterized by a font name, a style, and a size. The available font names are system dependent, but you can always use the following four strings as font names: "Serif", "SansSerif", "Monospaced", and "Dialog". (A "serif" is a little decoration on a character, such as a short horizontal line at the bottom of the letter i. "SansSerif" means "without serifs." "Monospaced" means that all the characters in the font have the same width. The "Dialog" font is the one that is typically used in dialog boxes.)

The style of a font is specified using named constants that are defined in the *Font* class. You can specify the style as one of the four values:

- `Font.PLAIN`,
- `Font.ITALIC`,
- `Font.BOLD`, or
- `Font.BOLD + Font.ITALIC`.

The size of a font is an integer. Size typically ranges from about 9 to 36, although larger sizes can also be used. The size of a font is usually about equal to the height of the largest characters in the font, in pixels, but this is not an exact rule. The size of the default font is 12.

Java uses the class named `java.awt.Font` for representing fonts. You can construct a new font by specifying its font name, style, and size in a constructor:

```
Font plainFont = new Font("Serif", Font.PLAIN, 12);
Font bigBoldFont = new Font("SansSerif", Font.BOLD, 24);
```

Every graphics context has a current font, which is used for drawing text. You can change the current font with the `setFont()` method. For example, if `g` is a graphics context and `bigBoldFont` is a font, then the command `g.setFont(bigBoldFont)` will set the current font of `g` to `bigBoldFont`. The new font will be used for any text that is drawn *after* the `setFont()` command is given. You can find out the current font of `g` by calling the method `g.getFont()`, which returns an object of type *Font*.

Every component also has an associated font. It can be set with the instance method `component.setFont(font)`, which is defined in the *Component* class. When a graphics context is created for drawing on a component, the graphic context's current font is set equal to the font of the component.

6.2.4 Shapes

The *Graphics* class includes a large number of instance methods for drawing various shapes, such as lines, rectangles, and ovals. The shapes are specified using the `(x,y)` coordinate system described above. They are drawn in the current drawing color of the graphics context. The

current drawing color is set to the foreground color of the component when the graphics context is created, but it can be changed at any time using the `setColor()` method.

Some drawing methods were already listed in Subsection 3.9.1. Here, I describe those methods in more detail and add a few more. With all these commands, any drawing that is done outside the boundaries of the component is ignored. Note that all these methods are in the *Graphics* class, so they all must be called through an object of type *Graphics*. It is shown here as `g`, but of course the name of the graphics context is up to the programmer.

- `g.drawString(String str, int x, int y)` -- Draws the text given by the string `str`. The string is drawn using the current color and font of the graphics context. `x` specifies the x-coordinate of the left end of the string. `y` is the y-coordinate of the baseline of the string. The baseline is a horizontal line on which the characters rest. Some parts of the characters, such as the tail on a y or g, extend below the baseline.
- `g.drawLine(int x1, int y1, int x2, int y2)` -- Draws a line from the point `(x1,y1)` to the point `(x2,y2)`. The line is drawn as if with a pen that extends one pixel to the right and one pixel down from the `(x,y)` point where the pen is located. For example, if `g` refers to an object of type *Graphics*, then the command `g.drawLine(x,y,x,y)`, which corresponds to putting the pen down at a point, colors the single pixel with upper left corner at the point `(x,y)`. Remember that coordinates really refer to the lines between the pixels.
- `g.drawRect(int x, int y, int width, int height)` -- Draws the outline of a rectangle. The upper left corner is at `(x,y)`, and the width and height of the rectangle are as specified. If `width` equals `height`, then the rectangle is a square. If the `width` or the `height` is negative, then nothing is drawn. The rectangle is drawn with the same pen that is used for `drawLine()`. This means that the actual width of the rectangle as drawn is `width+1`, and similarly for the height. There is an extra pixel along the right edge and the bottom edge. For example, if you want to draw a rectangle around the edges of the component, you can say "`g.drawRect(0, 0, getWidth()-1, getHeight()-1);`". If you use "`g.drawRect(0, 0, getWidth(), getHeight());`", then the right and bottom edges of the rectangle will be drawn *outside* the component and will not appear on the screen.
- `g.drawOval(int x, int y, int width, int height)` -- Draws the outline of an oval. The oval is one that just fits inside the rectangle specified by `x`, `y`, `width`, and `height`. If `width` equals `height`, the oval is a circle.
- `g.drawRoundRect(int x, int y, int width, int height, int xdiam, int ydiam)` -- Draws the outline of a rectangle with rounded corners. The basic rectangle is specified by `x`, `y`, `width`, and `height`, but the corners are rounded. The degree of rounding is given by `xdiam` and `ydiam`. The corners are arcs of an ellipse with horizontal diameter `xdiam` and vertical diameter `ydiam`. A typical value for `xdiam` and `ydiam` is 16, but the value used should really depend on how big the rectangle is.
- `g.draw3DRect(int x, int y, int width, int height, boolean raised)` -- Draws the outline of a rectangle that is supposed to have a three-dimensional effect, as if it is raised from the screen or pushed into the screen. The basic rectangle is specified by `x`, `y`, `width`, and `height`. The `raised` parameter tells whether the rectangle seems to be raised from the screen or pushed into it. The 3D effect is achieved by using brighter and darker versions of the drawing color for different edges of the rectangle. The documentation recommends setting the drawing color equal to the background color before using this method. The effect won't work well for some colors.

- `g.drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)` -- Draws part of the oval that just fits inside the rectangle specified by `x`, `y`, `width`, and `height`. The part drawn is an arc that extends `arcAngle` degrees from a starting angle at `startAngle` degrees. Angles are measured with 0 degrees at the 3 o'clock position (the positive direction of the horizontal axis). Positive angles are measured counterclockwise from zero, and negative angles are measured clockwise. To get an arc of a circle, make sure that `width` is equal to `height`.
- `g.fillRect(int x, int y, int width, int height)` -- Draws a filled-in rectangle. This fills in the interior of the rectangle that would be drawn by `drawRect(x,y,width,height)`. The extra pixel along the bottom and right edges is not included. The `width` and `height` parameters give the exact width and height of the rectangle. For example, if you wanted to fill in the entire component, you could say "`g.fillRect(0, 0, getWidth(), getHeight());`"
- `g.fillOval(int x, int y, int width, int height)` -- Draws a filled-in oval.
- `g.fillRoundRect(int x, int y, int width, int height, int xdiam, int ydiam)` -- Draws a filled-in rounded rectangle.
- `g.fill3DRect(int x, int y, int width, int height, boolean raised)` -- Draws a filled-in three-dimensional rectangle.
- `g.fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)` -- Draw a filled-in arc. This looks like a wedge of pie, whose crust is the arc that would be drawn by the `drawArc` method.

6.2.5 Graphics2D

All drawing in Java is done through an object of type *Graphics*. The *Graphics* class provides basic commands for such things as drawing shapes and text and for selecting a drawing color. These commands are adequate in many cases, but they fall far short of what's needed in a serious computer graphics program. Java has another class, *Graphics2D*, that provides a larger set of drawing operations. *Graphics2D* is a sub-class of *Graphics*, so all the methods from the *Graphics* class are also available in a *Graphics2D*.

The `paintComponent()` method of a `JComponent` gives you a graphics context of type *Graphics* that you can use for drawing on the component. In fact, the graphics context actually belongs to the sub-class *Graphics2D*, and can be type-cast to gain access to the advanced *Graphics2D* drawing methods:

```
public void paintComponent(Graphics g) {
   super.paintComponent(g);
   Graphics2D g2;
   g2 = (Graphics2D)g;
     .
     . // Draw on the component using g2.
     .
}
```

I mention *Graphics2D* here for completeness. I will cover some important aspects of *Graphics2D* in Section 13.2, but a full treatment is more than we will have time for in this book. However, there are two simple applications that I would like to start using now, without explaining how they work. If `g2` is a variable of type *Graphics2D*, as in the `paintComponent()` method above, then the intimidating-looking command

```
g2.setRenderingHint( RenderingHints.KEY_ANTIALIASING,
                                     RenderingHints.VALUE_ANTIALIAS_ON
);
```

turns on antialiasing in the graphics context. Aliasing is a jagged appearance that can be seen when shapes are drawn using pixels. Antialiasing tries to reduce the jaggedness. It can make diagonal lines and the outlines of ovals look much nicer. It can also improve the appearance of text. Another useful command is

```
g2.setStroke( new BasicStroke(lineWidth) );
```

where `lineWidth` is an integer or a float. This command can be use to draw thicker lines. Lines drawn after the command will be `lineWidth` pixels wide. This also affects the thickness of the outlined shapes drawn by methods such as `g.drawRect` and `g.drawOval()`.

6.2.6 An Example

Let's use some of the material covered in this section to write a subclass of *JPanel* for use as a drawing surface. All the drawing will be done in the `paintComponent()` method of the panel class. The panel will draw multiple copies of a message on a black background. Each copy of the message is in a random color. Five different fonts are used, with different sizes and styles. The message can be specified in the constructor; if the default constructor is used, the message is the string "Java!". The panel works OK no matter what its size.

There is one problem with the way this class works. When the panel's `paintComponent()` method is called, it chooses random colors, fonts, and locations for the messages. The information about which colors, fonts, and locations are used is not stored anywhere. The next time `paintComponent()` is called, it will make different random choices and will draw a different picture. If you resize a window containing the panel, the picture will be continually redrawn as the size of the window is changed! To avoid that, you would store enough information about the picture in instance variables to enable the `paintComponent()` method to draw the same picture each time it is called.

The source code for the panel class is shown below. I use an instance variable called `message` to hold the message that the panel will display. There are five instance variables of type *Font* that represent different sizes and styles of text. These variables are initialized in the constructor and are used in the `paintComponent()` method.

The `paintComponent()` method for the panel simply draws 25 copies of the message. For each copy, it chooses one of the five fonts at random, and it uses `g.setFont()` to select that font for drawing. It creates a random HSB color and uses `g.setColor()` to select that color for drawing. It then chooses random `(x,y)` coordinates for the location of the message. The `x` coordinate gives the horizontal position of the left end of the string. The formula used for the `x` coordinate is "`-50 + (int)(Math.random() * (width+40))`". This gives a random integer in the range from `-50` to `width-10`. This makes it possible for the string to extend beyond the left edge or the right edge of the panel. Similarly, the formula for `y` allows the string to extend beyond the top and bottom.

Here is the complete source code for the *RandomStringsPanel*:

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

/**
 * This panel displays 25 copies of a message.  The color and
 * position of each message is selected at random.  The font
 * of each message is randomly chosen from among five possible
 * fonts.  The messages are displayed on a black background.
 * Note:  The style of drawing used here is poor, because every
 * time the paintComponent() method is called, new random values
are
 * used.  This means that a different picture will be drawn each
time.
 */
public class RandomStringsPanel extends JPanel {

    private String message;  // The message to be displayed.  This
can be set in
                             // the constructor.  If no value is
provided in the
                             // constructor, then the string
"Java!" is used.

    private Font font1, font2, font3, font4, font5;  // The five
fonts.

    /**
     * Default constructor creates a panel that displays the
message "Java!".
     */
    public RandomStringsPanel() {
        this(null);  // Call the other constructor, with parameter
null.
    }

    /**
     * Constructor creates a panel to display 25 copies of a
specified message.
     * @param messageString The message to be displayed.  If this
is null,
     * then the default message "Java!" is displayed.
```

```
     */
    public RandomStringsPanel(String messageString) {

        message = messageString;
        if (message == null)
            message = "Java!";

        font1 = new Font("Serif", Font.BOLD, 14);
        font2 = new Font("SansSerif", Font.BOLD + Font.ITALIC, 24);
        font3 = new Font("Monospaced", Font.PLAIN, 30);
        font4 = new Font("Dialog", Font.PLAIN, 36);
        font5 = new Font("Serif", Font.ITALIC, 48);

        setBackground(Color.BLACK);

    }

    /**
     * The paintComponent method is responsible for drawing the
content of the panel.
     * It draws 25 copies of the message string, using a random
color, font, and
     * position for each string.
     */
    public void paintComponent(Graphics g) {

        super.paintComponent(g);  // Call the paintComponent method
from the
                                  // superclass, JPanel.  This
simply fills the
                                  // entire panel with the
background color, black.

        Graphics2D g2 = (Graphics2D)g;  // (To make the text
smoother.)
        g2.setRenderingHint( RenderingHints.KEY_ANTIALIASING,
                                RenderingHints.VALUE_ANTIALIAS_ON
);

        int width = getWidth();
        int height = getHeight();

        for (int i = 0; i < 25; i++) {

            // Draw one string.  First, set the font to be one of
the five
            // available fonts, at random.

            int fontNum = (int)(5*Math.random()) + 1;
            switch (fontNum) {
            case 1:
                g.setFont(font1);
                break;
            case 2:
                g.setFont(font2);
                break;
            case 3:
```

```
                    g.setFont(font3);
                    break;
                case 4:
                    g.setFont(font4);
                    break;
                case 5:
                    g.setFont(font5);
                    break;
                } // end switch

                // Set the color to a bright, saturated color, with
random hue.

                float hue = (float)Math.random();
                g.setColor( Color.getHSBColor(hue, 1.0F, 1.0F) );

                // Select the position of the string, at random.

                int x,y;
                x = -50 + (int)(Math.random()*(width+40));
                y = (int)(Math.random()*(height+20));

                // Draw the message.

                g.drawString(message,x,y);

            } // end for

        } // end paintComponent()


}  // end class RandomStringsPanel
```

6.2.7 Where is main()?

The source code for the *RandomStringsPanel* class can be found in the example file *RandomStringsPanel.java*. You can compile that file, but you won't be able to run the compiled class. The problem is that the class doesn't have a `main()` routine. Only a class that has a `main()` routine can be run as a program.

Another problem is that a *JPanel* is not something that can stand on its own. It has to be placed into a container such as another panel or a window. In general, to make a complete program, we need a `main()` routine that will create a window of type *JFrame*. It can then create a panel and place the panel in the window. Here is a class with a `main()` routine that does this:

```
import javax.swing.JFrame;

public class RandomStrings {

    public static void main(String[] args) {
        JFrame window = new JFrame("Java!");
        RandomStringsPanel content = new RandomStringsPanel();
```

```
            window.setContentPane(content);
            window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            window.setLocation(120,70);
            window.setSize(350,250);
            window.setVisible(true);
        }

    }
```

This class is defined by the file *RandomStrings.java*. You can compile and run the program, as long as the *RandomStringsPanel* class is also available.

The main routine is not logically a part of the panel class. It is just one way of using a panel. However, it's possible to include `main()` as part of the panel class, even if it doesn't logically belong there. This makes it possible to run the panel class as a program, and it has the advantage of keeping everything in one file. For an example, you can look at *RandomStringsPanelWithMain.java*, which is identical to the original class except for the addition of a `main()` routine. Although it might not be great style, I will usually take a similar approach in future examples

Mouse Events

EVENTS ARE CENTRAL to programming for a graphical user interface. A GUI program doesn't have a `main()` routine that outlines what will happen when the program is run, in a step-by-step process from beginning to end. Instead, the program must be prepared to respond to various kinds of events that can happen at unpredictable times and in an order that the program doesn't control. The most basic kinds of events are generated by the mouse and keyboard. The user can press any key on the keyboard, move the mouse, or press a button on the mouse. The user can do any of these things at any time, and the computer has to respond appropriately.

In Java, events are represented by objects. When an event occurs, the system collects all the information relevant to the event and constructs an object to contain that information. Different types of events are represented by objects belonging to different classes. For example, when the user presses one of the buttons on a mouse, an object belonging to a class called *MouseEvent* is constructed. The object contains information such as the source of the event (that is, the component on which the user clicked), the `(x,y)` coordinates of the point in the component where the click occurred, the exact time of the click, and which button on the mouse was pressed. When the user presses a key on the keyboard, a *KeyEvent* is created. After the event object is constructed, it can be passed as a parameter to a designated method. By writing that method, the programmer says what should happen when the event occurs.

As a Java programmer, you get a fairly high-level view of events. There is a lot of processing that goes on between the time that the user presses a key or moves the mouse and the time that a subroutine in your program is called to respond to the event. Fortunately, you don't need to know much about that processing. But you should understand this much: Even though you didn't write it, there is a routine running somewhere that executes a loop of the form

```
while the program is still running:
    Wait for the next event to occur
    Call a subroutine to handle the event
```

This loop is called an event loop. Every GUI program has an event loop. In Java, you don't have to write the loop. It's part of "the system." If you write a GUI program in some other language, you might have to provide a main routine that runs the event loop.

In this section, we'll look at handling mouse events in Java, and we'll cover the framework for handling events in general. The next section will cover keyboard-related events and timer events. Java also has other types of events, which are produced by GUI components. These will be introduced in Section 6.5.

6.3.1 Event Handling

For an event to have any effect, a program must detect the event and react to it. In order to detect an event, the program must "listen" for it. Listening for events is something that is done by an object called an event listener. An event listener object must contain instance methods for handling the events for which it listens. For example, if an object is to serve as a listener for events of type *MouseEvent*, then it must contain the following method (among several others):

```
public void mousePressed(MouseEvent evt) { . . . }
```

The body of the method defines how the object responds when it is notified that a mouse button has been pressed. The parameter, `evt`, contains information about the event. This information can be used by the listener object to determine its response.

The methods that are required in a mouse event listener are specified in an `interface` named *MouseListener*. To be used as a listener for mouse events, an object must implement this *MouseListener* interface. Java `interfaces` were covered in Section 5.7. (To review briefly: An `interface` in Java is just a list of instance methods. A class can "implement" an interface by doing two things: First, the class must be declared to implement the interface, as in "`class MouseHandler implements MouseListener`" or "`class MyPanel extends JPanel implements MouseListener`"; and second, the class must include a definition for each instance method specified in the interface. An `interface` can be used as the type for a variable or formal parameter. We say that an *object* implements the *MouseListener* interface if it belongs to a *class* that implements the *MouseListener* interface. Note that it is not enough for the object to include the specified methods. It must also belong to a class that is specifically declared to implement the interface.)

Many events in Java are associated with GUI components. For example, when the user presses a button on the mouse, the associated component is the one that the user clicked on. Before a listener object can "hear" events associated with a given component, the listener object must be registered with the component. If a *MouseListener* object, `mListener`, needs to hear mouse

events associated with a *Component* object, `comp`, the listener must be registered with the component by calling

```
comp.addMouseListener(mListener);
```

The `addMouseListener()` method is an instance method in class *Component*, and so can be used with any GUI component object. In our first few examples, we will listen for events on a *JPanel* that is being used as a drawing surface.

The event classes, such as *MouseEvent*, and the listener interfaces, such as *MouseListener*, are defined in the package `java.awt.event`. This means that if you want to work with events, you should either include the line "`import java.awt.event.*;`" at the beginning of your source code file or import the individual classes and interfaces.

Admittedly, there is a large number of details to tend to when you want to use events. To summarize, you must

1. Put the import specification "`import java.awt.event.*;`" (or individual imports) at the beginning of your source code;
2. Declare that some class implements the appropriate listener interface, such as *MouseListener*;
3. Provide definitions in that class for the methods specified by the interface;
4. Register an object that belongs to the listener class with the component that will generate the events by calling a method such as `addMouseListener()` in the component.

Any object can act as an event listener, provided that it implements the appropriate interface. A component can listen for the events that it itself generates. A panel can listen for events from components that are contained in the panel. A special class can be created just for the purpose of defining a listening object. Many people consider it to be good form to use anonymous inner classes to define listening objects (see Subsection 5.8.3), and named nested classes can also be appropriate. You will see all of these patterns in examples in this textbook.

6.3.2 MouseEvent and MouseListener

The *MouseListener* interface specifies these five instance methods:

```
public void mousePressed(MouseEvent evt);
public void mouseReleased(MouseEvent evt);
public void mouseClicked(MouseEvent evt);
public void mouseEntered(MouseEvent evt);
public void mouseExited(MouseEvent evt);
```

The `mousePressed` method is called as soon as the user presses down on one of the mouse buttons, and `mouseReleased` is called when the user releases a button. These are the two methods that are most commonly used, but any mouse listener object must define all five methods; you can leave the body of a method empty if you don't want to define a response. The

`mouseClicked` method is called if the user presses a mouse button and then releases it, without moving the mouse. (When the user does this, all three routines -- `mousePressed`, `mouseReleased`, and `mouseClicked` -- will be called in that order.) In most cases, you should define `mousePressed` instead of `mouseClicked`. The `mouseEntered` and `mouseExited` methods are called when the mouse cursor enters or leaves the component. For example, if you want the component to change appearance whenever the user moves the mouse over the component, you could define these two methods.

As a first example, we will look at a small addition to the *RandomStringsPanel* example from the previous section. In the new version, the panel will repaint itself when the user clicks on it. In order for this to happen, a mouse listener should listen for mouse events on the panel, and when the listener detects a `mousePressed` event, it should respond by calling the `repaint()` method of the panel.

For the new version of the program, we need an object that implements the *MouseListener* interface. One way to create the object is to define a separate class, such as:

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

/**
 * An object of type RepaintOnClick is a MouseListener that
 * will respond to a mousePressed event by calling the repaint()
 * method of the source of the event.  That is, a RepaintOnClick
 * object can be added as a mouse listener to any Component;
 * when the user clicks that component, the component will be
 * repainted.
 */
public class RepaintOnClick implements MouseListener {

   public void mousePressed(MouseEvent evt) {
      Component source = (Component)evt.getSource();
      source.repaint();  // Call repaint() on the Component that
was clicked.
   }

   public void mouseClicked(MouseEvent evt) { }
   public void mouseReleased(MouseEvent evt) { }
   public void mouseEntered(MouseEvent evt) { }
   public void mouseExited(MouseEvent evt) { }

}
```

This class does three of the four things that we need to do in order to handle mouse events: First, it imports `java.awt.event.*` for easy access to event-related classes. Second, it is declared that the class "`implements MouseListener`". And third, it provides definitions for the five methods that are specified in the *MouseListener* interface. (Note that four of the methods have empty bodies, since we don't want to do anything in response to those events.)

We must do one more thing to set up the event handling for this example: We must register an event-handling object as a listener with the component that will generate the events. In this case, the mouse events that we are interested in will be generated by an object of type *RandomStringsPanel*. If `panel` is a variable that refers to the panel object, we can create a mouse listener object and register it with the panel with the statements:

```
RepaintOnClick listener = new RepaintOnClick();  // Create
MouseListener object.
panel.addMouseListener(listener);  // Register MouseListener with
the panel.
```

This could be done, for example, in the `main()` routine where the panel is created. Once the `listener` has been registered in this way, it will be notified of mouse events on the panel. When a `mousePressed` event occurs, the `mousePressed()` method in the `listener` will be called. The code in this method calls the `repaint()` method in the component that is the source of the event, that is, in the panel. The result is that the `RandomStringsPanel` is repainted with its strings in new random colors, fonts, and positions.

Although we have written the *RepaintOnClick* class for use with our *RandomStringsPanel* example, the event-handling class contains no reference at all to the *RandomStringsPanel* class. How can this be? The `mousePressed()` method in class *RepaintOnClick* looks at the source of the event, and calls its `repaint()` method. If we have registered the *RepaintOnClick* object as a listener on a *RandomStringsPanel*, then it is that panel that is repainted. But the listener object could be used with any type of component, and it would work in the same way.

Similarly, the *RandomStringsPanel* class contains no reference to the *RepaintOnClick* class -- in fact, *RandomStringsPanel* was written before we even knew anything about mouse events! The panel will send mouse events to any object that has registered with it as a mouse listener. It does not need to know anything about that object except that it is capable of receiving mouse events.

The relationship between an object that generates an event and an object that responds to that event is rather loose. The relationship is set up by registering one object to listen for events from the other object. This is something that can potentially be done from outside both objects. Each object can be developed independently, with no knowledge of the internal operation of the other object. This is the essence of modular design: Build a complex system out of modules that interact only in straightforward, easy to understand ways. Then each module is a separate design problem that can be tackled independently. Java's event-handling framework is designed to offer strong support for modular design.

To make this clearer, let's look at a new version of *RandomStrings.java*, the program from Subsection 6.2.7 that uses *RandomStringsPanel*. The new version is *ClickableRandomStrings.java*. For convenience, I have added *RepaintOnClick* as a static nested class, although it would work just as well as a separate class:

```
import java.awt.Component;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
```

```
import javax.swing.JFrame;

/**
 * Displays a window that shows 25 copies of the string "Java!" in
 * random colors, fonts, and positions.  The content of the window
 * is an object of type RandomStringsPanel.  When the user clicks
 * the window, the content of the window is repainted, with the
 * strings in newly selected random colors, fonts, and positions.
 */
public class ClickableRandomStrings {

    public static void main(String[] args) {
        JFrame window = new JFrame("Click Me to Redraw!");
        RandomStringsPanel content = new RandomStringsPanel();
        content.addMouseListener( new RepaintOnClick() );
        window.setContentPane(content);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setLocation(120,70);
        window.setSize(350,250);
        window.setVisible(true);
    }

    private static class RepaintOnClick implements MouseListener {

        public void mousePressed(MouseEvent evt) {
            Component source = (Component)evt.getSource();
            source.repaint();
        }

        public void mouseClicked(MouseEvent evt) { }
        public void mouseReleased(MouseEvent evt) { }
        public void mouseEntered(MouseEvent evt) { }
        public void mouseExited(MouseEvent evt) { }

    }

} end class ClickableRandomStrings
```

6.3.3 MouseEvent Data

Often, when a mouse event occurs, you want to know the location of the mouse cursor. This information is available from the *MouseEvent* parameter to the event-handling method, which contains instance methods that return information about the event. If `evt` is the parameter, then you can find out the coordinates of the mouse cursor by calling `evt.getX()` and `evt.getY()`. These methods return integers which give the `x` and `y` coordinates where the mouse cursor was positioned at the time when the event occurred. The coordinates are expressed in the coordinate system of the component that generated the event, where the top left corner of the component is (0,0).

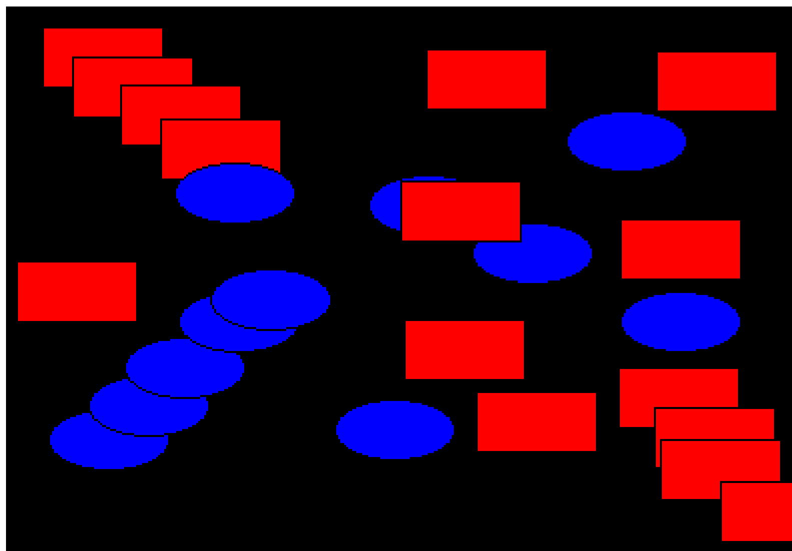
The user can hold down certain modifier keys while using the mouse. The possible modifier keys include: the Shift key, the Control key, the Alt key (called the Option key on the Mac), and the Meta key (called the Command or Apple key on the Mac). You might want to respond to a

mouse event differently when the user is holding down a modifier key. The boolean-valued instance methods `evt.isShiftDown()`, `evt.isControlDown()`, `evt.isAltDown()`, and `evt.isMetaDown()` can be called to test whether the modifier keys are pressed.

You might also want to have different responses depending on whether the user presses the left mouse button, the middle mouse button, or the right mouse button. For events triggered by a mouse button, you can determine which button was pressed or released by calling `evt.getButton()`, which returns one of the integer constants `MouseEvent.BUTTON1`, `MouseEvent.BUTTON2`, or `MouseEvent.BUTTON3` for the left, middle, and right buttons. For events such as mouseEntered and mouseExited that are not triggered by buttons, `evt.getButton()` returns `MouseEvent.NOBUTTON`.

Now, not every mouse has a middle button and a right button, and Java deals with that fact in a somewhat peculiar way. If the user clicks with the right mouse button, then `evt.isMetaDown()` will return true, even if the user was not holding down the Meta key. Similarly, if the user clicks with the middle mouse button, then `evt.isAltDown()` will return true, even if the user is not holding down the Alt/Option key. By using these functions, you can design an interface that will work even on computers that lack a middle or right mouse button. Note that there is a subtle difference between these functions and `evt.getButton()`: `evt.getButton()` really only applies to mousePressed, mouseReleased, and mouseClicked events, while `evt.isMetaDown()` and `evt.isAltDown()` are useful in any mouse event. I will often use them instead of `evt.getButton()`.

As an example, consider a *JPanel* that does the following: Clicking on the panel with the left mouse button will place a red rectangle on the panel at the point where the mouse was clicked. Clicking with the right mouse button will place a blue oval on the panel. Holding down the Shift key while clicking will clear the panel by removing all the shapes that have been placed. You can try the sample program *SimpleStamper.java*. Here is what the panel looks like after some shapes have been added:



There are several ways to write this example. There could be a separate class to handle mouse events, as in the previous example. However, in this case, I decided to let the panel itself respond to mouse events. Any object can be a mouse listener, as long as it implements the *MouseListener* interface. In this case, the panel class implements the *MouseListener* interface, so the object that represents the main panel of the program can be the mouse listener for the program. The constructor for the panel class contains the statement

```
addMouseListener(this);
```

which is equivalent to saying `this.addMouseListener(this)`. Now, the ordinary way to register a mouse listener is to say `X.addMouseListener(Y)` where `Y` is the listener and `X` is the component that will generate the mouse events. In the statement `addMouseListener(this)`, both roles are played by `this`; that is, "this object" (the panel) is generating mouse events and is also listening for those events. Although this might seem a little strange, you should get used to seeing things like this. In a large program, however, it's usually a better idea to write a separate class to do the listening in order to have a more organized division of responsibilities.

The source code for the panel class is shown below. I have included a `main()` routine to allow the class to be run as a program, as discussed in Subsection 6.2.7. You should check how the instance methods in the *MouseEvent* object are used. You can also check for the Four Steps of Event Handling ("`import java.awt.event.*`", "`implements MouseListener`", definitions for the event-handling methods, and "`addMouseListener`"):

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

/**
 * A simple demonstration of MouseEvents.  Shapes are drawn
 * on a black background when the user clicks the panel.  If
 * the user Shift-clicks, the panel is cleared.  If the user
 * right-clicks the panel, a blue oval is drawn.  Otherwise,
 * when the user clicks, a red rectangle is drawn.  The contents of
 * the panel are not persistent.  For example, they might disappear
 * if the panel is resized.
 * This class has a main() routine to allow it to be run as an
application.
 */
public class SimpleStamper extends JPanel implements MouseListener
{

    public static void main(String[] args) {
        JFrame window = new JFrame("Simple Stamper");
        SimpleStamper content = new SimpleStamper();
        window.setContentPane(content);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setLocation(120,70);
        window.setSize(450,350);
        window.setVisible(true);
    }
```

```
    // ------------------------------------------------------------
----------

   /**
    * This constructor simply sets the background color of the
panel to be black
    * and sets the panel to listen for mouse events on itself.
    */
   public SimpleStamper() {
      setBackground(Color.BLACK);
      addMouseListener(this);
   }


   /**
    *  Since this panel has been set to listen for mouse events on
itself,
    *  this method will be called when the user clicks the mouse on
the panel.
    *  This method is part of the MouseListener interface.
    */
   public void mousePressed(MouseEvent evt) {

      if ( evt.isShiftDown() ) {
            // The user was holding down the Shift key.  Just
repaint the panel.
            // Since this class does not define a paintComponent()
method, the
            // method from the superclass, JPanel, is called.  That
method simply
            // fills the panel with its background color, which is
black.  The
            // effect is to clear the panel.
         repaint();
         return;
      }

      int x = evt.getX();  // x-coordinate where user clicked.
      int y = evt.getY();  // y-coordinate where user clicked.

      Graphics g = getGraphics(); // Graphics context for drawing
directly.
                                  // NOTE: This is considered to be
bad style!

      if ( evt.isMetaDown() ) {
             // User right-clicked at the point (x,y). Draw a blue
oval centered
             // at the point (x,y). (A black outline around the
oval will make it
             // more distinct when shapes overlap.)
         g.setColor(Color.BLUE);  // Blue interior.
         g.fillOval( x - 30, y - 15, 60, 30 );
         g.setColor(Color.BLACK); // Black outline.
         g.drawOval( x - 30, y - 15, 60, 30 );
      }
```

```
         else {
               // User left-clicked (or middle-clicked) at (x,y).
               // Draw a red rectangle centered at (x,y).
            g.setColor(Color.RED);   // Red interior.
            g.fillRect( x - 30, y - 15, 60, 30 );
            g.setColor(Color.BLACK); // Black outline.
            g.drawRect( x - 30, y - 15, 60, 30 );
         }

         g.dispose();  // We are finished with the graphics context,
so dispose of it.

   } // end mousePressed();


   // The next four empty routines are required by the
MouseListener interface.
   // They don't do anything in this class, so their definitions
are empty.

   public void mouseEntered(MouseEvent evt) { }
   public void mouseExited(MouseEvent evt) { }
   public void mouseClicked(MouseEvent evt) { }
   public void mouseReleased(MouseEvent evt) { }

} // end class SimpleStamper
```

Note, by the way, that this class violates the rule that all drawing should be done in a `paintComponent()` method. The rectangles and ovals are drawn directly in the `mousePressed()` routine. To make this possible, I need to obtain a graphics context by saying "`g = getGraphics()`". After using `g` for drawing, I call `g.dispose()` to inform the operating system that I will no longer be using `g` for drawing. I do not advise doing this type of direct drawing if it can be avoided, but you can see that it does work in this case.

6.3.4 MouseMotionListeners and Dragging

Whenever the mouse is moved, it generates events. The operating system of the computer detects these events and uses them to move the mouse cursor on the screen. It is also possible for a program to listen for these "mouse motion" events and respond to them. The most common reason to do so is to implement dragging. Dragging occurs when the user moves the mouse while holding down a mouse button.

The methods for responding to mouse motion events are defined in an interface named *MouseMotionListener*. This interface specifies two event-handling methods:

```
public void mouseDragged(MouseEvent evt);
public void mouseMoved(MouseEvent evt);
```

The `mouseDragged` method is called if the mouse is moved while a button on the mouse is pressed. If the mouse is moved while no mouse button is down, then `mouseMoved` is called instead. The parameter, `evt`, is an object of type *MouseEvent*, which contains the `x` and `y` coordinates of the mouse's location, as usual. As long as the user continues to move the mouse, one of these methods will be called over and over. (So many events are generated that it would be inefficient for a program to hear them all, if it doesn't want to do anything in response. This is why the mouse motion event-handlers are defined in a separate interface from the other mouse events: You can listen for the mouse events defined in *MouseListener* without automatically hearing all mouse motion events as well.)

If you want your program to respond to mouse motion events, you must create an object that implements the *MouseMotionListener* interface, and you must register that object to listen for events. The registration is done by calling a component's `addMouseMotionListener()` method. The object will then listen for `mouseDragged` and `mouseMoved` events associated with that component. In most cases, the listener object will also implement the *MouseListener* interface so that it can respond to the other mouse events as well.

(To get a better idea of how mouse events work, you should try the sample program *SimpleTrackMouse.java*. This program responds to any of the seven different kinds of mouse events by displaying the coordinates of the mouse, the type of event, and a list of the modifier keys that are down (Shift, Control, Meta, and Alt). You can experiment with the program to see what happens as you do various things with the mouse. I also encourage you to read the source code. You should now be familiar with all the techniques that it uses.)

It is interesting to look at what a program needs to do in order to respond to dragging operations. In general, the response involves three methods: `mousePressed()`, `mouseDragged()`, and `mouseReleased()`. The dragging gesture starts when the user presses a mouse button, it continues while the mouse is dragged, and it ends when the user releases the button. This means that the programming for the response to one dragging gesture must be spread out over the three methods! Furthermore, the `mouseDragged()` method can be called many times as the mouse moves. To keep track of what is going on between one method call and the next, you need to set up some instance variables. In many applications, for example, in order to process a `mouseDragged` event, you need to remember the previous coordinates of the mouse. You can store this information in two instance variables `prevX` and `prevY` of type int. It can also be useful to save the starting coordinates, where the original `mousePressed` event occurred, in instance variables. And I suggest having a boolean variable, `dragging`, which is set to `true` while a dragging gesture is being processed. This is necessary because in many applications, not every `mousePressed` event starts a dragging operation to which you want to respond. The `mouseDragged` and `mouseReleased` methods can use the value of `dragging` to check whether a drag operation is actually in progress. You might need other instance variables as well, but in general outline, a class that handles mouse dragging looks like this:

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

public class MouseDragHandler implements MouseListener,
MouseMotionListener {
```

```
   private int startX, startY; // Point where the original
mousePress occurred.
   private int prevX, prevY;   // Most recently processed mouse
coords.
   private boolean dragging;   // Set to true when dragging is in
process.
   . . . // other instance variables for use in dragging

   public void mousePressed(MouseEvent evt) {
      if ( we-want-to-start-dragging ) {
          dragging = true;
          startX = evt.getX();  // Remember starting position.
          startY = evt.getY();
          prevX = startX;       // Remember most recent coords.
          prevY = startY;
             .
             . // Other processing.
             .
      }
   }

   public void mouseDragged(MouseEvent evt) {
       if ( dragging == false )  // First, check if we are
           return;               //   processing a dragging
gesture.
       int x = evt.getX(); // Current position of Mouse.
       int y = evt.getY();
         .
         .  // Process a mouse movement from (prevX, prevY) to
(x,y).
         .
       prevX = x;  // Remember the current position for the next
call.
       prevY = y;
   }

   public void mouseReleased(MouseEvent evt) {
       if ( dragging == false )  // First, check if we are
           return;               //   processing a dragging
gesture.
       dragging = false;  // We are done dragging.
        .
        .  // Other processing and clean-up.
        .
   }

}
```

As an example, let's look at a typical use of dragging: allowing the user to sketch a curve by dragging the mouse. This example also shows many other features of graphics and mouse processing. In the program, you can draw a curve by dragging the mouse on a large white drawing area, and you can select a color for drawing by clicking on one of several colored rectangles to the right of the drawing area. The complete source code can be found in *SimplePaint.java*. Here is a picture of the program after some drawing has been done:

I will discuss a few aspects of the source code here, but I encourage you to read it carefully in its entirety. There are lots of informative comments in the source code.

The panel for this example is designed to work for any reasonable size, that is, unless the panel is too small. This means that coordinates are computed in terms of the actual width and height of the panel. (The width and height are obtained by calling `getWidth()` and `getHeight()`.) This makes things quite a bit harder than they would be if we assumed some particular fixed size for the panel. Let's look at some of these computations in detail. For example, the large white drawing area extends from `y = 3` to `y = height - 3` vertically and from `x = 3` to `x = width - 56` horizontally. These numbers are needed in order to interpret the meaning of a mouse click. They take into account a gray border around the panel and the color palette along the right edge of the panel. The gray border is 3 pixels wide. The colored rectangles are 50 pixels wide. Together with the 3-pixel border around the panel and a 3-pixel divider between the drawing area and the colored rectangles, this adds up to put the right edge of the drawing area 56 pixels from the right edge of the panel.

A white square labeled "`CLEAR`" occupies a 50-by-50 pixel region beneath the colored rectangles on the right edge of the panel. Allowing for this square, we can figure out how much vertical space is available for the seven colored rectangles, and then divide that space by 7 to get the vertical space available for each rectangle. This quantity is represented by a variable, `colorSpace`. Out of this space, 3 pixels are used as spacing between the rectangles, so the height of each rectangle is `colorSpace - 3`. The top of the `N`-th rectangle is located `(N*colorSpace + 3)` pixels down from the top of the panel, assuming that we count the rectangles starting with zero. This is because there are `N` rectangles above the `N`-th rectangle, each of which uses `colorSpace` pixels. The extra 3 is for the border at the top of the panel. After all that, we can write down the command for drawing the `N`-th rectangle:

```
g.fillRect(width - 53, N*colorSpace + 3, 50, colorSpace - 3);
```

That was not easy! But it shows the kind of careful thinking and precision graphics that are sometimes necessary to get good results.

The mouse in this program is used to do three different things: Select a color, clear the drawing, and draw a curve. Only the third of these involves dragging, so not every mouse click will start a dragging operation. The `mousePressed()` method has to look at the `(x,y)` coordinates where the mouse was clicked and decide how to respond. If the user clicked on the `CLEAR` rectangle, the drawing area is cleared by calling `repaint()`. If the user clicked somewhere in the strip of colored rectangles, the corresponding color is selected for drawing. This involves computing which color the user clicked on, which is done by dividing the `y` coordinate by `colorSpace`. Finally, if the user clicked on the drawing area, a drag operation is initiated. In this case, a boolean variable, `dragging`, is set to `true` so that the `mouseDragged` and `mouseReleased` methods will know that a curve is being drawn. The code for this follows the general form given above. The actual drawing of the curve is done in the `mouseDragged()` method, which draws a line from the previous location of the mouse to its current location. Some effort is required to make sure that the line does not extend beyond the white drawing area of the panel. This is not automatic, since as far as the computer is concerned, the border and the color bar are part of the drawing surface. If the user drags the mouse outside the drawing area while drawing a line, the `mouseDragged()` routine changes the `x` and `y` coordinates to make them lie within the drawing area.

6.3.5 Anonymous Event Handlers and Adapter Classes

As I mentioned above, it is a fairly common practice to use anonymous inner classes to define listener objects. As discussed in Subsection 5.8.3, a special form of the `new` operator is used to create an object that belongs to an anonymous class. For example, a mouse listener object can be created with an expression of the form:

```
new MouseListener() {
   public void mousePressed(MouseEvent evt) { . . . }
   public void mouseReleased(MouseEvent evt) { . . . }
   public void mouseClicked(MouseEvent evt) { . . . }
   public void mouseEntered(MouseEvent evt) { . . . }
   public void mouseExited(MouseEvent evt) { . . . }
}
```

This is all just one long expression that both defines an unnamed class and creates an object that belongs to that class. To use the object as a mouse listener, it can be passed as the parameter to some component's `addMouseListener()` method in a command of the form:

```
component.addMouseListener( new MouseListener() {
      public void mousePressed(MouseEvent evt) { . . . }
      public void mouseReleased(MouseEvent evt) { . . . }
      public void mouseClicked(MouseEvent evt) { . . . }
      public void mouseEntered(MouseEvent evt) { . . . }
      public void mouseExited(MouseEvent evt) { . . . }
```

```
    } );
```

Now, in a typical application, most of the method definitions in this class will be empty. A class that implements an `interface` must provide definitions for all the methods in that interface, even if the definitions are empty. To avoid the tedium of writing empty method definitions in cases like this, Java provides adapter classes. An adapter class implements a listener interface by providing empty definitions for all the methods in the interface. An adapter class is useful only as a basis for making subclasses. In the subclass, you can define just those methods that you actually want to use. For the remaining methods, the empty definitions that are provided by the adapter class will be used. The adapter class *MouseAdapter* implements both the *MouseListener* interface and the *MouseMotionListener* interface, so it can be used as a basis for creating a listener for any mouse event. As an example, if you want a mouse listener that only responds to mouse-pressed events, you can use a command of the form:

```
component.addMouseListener( new MouseAdapter() {
      public void mousePressed(MouseEvent evt) { . . . }
   } );
```

To see how this works in a real example, let's write another version of the *ClickableRandomStrings* program from Subsection 6.3.2. This version uses an anonymous class based on *MouseAdapter* to handle mouse events:

```
import java.awt.Component;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.JFrame;

public class ClickableRandomStrings2 {

   public static void main(String[] args) {
      JFrame window = new JFrame("Random Strings");
      RandomStringsPanel content = new RandomStringsPanel();

      content.addMouseListener( new MouseAdapter() {
            // Register a mouse listener that is defined by an
anonymous subclass
            // of MouseAdapter.  This replaces the RepaintOnClick
class that was
            // used in the original version.
         public void mousePressed(MouseEvent evt) {
            Component source = (Component)evt.getSource();
            source.repaint();
         }
      } );

      window.setContentPane(content);
      window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      window.setLocation(100,75);
      window.setSize(300,240);
      window.setVisible(true);
   }
```

```
}
```

Anonymous inner classes can be used for other purposes besides event handling. For example, suppose that you want to define a subclass of *JPanel* to represent a drawing surface. The subclass will only be used once. It will redefine the `paintComponent()` method, but will make no other changes to *JPanel*. It might make sense to define the subclass as an anonymous inner class. You will see this pattern used in some future

Timers, KeyEvents, and State Machines

NOT EVERY EVENT is generated by an action on the part of the user. Events can also be generated by objects as part of their regular programming, and these events can be monitored by other objects so that they can take appropriate actions when the events occur. One example of this is the class `javax.swing.Timer`. A *Timer* generates events at regular intervals. These events can be used to drive an animation or to perform some other task at regular intervals. We will begin this section with a look at timer events and animation. We will then look at another type of basic user-generated event: the *KeyEvents* that are generated when the user types on the keyboard. The example at the end of the section uses both a timer and keyboard events to implement a simple game and introduces the important idea of state machines.

6.4.1 Timers and Animation

An object belonging to the class `javax.swing.Timer` exists only to generate events. A *Timer*, by default, generates a sequence of events with a fixed delay between each event and the next. (It is also possible to set a *Timer* to emit a single event after a specified time delay; in that case, the timer is being used as an "alarm.") Each event belongs to the class *ActionEvent*. An object that is to listen for the events must implement the interface *ActionListener*, which defines just one method:

```
public void actionPerformed(ActionEvent evt)
```

To use a *Timer*, you must create an object that implements the *ActionListener* interface. That is, the object must belong to a class that is declared to "`implement ActionListener`", and that class must define the `actionPerformed` method. Then, if the object is set to listen for events from the timer, the code in the listener's `actionPerformed` method will be executed every time the timer generates an event.

Since there is no point to having a timer without having a listener to respond to its events, the action listener for a timer is specified as a parameter in the timer's constructor. The time delay between timer events is also specified in the constructor. If `timer` is a variable of type *Timer*, then the statement

```
timer = new Timer( millisDelay, listener );
```

creates a timer with a delay of `millisDelay` milliseconds between events (where 1000 milliseconds equal one second). Events from the timer are sent to the `listener`. (`millisDelay` must be of type int, and `listener` must be of type *ActionListener*.) The listener's `actionPerfomed()` will be executed every time the timer emits an event. Note that a timer is not guaranteed to deliver events at precisely regular intervals. If the computer is busy with some other task, an event might be delayed or even dropped altogether.

A timer does not automatically start generating events when the timer object is created. The `start()` method in the timer must be called to tell the timer to start running. The timer's `stop()` method can be used to turn the stream of events off. It can be restarted later by calling `start()` again.

One application of timers is computer animation. A computer animation is just a sequence of still images, presented to the user one after the other. If the time between images is short, and if the change from one image to another is not too great, then the user perceives continuous motion. The easiest way to do animation in Java is to use a *Timer* to drive the animation. Each time the timer generates an event, the next frame of the animation is computed and drawn on the screen -- the code that implements this goes in the `actionPerformed` method of an object that listens for events from the timer.

Our first example of using a timer is not exactly an animation, but it does display a new image for each timer event. The program shows randomly generated images that vaguely resemble works of abstract art. In fact, the program draws a new random image every time its `paintComponent()` method is called, and the response to a timer event is simply to call `repaint()`, which in turn triggers a call to `paintComponent`. The work of the program is done in a subclass of *JPanel*, which starts like this:

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

public class RandomArtPanel extends JPanel {

   /**
    * A RepaintAction object calls the repaint method of this panel
each
    * time its actionPerformed() method is called.  An object of
this
    * type is used as an action listener for a Timer that generates
an
    * ActionEvent every four seconds.  The result is that the panel
is
    * redrawn every four seconds.
    */
   private class RepaintAction implements ActionListener {
      public void actionPerformed(ActionEvent evt) {
```

```
         repaint();  // Call the repaint() method in the panel
class.
      }
   }

   /**
    * The constructor creates a timer with a delay time of four
seconds
    * (4000 milliseconds), and with a RepaintAction object as its
    * ActionListener.  It also starts the timer running.
    */
   public RandomArtPanel() {
      RepaintAction action = new RepaintAction();
      Timer timer = new Timer(4000, action);
      timer.start();
   }

   /**
    * The paintComponent() method fills the panel with a random
shade of
    * gray and then draws one of three types of random "art".  The
type
    * of art to be drawn is chosen at random.
    */
   public void paintComponent(Graphics g) {
        .
        .   // The rest of the class is omitted
        .
```

You can find the full source code for this class in the file *RandomArt.java*. I will only note that the very short *RepaintAction* class is a natural candidate to be replaced by an anonymous inner class. That can be done where the timer is created:

```
Timer timer = new timer(4000, new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            repaint();
        }
    });
```

Later in this section, we will use a timer to drive the animation in a simple computer game.

6.4.2 Keyboard Events

In Java, user actions become events in a program. These events are associated with GUI components. When the user presses a button on the mouse, the event that is generated is associated with the component that contains the mouse cursor. What about keyboard events? When the user presses a key, what component is associated with the key event that is generated?

A GUI uses the idea of input focus to determine the component associated with keyboard events. At any given time, exactly one interface element on the screen has the input focus, and that is

where all keyboard events are directed. If the interface element happens to be a Java component, then the information about the keyboard event becomes a Java object of type *KeyEvent*, and it is delivered to any listener objects that are listening for `KeyEvents` associated with that component. The necessity of managing input focus adds an extra twist to working with keyboard events.

It's a good idea to give the user some visual feedback about which component has the input focus. For example, if the component is the typing area of a word-processor, the feedback is usually in the form of a blinking text cursor. Another possible visual clue is to draw a brightly colored border around the edge of a component when it has the input focus, as I do in the examples given later in this section.

If `comp` is any component, and you would like it to have the input focus, you can call `requestFocusInWindow()`, which should work as long as the window that contains the component is active and there is only one component that is requesting focus. In some cases, when there is only one component involved, it is is enough to call this method once, just after opening the window, and the component will retain the focus for the rest of the program. (Note that there is also a `requestFocus()` method that might work even when the window is not active, but the newer method `requestFocusInWindow()` is preferred in most cases.)

In a typical user interface, the user can choose to give the focus to a component by clicking on that component with the mouse. And pressing the tab key will often move the focus from one component to another. This is handled automatically by the components involved, without any programming on your part. However, some components do not automatically request the input focus when the user clicks on them. To solve this problem, a program can register a mouse listener with the component to detect user clicks. In response to a user click, the `mousePressed()` method should call `requestFocusInWindow()` for the component. This is true, in particular, for *JPanels* that are used as drawing surfaces, since *JPanel* objects do not receive the input focus automatically.

As our first example of processing key events, we look at a simple program in which the user moves a square up, down, left, and right by pressing arrow keys. When the user hits the 'R', 'G', 'B', or 'K' key, the color of the square is set to red, green, blue, or black, respectively. Of course, none of these key events are delivered to the panel unless it has the input focus. The panel in the program changes its appearance when it has the input focus: When it does, a cyan-colored border is drawn around the panel; when it does not, a gray-colored border is drawn. The complete source code for this example can be found in the file *KeyboardAndFocusDemo.java*. I will discuss some aspects of it below. After reading this section, you should be able to understand the source code in its entirety. I suggest running the program to see how it works.

In Java, keyboard event objects belong to a class called *KeyEvent*. An object that needs to listen for *KeyEvents* must implement the interface named *KeyListener*. Furthermore, the object must be registered with a component by calling the component's `addKeyListener()` method. The registration is done with the command "`component.addKeyListener(listener);`" where `listener` is the object that is to listen for key events, and `component` is the object that will generate the key events (when it has the input focus). It is possible for `component` and

`listener` to be the same object. All this is, of course, directly analogous to what you learned about mouse events in the previous section. The *KeyListener* interface defines the following methods, which must be included in any class that implements *KeyListener*:

```
public void keyPressed(KeyEvent evt);
public void keyReleased(KeyEvent evt);
public void keyTyped(KeyEvent evt);
```

Java makes a careful distinction between *the keys that you press* and *the characters that you type*. There are lots of keys on a keyboard: letter keys, number keys, modifier keys such as Control and Shift, arrow keys, page up and page down keys, keypad keys, function keys, and so on. In some cases, such as the shift key, pressing a key does not type a character. On the other hand, typing a character sometimes involves pressing several keys. For example, to type an uppercase 'A', you have to press the Shift key and then press the A key before releasing the Shift key. On my Mac OS computer, I can type an accented e, by holding down the Option key, pressing the E key, releasing the Option key, and pressing E again. Only one character was typed, but I had to perform three key-presses and I had to release a key at the right time. In Java, there are three types of *KeyEvent*. The types correspond to pressing a key, releasing a key, and typing a character. The `keyPressed` method is called when the user presses a key, the `keyReleased` method is called when the user releases a key, and the `keyTyped` method is called when the user types a character (whether that's done with one key press or several). Note that one user action, such as pressing the E key, can be responsible for two events, a `keyPressed` event and a `keyTyped` event. Typing an upper case 'A' can generate two `keyPressed` events, two `keyReleased` events, and one `keyTyped` event.

Usually, it is better to think in terms of two separate streams of events, one consisting of `keyPressed` and `keyReleased` events and the other consisting of `keyTyped` events. For some applications, you want to monitor the first stream; for other applications, you want to monitor the second one. Of course, the information in the `keyTyped` stream could be extracted from the `keyPressed/keyReleased` stream, but it would be difficult (and also system-dependent to some extent). Some user actions, such as pressing the Shift key, can only be detected as `keyPressed` events. I used to have a computer solitaire game that highlighted every card that could be moved, when I held down the Shift key. You can do something like that in Java by hiliting the cards when the Shift key is pressed and removing the highlight when the Shift key is released.

There is one more complication. Usually, when you hold down a key on the keyboard, that key will auto-repeat. This means that it will generate multiple `keyPressed` events with just one `keyReleased` at the end of the sequence. It can also generate multiple `keyTyped` events. For the most part, this will not affect your programming, but you should not expect every `keyPressed` event to have a corresponding `keyReleased` event.

Every key on the keyboard has an integer code number. (Actually, this is only true for keys that Java knows about. Many keyboards have extra keys that can't be used with Java.) When the `keyPressed` or `keyReleased` method is called, the parameter, `evt`, contains the code of the key that was pressed or released. The code can be obtained by calling the function

evt.getKeyCode(). Rather than asking you to memorize a table of code numbers, Java provides a named constant for each key. These constants are defined in the *KeyEvent* class. For example the constant for the shift key is KeyEvent.VK_SHIFT. If you want to test whether the key that the user pressed is the Shift key, you could say "if (evt.getKeyCode() == KeyEvent.VK_SHIFT)". The key codes for the four arrow keys are KeyEvent.VK_LEFT, KeyEvent.VK_RIGHT, KeyEvent.VK_UP, and KeyEvent.VK_DOWN. Other keys have similar codes. (The "VK" stands for "Virtual Keyboard". In reality, different keyboards use different key codes, but Java translates the actual codes from the keyboard into its own "virtual" codes. Your program only sees these virtual key codes, so it will work with various keyboards on various platforms without modification.)

In the case of a keyTyped event, you want to know which character was typed. This information can be obtained from the parameter, evt, in the keyTyped method by calling the function evt.getKeyChar(). This function returns a value of type char representing the character that was typed.

In the KeyboardAndFocusDemo program, I use the keyPressed routine to respond when the user presses one of the arrow keys. The program includes instance variables, squareLeft and squareTop, that give the position of the upper left corner of the movable square. When the user presses one of the arrow keys, the keyPressed routine modifies the appropriate instance variable and calls repaint() to redraw the panel with the square in its new position. Note that the values of squareLeft and squareTop are restricted so that the square never moves outside the white area of the panel:

```
/**
 * This is called each time the user presses a key while the panel
has
 * the input focus.  If the key pressed was one of the arrow keys,
 * the square is moved (except that it is not allowed to move off
the
 * edge of the panel, allowing for a 3-pixel border).
 */
public void keyPressed(KeyEvent evt) {

   int key = evt.getKeyCode();  // keyboard code for the pressed
key

   if (key == KeyEvent.VK_LEFT) {  // left-arrow key; move the
square left
      squareLeft -= 8;
      if (squareLeft < 3)
         squareLeft = 3;
      repaint();
   }
   else if (key == KeyEvent.VK_RIGHT) {  // right-arrow key; move
the square right
      squareLeft += 8;
      if (squareLeft > getWidth() - 3 - SQUARE_SIZE)
         squareLeft = getWidth() - 3 - SQUARE_SIZE;
      repaint();
```

```
   }
   else if (key == KeyEvent.VK_UP) {  // up-arrow key; move the
square up
      squareTop -= 8;
      if (squareTop < 3)
         squareTop = 3;
      repaint();
   }
   else if (key == KeyEvent.VK_DOWN) {  // down-arrow key; move the
square down
      squareTop += 8;
      if (squareTop > getHeight() - 3 - SQUARE_SIZE)
         squareTop = getHeight() - 3 - SQUARE_SIZE;
      repaint();
   }

}  // end keyPressed()
```

Color changes -- which happen when the user types the characters 'R', 'G', 'B', and 'K', or the lower case equivalents -- are handled in the keyTyped method. I won't include it here, since it is so similar to the keyPressed method. Finally, to complete the *KeyListener* interface, the keyReleased method must be defined. In the sample program, the body of this method is empty since the program does nothing in response to keyReleased events.

6.4.3 Focus Events

If a component is to change its appearance when it has the input focus, it needs some way to know when it has the focus. In Java, objects are notified about changes of input focus by events of type *FocusEvent*. An object that wants to be notified of changes in focus can implement the *FocusListener* interface. This interface declares two methods:

```
public void focusGained(FocusEvent evt);
public void focusLost(FocusEvent evt);
```

Furthermore, the addFocusListener() method must be used to set up a listener for the focus events. When a component gets the input focus, it calls the focusGained() method of any registered with *FocusListener*. When it loses the focus, it calls the listener's focusLost() method.

In the sample KeyboardAndFocusDemo program, the response to a focus event is simply to redraw the panel. The paintComponent() method checks whether the panel has the input focus by calling the boolean-valued function hasFocus(), which is defined in the *Component* class, and it draws a different picture depending on whether or not the panel has the input focus. The net result is that the appearance of the panel changes when the panel gains or loses focus. The methods from the *FocusListener* interface are defined simply as:

```
public void focusGained(FocusEvent evt) {
```

```
        // The panel now has the input focus.
    repaint();  // will redraw with a new message and a cyan border
}

public void focusLost(FocusEvent evt) {
       // The panel has now lost the input focus.
    repaint();  // will redraw with a new message and a gray border
}
```

The other aspect of handling focus is to make sure that the panel actually gets the focus. In this case, I called `requestFocusInWindow()` for the panel in the program's `main()` routine, just after opening the window. This approach works because there is only one component in the window, and it should have focus as long as the window is active. If the user clicks over to another window while using the program, the window becomes inactive and the panel loses focus temporarily, but gets is back when the user clicks back to the program window.

There are still decisions to be made about the overall structure of the program. In this case, I decided to use a nested class named *Listener* to define an object that listens for both focus and key events. In the constructor for the panel, I create an object of type *Listener* and register it to listen for both key events and focus events from the panel. See the *source code* for full details.

6.4.4 State Machines

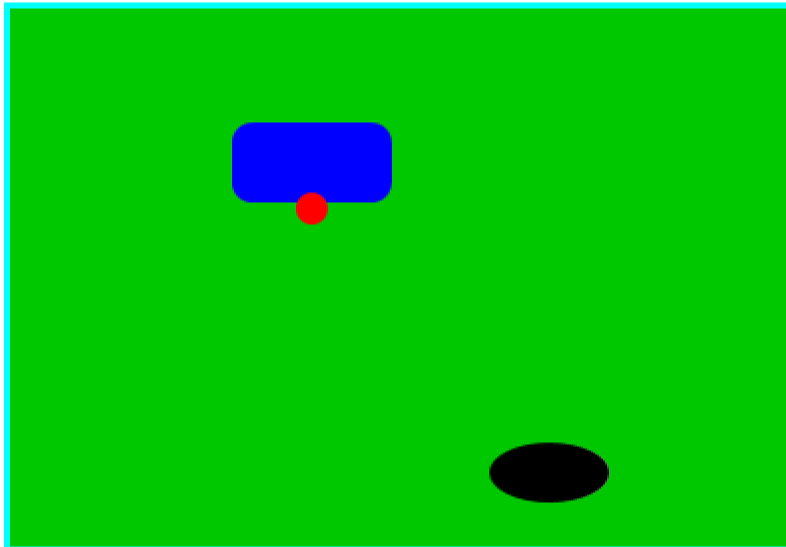
The information stored in an object's instance variables is said to represent the state of that object. When one of the object's methods is called, the action taken by the object can depend on its state. (Or, in the terminology we have been using, the definition of the method can look at the instance variables to decide what to do.) Furthermore, the state can change. (That is, the definition of the method can assign new values to the instance variables.) In computer science, there is the idea of a state machine, which is just something that has a state and can change state in response to events or inputs. The response of a state machine to an event depends on what state it's in when the event occurs. An object is a kind of state machine. Sometimes, this point of view can be very useful in designing classes.

The state machine point of view can be especially useful in the type of event-oriented programming that is required by graphical user interfaces. When designing a GUI program, you can ask yourself: What information about state do I need to keep track of? What events can change the state of the program? How will my response to a given event depend on the current state? Should the appearance of the GUI be changed to reflect a change in state? How should the `paintComponent()` method take the state into account? All this is an alternative to the top-down, step-wise-refinement style of program design, which does not apply to the overall design of an event-oriented program.

In the *KeyboardAndFocusDemo* program, shown above, the state of the program is recorded in the instance variables `squareColor`, `squareLeft`, and `squareTop`. These state variables

are used in the `paintComponent()` method to decide how to draw the panel. Their values are changed in the two key-event-handling methods.

In the rest of this section, we'll look at another example, where the state plays an even bigger role. In this example, the user plays a simple arcade-style game by pressing the arrow keys. The program is defined in the source code file *SubKiller.java*. As usual, it would be a good idea to compile and run the program as well as read the full source code. Here is a picture:



The program shows a black "submarine" near the bottom of the panel. While the panel has the input focus, this submarine moves back and forth erratically near the bottom. Near the top, there is a blue "boat." You can move this boat back and forth by pressing the left and right arrow keys. Attached to the boat is a red "bomb" (or "depth charge"). You can drop the bomb by hitting the down arrow key. The objective is to blow up the submarine by hitting it with the bomb. If the bomb falls off the bottom of the screen, you get a new one. If the submarine explodes, a new sub is created and you get a new bomb. Try it! Make sure to hit the sub at least once, so you can see the explosion.

Let's think about how this game can be programmed. First of all, since we are doing object-oriented programming, I decided to represent the boat, the depth charge, and the submarine as objects. Each of these objects is defined by a separate nested class inside the main panel class, and each object has its own state which is represented by the instance variables in the corresponding class. I use variables `boat`, `bomb`, and `sub` in the panel class to refer to the boat, bomb, and submarine objects.

Now, what constitutes the "state" of the program? That is, what things change from time to time and affect the appearance or behavior of the program? Of course, the state includes the positions of the boat, submarine, and bomb, so those objects have instance variables to store the positions. Anything else, possibly less obvious? Well, sometimes the bomb is falling, and sometimes it's not. That is a difference in state. Since there are two possibilities, I represent this aspect of the state with a boolean variable in the `bomb` object, `bomb.isFalling`. Sometimes the

submarine is moving left and sometimes it is moving right. The difference is represented by another boolean variable, `sub.isMovingLeft`. Sometimes, the sub is exploding. This is also part of the state, and it is represented by a boolean variable, `sub.isExploding`. However, the explosions require a little more thought. An explosion is something that takes place over a series of frames. While an explosion is in progress, the sub looks different in each frame, as the size of the explosion increases. Also, I need to know when the explosion is over so that I can go back to moving and drawing the sub as usual. So, I use an integer variable, `sub.explosionFrameNumber` to record how many frames have been drawn since the explosion started; the value of this variable is used only when an explosion is in progress.

How and when do the values of these state variables change? Some of them seem to change on their own: For example, as the sub moves left and right, the state variables that specify its position change. Of course, these variables are changing because of an animation, and that animation is driven by a timer. Each time an event is generated by the timer, some of the state variables have to change to get ready for the next frame of the animation. The changes are made by the action listener that listens for events from the timer. The `boat`, `bomb`, and `sub` objects each contain an `updateForNextFrame()` method that updates the state variables of the object to get ready for the next frame of the animation. The action listener for the timer calls these methods with the statements

```
boat.updateForNewFrame();
bomb.updateForNewFrame();
sub.updateForNewFrame();
```

The action listener also calls `repaint()`, so that the panel will be redrawn to reflect its new state. There are several state variables that change in these update methods, in addition to the position of the sub: If the bomb is falling, then its y-coordinate increases from one frame to the next. If the bomb hits the sub, then the `isExploding` variable of the sub changes to true, and the `isFalling` variable of the bomb becomes `false`. The `isFalling` variable also becomes false when the bomb falls off the bottom of the screen. If the sub is exploding, then its `explosionFrameNumber` increases from one frame to the next, and when it reaches a certain value, the explosion ends and `isExploding` is reset to false. At random times, the sub switches between moving to the left and moving to the right. Its direction of motion is recorded in the sub's `isMovingLeft` variable. The sub's `updateForNewFrame()` method includes these lines to change the value of `isMovingLeft` at random times:

```
if ( Math.random() < 0.04 )
   isMovingLeft = ! isMovingLeft;
```

There is a 1 in 25 chance that `Math.random()` will be less than 0.04, so the statement "`isMovingLeft = ! isMovingLeft`" is executed in one in every twenty-five frames, on average. The effect of this statement is to reverse the value of `isMovingLeft`, from false to true or from true to false. That is, the direction of motion of the sub is reversed.

In addition to changes in state that take place from one frame to the next, a few state variables change when the user presses certain keys. In the program, this is checked in a method that

responds to user keystrokes. If the user presses the left or right arrow key, the position of the boat is changed. If the user presses the down arrow key, the bomb changes from not-falling to falling. This is coded in the `keyPressed()` method of a *KeyListener* that is registered to listen for key events on the panel; that method reads as follows:

```
public void keyPressed(KeyEvent evt) {
   int code = evt.getKeyCode();  // which key was pressed.
   if (code == KeyEvent.VK_LEFT) {
        // Move the boat left.  (If this moves the boat out of the
frame, its
        // position will be adjusted in the
boat.updateForNewFrame() method.)
      boat.centerX -= 15;
   }
   else if (code == KeyEvent.VK_RIGHT) {
        // Move the boat right.  (If this moves boat out of the
frame, its
        // position will be adjusted in the
boat.updateForNewFrame() method.)
      boat.centerX += 15;
   }
   else if (code == KeyEvent.VK_DOWN) {
         // Start the bomb falling, if it is not already falling.
      if ( bomb.isFalling == false )
         bomb.isFalling = true;
   }
}
```

Note that it's not necessary to call `repaint()` in this method, since this panel shows an animation that is constantly being redrawn anyway. Any changes in the state will become visible to the user as soon as the next frame is drawn. At some point in the program, I have to make sure that the user does not move the boat off the screen. I could have done this in `keyPressed()`, but I choose to check for this in another routine, in the boat object.

The program uses four listeners, to respond to action events from the timer, key events from the user, focus events, and mouse events. In this program, the user must click the panel to start the game. The game is programmed to run as long as the panel has the input focus. In this example, the program does not automatically request the focus; the user has to do it. When the user clicks the panel, the mouse listener requests the input focus and the game begins. The timer runs only when the panel has the input focus; this is programmed by having the focus listener start the timer when the panel gains the input focus and stop the timer when the panel loses the input focus. All four listeners are created in the constructor of the *SubKillerPanel* class using anonymous inner classes. (See Subsection 6.3.5.)

I encourage you to read the source code in *SubKiller.java*. Although a few points are tricky, you should with some effort be able to read and understand the entire program. Try to understand the program in terms of state machines. Note how the state of each of the three objects in the program changes in response to events from the timer and from the user.

While it's not at all sophisticated as arcade games go, the SubKiller game does use some interesting programming. And it nicely illustrates how to apply state-machine thinking in event-oriented programming.

Basic Components

IN PRECEDING SECTIONS, you've seen how to use a graphics context to draw on the screen and how to handle mouse events and keyboard events. In one sense, that's all there is to GUI programming. If you're willing to program all the drawing and handle all the mouse and keyboard events, you have nothing more to learn. However, you would either be doing a lot more work than you need to do, or you would be limiting yourself to very simple user interfaces. A typical user interface uses standard GUI components such as buttons, scroll bars, text-input boxes, and menus. These components have already been written for you, so you don't have to duplicate the work involved in developing them. They know how to draw themselves, and they can handle the details of processing the mouse and keyboard events that concern them.

Consider one of the simplest user interface components, a push button. The button has a border, and it displays some text. This text can be changed. Sometimes the button is disabled, so that clicking on it doesn't have any effect. When it is disabled, its appearance changes. When the user clicks on the push button, the button changes appearance while the mouse button is pressed and changes back when the mouse button is released. In fact, it's more complicated than that. If the user moves the mouse outside the push button before releasing the mouse button, the button changes to its regular appearance. To implement this, it is necessary to respond to mouse exit or mouse drag events. Furthermore, on many platforms, a button can receive the input focus. The button changes appearance when it has the focus. If the button has the focus and the user presses the space bar, the button is triggered. This means that the button must respond to keyboard and focus events as well.

Fortunately, you don't have to program **any** of this, provided you use an object belonging to the standard class `javax.swing.JButton`. A *JButton* object draws itself and processes mouse, keyboard, and focus events on its own. You only hear from the *JButton* when the user triggers it by clicking on it or pressing the space bar while the button has the input focus. When this happens, the *JButton* object creates an event object belonging to the class `java.awt.event.ActionEvent`. The event object is sent to any registered listeners to tell them that the button has been pushed. Your program gets only the information it needs -- the fact that a button was pushed.

The standard components that are defined as part of the Swing graphical user interface API are defined by subclasses of the class *JComponent*, which is itself a subclass of *Component*. (Note that this includes the *JPanel* class that we have already been working with extensively.) Many useful methods are defined in the *Component* and *JComponent* classes and so can be used with any Swing component. We begin by looking at a few of these methods. Suppose that `comp` is a variable that refers to some *JComponent*. Then the following methods can be used:

- `comp.getWidth()` and `comp.getHeight()` are functions that give the current size of the component, in pixels. One warning: When a component is first created, its size is zero. The size will be set later, probably by a layout manager. A common mistake is to check the size of a component before that size has been set, such as in a constructor.
- `comp.setEnabled(true)` and `comp.setEnabled(false)` can be used to enable and disable the component. When a component is disabled, its appearance might change, and the user cannot do anything with it. There is a boolean-valued function, `comp.isEnabled()` that you can call to discover whether the component is enabled.
- `comp.setVisible(true)` and `comp.setVisible(false)` can be called to hide or show the component.
- `comp.setFont(font)` sets the font that is used for text displayed on the component. See Subsection 6.2.3 for a discussion of fonts.
- `comp.setBackground(color)` and `comp.setForeground(color)` set the background and foreground colors for the component. See Subsection 6.2.2.
- `comp.setOpaque(true)` tells the component that the area occupied by the component should be filled with the component's background color before the content of the component is painted. By default, only *JLabels* are non-opaque. A non-opaque, or "transparent", component ignores its background color and simply paints its content over the content of its container. This usually means that it inherits the background color from its container.
- `comp.setToolTipText(string)` sets the specified string as a "tool tip" for the component. The tool tip is displayed if the mouse cursor is in the component and the mouse is not moved for a few seconds. The tool tip should give some information about the meaning of the component or how to use it.
- `comp.setPreferredSize(size)` sets the size at which the component should be displayed, if possible. The parameter is of type `java.awt.Dimension`, where an object of type *Dimension* has two public integer-valued instance variables, `width` and `height`. A call to this method usually looks something like "`setPreferredSize( new Dimension(100,50) )`". The preferred size is used as a hint by layout managers, but will not be respected in all cases. Standard components generally compute a correct preferred size automatically, but it can be useful to set it in some cases. For example, if you use a *JPanel* as a drawing surface, it is usually a good idea to set a preferred size for it, since its default preferred size is zero.

Note that using any component is a multi-step process. The component object must be created with a constructor. It must be added to a container. In many cases, a listener must be registered to respond to events from the component. And in some cases, a reference to the component must be saved in an instance variable so that the component can be manipulated by the program after it has been created. In this section, we will look at a few of the basic standard components that are available in Swing. In the next section we will consider the problem of laying out components in containers.

6.5.1 JButton

An object of class *JButton* is a push button that the user can click to trigger some action. You've already seen buttons used in Section 6.1, but we consider them in much more detail here. To use

any component effectively, there are several aspects of the corresponding class that you should be familiar with. For *JButton*, as an example, I list these aspects explicitly:

- **Constructors**: The *JButton* class has a constructor that takes a string as a parameter. This string becomes the text displayed on the button. For example: `stopGoButton = new JButton("Go")`. This creates a button object that will display the text, "Go" (but remember that the button must still be added to a container before it can appear on the screen).
- **Events**: When the user clicks on a button, the button generates an event of type *ActionEvent*. This event is sent to any listener that has been registered with the button as an *ActionListener*.
- **Listeners**: An object that wants to handle events generated by buttons must implement the *ActionListener* interface. This interface defines just one method, "`public void actionPerformed(ActionEvent evt)`", which is called to notify the object of an action event.
- **Registration of Listeners**: In order to actually receive notification of an event from a button, an *ActionListener* must be registered with the button. This is done with the button's `addActionListener()` method. For example: `stopGoButton.addActionListener( buttonHandler );`
- **Event methods**: When `actionPerformed(evt)` is called by the button, the parameter, `evt`, contains information about the event. This information can be retrieved by calling methods in the *ActionEvent* class. In particular, `evt.getActionCommand()` returns a *String* giving the command associated with the button. By default, this command is the text that is displayed on the button, but it is possible to set it to some other string. The method `evt.getSource()` returns a reference to the object that produced the event, that is, to the *JButton* that was pressed. The return value is of type *Object*, not *JButton*, because other types of components can also produce *ActionEvents*.
- **Component methods**: Several useful methods are defined in the *JButton* class, in addition to the standard *Component* methods. For example, `stopGoButton.setText("Stop")` changes the text displayed on the button to "Stop". And `stopGoButton.setActionCommand("sgb")` changes the action command associated with this button for action events. The `setEnabled()` and `setText()` methods are particularly useful for giving the user information about what is going on in the program. A disabled button is better than a button that gives an obnoxious error message such as "Sorry, you can't click on me now!"

6.5.2 JLabel

JLabel is certainly the simplest type of component. An object of type *JLabel* exists just to display a line of text. The text cannot be edited by the user, although it can be changed by your program. The constructor for a *JLabel* specifies the text to be displayed:

```
JLabel message = new JLabel("Hello World!");
```

There is another constructor that specifies where in the label the text is located, if there is extra space. The possible alignments are given by the constants `JLabel.LEFT`, `JLabel.CENTER`, and `JLabel.RIGHT`. For example,

```
JLabel message = new JLabel("Hello World!", JLabel.CENTER);
```

creates a label whose text is centered in the available space. You can change the text displayed in a label by calling the label's `setText()` method:

```
message.setText("Goodbye World!");
```

Since the *JLabel* class is a subclass of *JComponent*, you can use methods such as `setForeground()` and `setFont()` with labels. If you want the background color to have any effect, you should call `setOpaque(true)` on the label, since otherwise the *JLabel* might not fill in its background. For example:

```
JLabel message = new JLabel("Hello World!", JLabel.CENTER);
message.setForeground(Color.RED);   // Display red text...
message.setBackground(Color.BLACK); //    on a black background...
message.setFont(new Font("Serif",Font.BOLD,18));  // in a big bold
font.
message.setOpaque(true);  // Make sure background is filled in.
```

6.5.3 JCheckBox

A *JCheckBox* is a component that has two states: selected or unselected. The user can change the state of a check box by clicking on it. The state of a checkbox is represented by a boolean value that is `true` if the box is selected and is `false` if the box is unselected. A checkbox has a label, which is specified when the box is constructed:

```
JCheckBox showTime = new JCheckBox("Show Current Time");
```

Usually, it's the user who sets the state of a *JCheckBox*, but you can also set the state programmatically. The current state of a checkbox is set using its `setSelected(boolean)` method. For example, if you want the checkbox `showTime` to be checked, you would say "`showTime.setSelected(true)`". To uncheck the box, say "`showTime.setSelected(false)`". You can determine the current state of a checkbox by calling its `isSelected()` method, which returns a boolean value.

In many cases, you don't need to worry about events from checkboxes. Your program can just check the state whenever it needs to know it by calling the `isSelected()` method. However, a checkbox does generate an event when its state is changed by the user, and you can detect this event and respond to it if you want something to happen at the moment the state changes. When the state of a checkbox is changed by the user, it generates an event of type *ActionEvent*. If you want something to happen when the user changes the state, you must register an *ActionListener* with the checkbox by calling its `addActionListener()` method. (Note that if you change

the state by calling the `setSelected()` method, no *ActionEvent* is generated. However, there is another method in the *JCheckBox* class, `doClick()`, which simulates a user click on the checkbox and does generate an *ActionEvent*.)

When handling an *ActionEvent*, you can call `evt.getSource()` in the `actionPerformed()` method to find out which object generated the event. (Of course, if you are only listening for events from one component, you don't have to do this.) The returned value is of type `Object`, but you can type-cast it to another type if you want. Once you know the object that generated the event, you can ask the object to tell you its current state. For example, if you know that the event had to come from one of two checkboxes, `cb1` or `cb2`, then your `actionPerformed()` method might look like this:

```
public void actionPerformed(ActionEvent evt) {
   Object source = evt.getSource();
   if (source == cb1) {
      boolean newState = cb1.isSelected();
      ... // respond to the change of state
   }
   else if (source == cb2) {
      boolean newState = cb2.isSelected();
      ... // respond to the change of state
   }
}
```

Alternatively, you can use `evt.getActionCommand()` to retrieve the action command associated with the source. For a *JCheckBox*, the action command is, by default, the label of the checkbox.

6.5.4 JTextField and JTextArea

The *JTextField* and *JTextArea* classes represent components that contain text that can be edited by the user. A *JTextField* holds a single line of text, while a *JTextArea* can hold multiple lines. It is also possible to set a *JTextField* or *JTextArea* to be read-only so that the user can read the text that it contains but cannot edit the text. Both classes are subclasses of an abstract class, *JTextComponent*, which defines their common properties.

JTextField and *JTextArea* have many methods in common. The instance method `setText()`, which takes a parameter of type *String*, can be used to change the text that is displayed in an input component. The contents of the component can be retrieved by calling its `getText()` instance method, which returns a value of type *String*. If you want to stop the user from modifying the text, you can call `setEditable(false)`. Call the same method with a parameter of `true` to make the input component user-editable again.

The user can only type into a text component when it has the input focus. The user can give the input focus to a text component by clicking it with the mouse, but sometimes it is useful to give

the input focus to a text field programmatically. You can do this by calling its `requestFocusInWindow()` method. For example, when I discover an error in the user's input, I usually call `requestFocusInWindow()` on the text field that contains the error. This helps the user see where the error occurred and lets the user start typing the correction immediately.

By default, there is no space between the text in a text component and the edge of the component, which usually doesn't look very good. You can use the `setMargin()` method of the component to add some blank space between the edge of the component and the text. This method takes a parameter of type `java.awt.Insets` which contains four integer instance variables that specify the margins on the top, left, bottom, and right edge of the component. For example,

```
textComponent.setMargin( new Insets(5,5,5,5) );
```

adds a five-pixel margin between the text in `textComponent` and each edge of the component.

The *JTextField* class has a constructor

```
public JTextField(int columns)
```

where `columns` is an integer that specifies the number of characters that should be visible in the text field. This is used to determine the preferred width of the text field. (Because characters can be of different sizes and because the preferred width is not always respected, the actual number of characters visible in the text field might not be equal to `columns`.) You don't have to specify the number of columns; for example, you might use the text field in a context where it will expand to fill whatever space is available. In that case, you can use the default constructor `JTextField()`, with no parameters. You can also use the following constructors, which specify the initial contents of the text field:

```
public JTextField(String contents);
public JTextField(String contents, int columns);
```

The constructors for a *JTextArea* are

```
public JTextArea()
public JTextArea(int rows, int columns)
public JTextArea(String contents)
public JTextArea(String contents, int rows, int columns)
```

The parameter `rows` specifies how many lines of text should be visible in the text area. This determines the preferred height of the text area, just as `columns` determines the preferred width. However, the text area can actually contain any number of lines; the text area can be scrolled to reveal lines that are not currently visible. It is common to use a *JTextArea* as the

CENTER component of a BorderLayout. In that case, it is less useful to specify the number of lines and columns, since the TextArea will expand to fill all the space available in the center area of the container.

The *JTextArea* class adds a few useful methods to those inherited from *JTextComponent*. For example, the instance method `append(moreText)`, where `moreText` is of type *String*, adds the specified text at the end of the current content of the text area. (When using `append()` or `setText()` to add text to a *JTextArea*, line breaks can be inserted in the text by using the newline character, `'\n'`.) And `setLineWrap(wrap)`, where `wrap` is of type boolean, tells what should happen when a line of text is too long to be displayed in the text area. If `wrap` is true, then any line that is too long will be "wrapped" onto the next line; if `wrap` is false, the line will simply extend outside the text area, and the user will have to scroll the text area horizontally to see the entire line. The default value of `wrap` is false.

Since it might be necessary to scroll a text area to see all the text that it contains, you might expect a text area to come with scroll bars. Unfortunately, this does not happen automatically. To get scroll bars for a text area, you have to put the *JTextArea* inside another component, called a *JScrollPane*. This can be done as follows:

```
JTextArea inputArea = new JTextArea();
JScrollPane scroller = new JScrollPane( inputArea );
```

The scroll pane provides scroll bars that can be used to scroll the text in the text area. The scroll bars will appear only when needed, that is when the size of the text exceeds the size of the text area. Note that when you want to put the text area into a container, you should add the scroll pane, not the text area itself, to the container. See the program *TextAreaDemo.java* for a very short example of using a text area in a scroll pane.

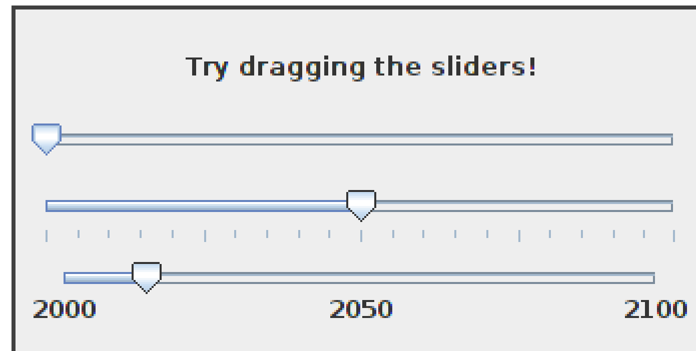
When the user is typing in a *JTextField* and presses return, an *ActionEvent* is generated. If you want to respond to such events, you can register an *ActionListener* with the text field, using the text field's `addActionListener()` method. (Since a *JTextArea* can contain multiple lines of text, pressing return in a text area does not generate an event; it simply begins a new line of text.)

JTextField has a subclass, *JPasswordField*, which is identical except that it does not reveal the text that it contains. The characters in a *JPasswordField* are all displayed as asterisks (or some other fixed character). A password field is, obviously, designed to let the user enter a password without showing that password on the screen.

Text components are actually quite complex, and I have covered only their most basic properties here. I will return to the topic of text components in Chapter 13.

6.5.5 JSlider

A *JSlider* provides a way for the user to select an integer value from a range of possible values. The user does this by dragging a "knob" along a bar. A slider can, optionally, be decorated with tick marks and with labels. This picture, from the sample program *SliderDemo.java*, shows three sliders with different decorations and with different ranges of values:



Here, the second slider is decorated with tick marks, and the third one is decorated with labels. It's possible for a single slider to have both types of decorations.

The most commonly used constructor for *JSliders* specifies the start and end of the range of values for the slider and its initial value when it first appears on the screen:

```
public JSlider(int minimum, int maximum, int value)
```

If the parameters are omitted, the values 0, 100, and 50 are used. By default, a slider is horizontal, but you can make it vertical by calling its method `setOrientation(JSlider.VERTICAL)`. The current value of a *JSlider* can be read at any time with its `getValue()` method, which returns a value of type int. If you want to change the value, you can do so with the method `setValue(n)`, which takes a parameter of type int.

If you want to respond immediately when the user changes the value of a slider, you can register a listener with the slider. `JSliders`, unlike other components we have seen, do not generate `ActionEvents`. Instead, they generate events of type *ChangeEvent*. *ChangeEvent* and related classes are defined in the package `javax.swing.event` rather than `java.awt.event`, so if you want to use `ChangeEvents`, you should `import javax.swing.event.*` at the beginning of your program. You must also define some object to implement the *ChangeListener* interface, and you must register the change listener with the slider by calling its `addChangeListener()` method. A *ChangeListener* must provide a definition for the method:

```
public void stateChanged(ChangeEvent evt)
```

This method will be called whenever the value of the slider changes. Note that it will be called when you change the value with the `setValue()` method, as well as when the user changes the value. In the `stateChanged()` method, you can call `evt.getSource()` to find out which object generated the event. If you want to know whether the user generated the change

event, call the slider's `getValueIsAdjusting()` method, which returns true if the user is dragging the knob on the slider.

Using tick marks on a slider is a two-step process: Specify the interval between the tick marks, and tell the slider that the tick marks should be displayed. There are actually two types of tick marks, "major" tick marks and "minor" tick marks. You can have one or the other or both. Major tick marks are a bit longer than minor tick marks. The method `setMinorTickSpacing(i)` indicates that there should be a minor tick mark every `i` units along the slider. The parameter is an integer. (The spacing is in terms of values on the slider, not pixels.) For the major tick marks, there is a similar command, `setMajorTickSpacing(i)`. Calling these methods is not enough to make the tick marks appear. You also have to call `setPaintTicks(true)`. For example, the second slider in the above illustration was created and configured using the commands:

```
slider2 = new JSlider();  // (Uses default min, max, and value.)
slider2.addChangeListener(this);
slider2.setMajorTickSpacing(25);
slider2.setMinorTickSpacing(5);
slider2.setPaintTicks(true);
```

Labels on a slider are handled similarly. You have to specify the labels and tell the slider to paint them. Specifying labels is a tricky business, but the *JSlider* class has a method to simplify it. You can create a set of labels and add them to a slider named `sldr` with the command:

```
sldr.setLabelTable( sldr.createStandardLabels(i) );
```

where `i` is an integer giving the spacing between the labels. To arrange for the labels to be displayed, call `setPaintLabels(true)`. For example, the third slider in the above illustration was created and configured with the commands:

```
slider3 = new JSlider(2000,2100,2014);
slider3.addChangeListener(this);
slider3.setLabelTable( slider3.createStandardLabels(50) );
slider3.setPaintLabels(true);
```

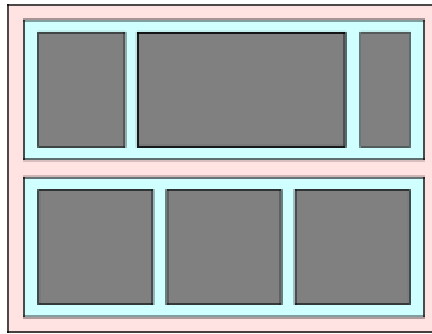
Basic Layout

COMPONENTS are the fundamental building blocks of a graphical user interface. But you have to do more with components besides create them. Another aspect of GUI programming is laying out components on the screen, that is, deciding where they are drawn and how big they are. You have probably noticed that computing coordinates can be a difficult problem, especially if you don't assume a fixed size for the drawing area. Java has a solution for this, as well.

Components are the visible objects that make up a GUI. Some components are containers, which can hold other components. Containers in Java are objects that belong to some subclass of `java.awt.Container`. The content pane of a *JFrame* is an example of a container. The

standard class *JPanel*, which we have mostly used as a drawing surface up until now, is another example of a container.

Because a *JPanel* object is a container, it can hold other components. Because a *JPanel* is itself a component, you can add a *JPanel* to another *JPanel*. This makes complex nesting of components possible. *JPanels* can be used to organize complicated user interfaces, as shown in this illustration:



In this picture, a large panel holds two smaller panels. Each of the two smaller panels in turn holds three components.

The components in a container must be "laid out," which means setting their sizes and positions. It's possible to program the layout yourself, but layout is ordinarily done by a layout manager. A layout manager is an object associated with a container that implements some policy for laying out the components in that container. Different types of layout manager implement different policies. In this section, we will cover the three most common types of layout manager, and then we will look at several programming examples that use components and layout.

Every container has a default layout manager and has an instance method, `setLayout()`, that takes a parameter of type *LayoutManager* and that is used to specify a different layout manager for the container. Components are added to a container by calling an instance method named `add()` in the container object. There are actually several versions of the `add()` method, with different parameter lists. Different versions of `add()` are appropriate for different layout managers, as we will see below.

6.6.1 Basic Layout Managers

Java has a variety of standard layout managers that can be used as parameters in the `setLayout()` method. They are defined by classes in the package `java.awt`. Here, we will look at just three of these layout manager classes: *FlowLayout*, *BorderLayout*, and *GridLayout*.

A *FlowLayout* simply lines up components in a row across the container. The size of each component is equal to that component's "preferred size." After laying out as many items as will fit in a row across the container, the layout manager will move on to the next row. The default

layout for a *JPanel* is a *FlowLayout*; that is, a *JPanel* uses a *FlowLayout* unless you specify a different layout manager by calling the panel's `setLayout()` method.

The components in a given row can be either left-aligned, right-aligned, or centered within that row, and there can be horizontal and vertical gaps between components. If the default constructor, "`new FlowLayout()`", is used, then the components on each row will be centered and both the horizontal and the vertical gaps will be five pixels. The constructor

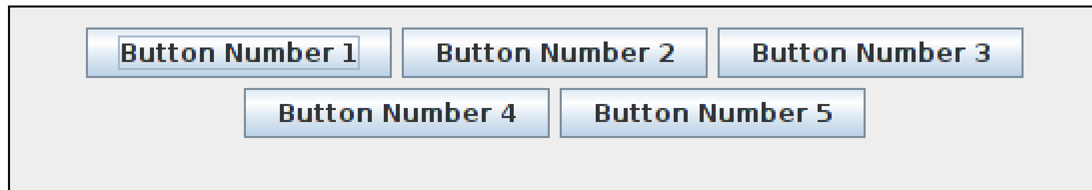
```
public FlowLayout(int align, int hgap, int vgap)
```

can be used to specify alternative alignment and gaps. The possible values of `align` are `FlowLayout.LEFT`, `FlowLayout.RIGHT`, and `FlowLayout.CENTER`.

Suppose that `container` is a container object that is using a *FlowLayout* as its layout manager. Then, a component, `comp`, can be added to the container with the statement

```
container.add(comp);
```

The *FlowLayout* will line up all the components that have been added to the container in this way. They will be lined up in the order in which they were added. For example, this picture shows five buttons in a panel that uses a *FlowLayout*:



Note that since the five buttons will not fit in a single row across the panel, they are arranged in two rows. In each row, the buttons are grouped together and are centered in the row. The buttons were added to the panel using the statements:

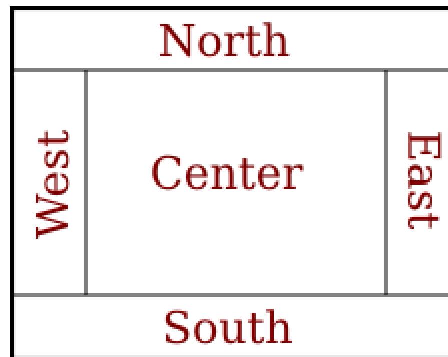
```
panel.add(button1);
panel.add(button2);
panel.add(button3);
panel.add(button4);
panel.add(button5);
```

When a container uses a layout manager, the layout manager is ordinarily responsible for computing the preferred size of the container (although a different preferred size could be set by calling the container's `setPreferredSize` method). A *FlowLayout* prefers to put its components in a single row, so the preferred width is the total of the preferred widths of all the components, plus the horizontal gaps between the components. The preferred height is the maximum preferred height of all the components.

A *BorderLayout* layout manager is designed to display one large, central component, with up to four smaller components arranged around the edges of the central component. If a container, `cntr`, is using a *BorderLayout*, then a component, `comp`, should be added to the container using a statement of the form

```
cntr.add( comp, borderLayoutPosition );
```

where `borderLayoutPosition` specifies what position the component should occupy in the layout and is given as one of the constants `BorderLayout.CENTER`, `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, or `BorderLayout.WEST`. The meaning of the five positions is shown in this diagram:



Note that a border layout can contain fewer than five components, so that not all five of the possible positions need to be filled. It would be very unusual, however, to have no center component.

A *BorderLayout* sets the sizes of its components as follows: The `NORTH` and `SOUTH` components (if present) are shown at their preferred heights, but their width is set equal to the full width of the container. The `EAST` and `WEST` components are shown at their preferred widths, but their height is set to the height of the container, minus the space occupied by the `NORTH` and `SOUTH` components. Finally, the `CENTER` component takes up any remaining space. The preferred size of the `CENTER` component is ignored when the layout is done, but it is taken into account when the preferred size of the container as a whole is computed. You should make sure that the components that you put into a *BorderLayout* are suitable for the positions that they will occupy. A horizontal slider or text field, for example, would work well in the `NORTH` or `SOUTH` position, but wouldn't make much sense in the `EAST` or `WEST` position.

The default constructor, `new BorderLayout()`, leaves no space between components. If you would like to leave some space, you can specify horizontal and vertical gaps in the constructor of the `BorderLayout` object. For example, if you say

```
panel.setLayout(new BorderLayout(5,7));
```

then the layout manager will insert horizontal gaps of 5 pixels between components and vertical gaps of 7 pixels between components. The background color of the container will show through

in these gaps. The default layout for the original content pane that comes with a *JFrame* is a `BorderLayout` with no horizontal or vertical gap.

Finally, we consider the *GridLayout* layout manager. A grid layout lays out components in a grid containing rows and columns of equal sized rectangles. This illustration shows how the components would be arranged in a grid layout with 4 rows and 3 columns:

#1	#2	#3
#4	#5	#6
#7	#8	#9
#10	#11	#12

If a container uses a *GridLayout*, the appropriate `add` method for the container takes a single parameter of type *Component* (for example: `cntr.add(comp)`). Components are added to the grid in the order shown; that is, each row is filled from left to right before going on the next row.

The constructor for a *GridLayout* takes the form "`new GridLayout(R,C)`", where `R` is the number of rows and `C` is the number of columns. If you want to leave horizontal gaps of `H` pixels between columns and vertical gaps of `V` pixels between rows, use "`new GridLayout(R,C,H,V)`" instead.

When you use a *GridLayout*, it's probably good form to add just enough components to fill the grid. However, this is not required. In fact, as long as you specify a non-zero value for the number of rows, then the number of columns is essentially ignored. The system will use just as many columns as are necessary to hold all the components that you add to the container. If you want to depend on this behavior, you should probably specify zero as the number of columns. You can also specify the number of rows as zero. In that case, you must give a non-zero number of columns. The system will use the specified number of columns, with just as many rows as necessary to hold the components that are added to the container.

Horizontal grids, with a single row, and vertical grids, with a single column, are very common. For example, suppose that `button1`, `button2`, and `button3` are buttons and that you'd like to display them in a horizontal row in a panel. If you use a horizontal grid for the panel, then the buttons will completely fill that panel and will all be the same size. The panel can be created as follows:

```
JPanel buttonBar = new JPanel();
buttonBar.setLayout( new GridLayout(1,3) );
    // (Note:  The "3" here is pretty much ignored, and
    //  you could also say "new GridLayout(1,0)".
```

```
        //  To leave gaps between the buttons, you could use
        //  "new GridLayout(1,0,5,5)".)
    buttonBar.add(button1);
    buttonBar.add(button2);
    buttonBar.add(button3);
```

You might find this button bar to be more attractive than the one that uses the default *FlowLayout* layout manager.

6.6.2 Borders

We have seen how to leave gaps between the components in a container, but what if you would like to leave a border around the outside of the container? This problem is not handled by layout managers. Instead, borders in Swing are represented by objects. A *Border* object can be added to any *JComponent*, not just to containers. Borders can be more than just empty space. The class `javax.swing.BorderFactory` contains a large number of static methods for creating border objects. For example, the function

```
BorderFactory.createLineBorder(Color.BLACK)
```

returns an object that represents a one-pixel wide black line around the outside of a component. If `comp` is a `JComponent`, a border can be added to `comp` using its `setBorder()` method. For example:

```
comp.setBorder( BorderFactory.createLineBorder(Color.BLACK) );
```

Once a border has been set for a `JComponent`, the border is drawn automatically, without any further effort on the part of the programmer. The border is drawn along the edges of the component, just inside its boundary. The layout manager of a `JPanel` or other container will take the space occupied by the border into account. The components that are added to the container will be displayed in the area inside the border. I don't recommend using a border on a `JPanel` that is being used as a drawing surface. However, if you do this, you should take the border into account. If you draw in the area occupied by the border, that part of your drawing will be covered by the border.

Here are some of the static methods that can be used to create borders:

- `BorderFactory.createEmptyBorder(top,left,bottom,right)` -- leaves an empty border around the edges of a component. Nothing is drawn in this space, so the background color of the component will appear in the area occupied by the border. The parameters are integers that give the width of the border along the top, left, bottom, and right edges of the component. This is actually very useful when used on a *JPanel* that contains other components. It puts some space between the components and the edge of the panel. It can also be useful on a *JLabel*, which otherwise would not have any space between the text and the edge of the label.

- `BorderFactory.createLineBorder(color,thickness)` -- draws a line around all four edges of a component. The first parameter is of type *Color* and specifies the color of the line. The second parameter is an integer that specifies the thickness of the border, in pixels. If the second parameter is omitted, a line of thickness 1 is drawn.
- `BorderFactory.createMatteBorder(top,left,bottom,right,color)` -- is similar to `createLineBorder`, except that you can specify individual thicknesses for the top, left, bottom, and right edges of the component.
- `BorderFactory.createEtchedBorder()` -- creates a border that looks like a groove etched around the boundary of the component. The effect is achieved using lighter and darker shades of the component's background color, and it does not work well with every background color.
- `BorderFactory.createLoweredBevelBorder()` -- gives a component a three-dimensional effect that makes it look like it is lowered into the computer screen. As with an EtchedBorder, this only works well for certain background colors.
- `BorderFactory.createRaisedBevelBorder()` -- similar to a LoweredBevelBorder, but the component looks like it is raised above the computer screen.
- `BorderFactory.createTitledBorder(title)` -- creates a border with a title. The title is a *String*, which is displayed in the upper left corner of the border.

There are many other methods in the `BorderFactory` class, most of them providing variations of the basic border styles given here. The following illustration shows six components with six different border styles. The text in each component is the command that created the border for that component:

(The source code for the program that produced this picture can be found in BorderDemo.java.)

6.6.3 SliderAndButtonDemo

Now that we have looked at components and layouts, it's time to put them together into some complete programs. We start with a simple demo that uses a *JLabel*, three *JButtons*, and a couple of *JSliders*, all laid out in a *GridLayout*:

The sliders in this program control the foreground and background color of the label, and the buttons control its font style. Writing this program is a matter of creating the components, laying them out, and programming listeners to respond to events from the sliders and buttons. My program is defined as a subclass of *JPanel* that implements *ChangeListener* and *ActionListener*, so that the panel itself can act as the listener for change events from the sliders and action events from the buttons. In the constructor, the six components are created and configured, a *GridLayout* is installed as the layout manager for the panel, and the components are added to the panel:

```
/* Create the display label, with properties to match the
       values of the sliders and the setting of the combo box. */

displayLabel = new JLabel("Hello World!", JLabel.CENTER);
displayLabel.setOpaque(true);
displayLabel.setBackground( new Color(100,100,100) );
displayLabel.setForeground( Color.RED );
displayLabel.setFont( new Font("Serif", Font.BOLD, 30) );
displayLabel.setBorder(BorderFactory.createEmptyBorder(0,8,0,8));

/* Create the sliders, and set up the panel to listen for
   ChangeEvents that are generated by the sliders. */

bgColorSlider = new JSlider(0,255,100);
bgColorSlider.addChangeListener(this);

fgColorSlider = new JSlider(0,100,0);
fgColorSlider.addChangeListener(this);

/* Create four buttons to control the font style, and set up the
   panel to listen for ActionEvents from the buttons. */

JButton plainButton = new JButton("Plain Font");
plainButton.addActionListener(this);
JButton italicButton = new JButton("Italic Font");
italicButton.addActionListener(this);
JButton boldButton = new JButton("Bold Font");
boldButton.addActionListener(this);


/* Set the layout for the panel, and add the four components.
```

```
        Use a GridLayout with 3 rows and 2 columns, and with
        5 pixels between components. */

setLayout(new GridLayout(3,2,5,5));
add(displayLabel);
add(plainButton);
add(bgColorSlider);
add(italicButton);
add(fgColorSlider);
add(boldButton);
```

The class also defines the methods required by the *ActionListener* and *ChangeListener* interfaces. The `actionPerformed()` method is called when the user clicks one of the buttons. This method changes the font in the *JLabel*, where the font depends on which button was clicked. To determine which button was clicked, the method uses `evt.getActionCommand()`, which returns the text from the button:

```
public void actionPerformed(ActionEvent evt) {
    String cmd = evt.getActionCommand();
    if (cmd.equals("Plain Font")) {
        displayLabel.setFont( new Font("Serif", Font.PLAIN, 30) );
    }
    else if (cmd.equals("Italic Font")) {
        displayLabel.setFont( new Font("Serif", Font.ITALIC, 30) );
    }
    else if (cmd.equals("Bold Font")) {
        displayLabel.setFont( new Font("Serif", Font.BOLD, 30) );
    }
}
```

And the `stateChanged()` method, which is called when the user manipulates one of the sliders, uses the value on the slider to compute a new foreground or background color for the label. The method checks `evt.getSource()` to determine which slider was changed:

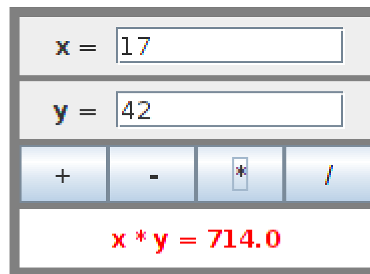
```
public void stateChanged(ChangeEvent evt) {
    if (evt.getSource() == bgColorSlider) {
        int bgVal = bgColorSlider.getValue();
        displayLabel.setBackground( new Color(bgVal,bgVal,bgVal) );
            // NOTE:  The background color is a shade of gray,
            //        determined by the setting on the slider.
    }
    else {
        float hue = fgColorSlider.getValue()/100.0f;
        displayLabel.setForeground( Color.getHSBColor(hue, 1.0f,
1.0f) );
            // Note:  The foreground color ranges through all the
colors
            // of the spectrum.
    }
}
```

Note that the slider variables are global variables in the program because they are referenced in the `stateChanged()` method as well as in the constructor. On the other hand, the button

variables are local variables in the constructor because that is the only place where they are used. The complete source code for this example is in the file SliderAndButtonDemo.java.

6.6.4 A Simple Calculator

As our next example, we look briefly at an example that uses nested subpanels to build a more complex user interface. The program has two *JTextFields* where the user can enter two numbers, four *JButtons* that the user can click to add, subtract, multiply, or divide the two numbers, and a *JLabel* that displays the result of the operation. Here is a picture from the program:



This example uses a panel with a *GridLayout* that has four rows and one column. In this case, the layout is created with the statement:

```
setLayout(new GridLayout(4,1,3,3));
```

which allows a 3-pixel gap between the rows where the gray background color of the panel is visible.

The first row of the grid layout actually contains two components, a *JLabel* displaying the text "x =" and a *JTextField*. A grid layout can only have one component in each position. In this case, the component in the first row is a *JPanel*, a subpanel that is nested inside the main panel. This subpanel in turn contains the label and text field. This can be programmed as follows:

```
xInput = new JTextField("0", 10); // Create a text field sized to
hold 10 chars.
JPanel xPanel = new JPanel();      // Create the subpanel.
xPanel.add( new JLabel(" x = ")); // Add a label to the subpanel.
xPanel.add(xInput);                // Add the text field to the
subpanel

add(xPanel);                       // Add the subpanel to the main
panel.
```

The subpanel uses the default *FlowLayout* layout manager, so the label and text field are simply placed next to each other in the subpanel at their preferred size, and are centered in the subpanel.

Similarly, the third row of the grid layout is a subpanel that contains four buttons. In this case, the subpanel uses a *GridLayout* with one row and four columns, so that the buttons are all the same size and completely fill the subpanel.

One other point of interest in this example is the `actionPerformed()` method that responds when the user clicks one of the buttons. This method must retrieve the user's numbers from the text field, perform the appropriate arithmetic operation on them (depending on which button was clicked), and set the text of the *JLabel* (named `answer`) to represent the result. However, the contents of the text fields can only be retrieved as strings, and these strings must be converted into numbers. If the conversion fails, the label is set to display an error message:

```
public void actionPerformed(ActionEvent evt) {

   double x, y;  // The numbers from the input boxes.

   try {
      String xStr = xInput.getText();
      x = Double.parseDouble(xStr);
   }
   catch (NumberFormatException e) {
          // The string xStr is not a legal number.
      answer.setText("Illegal data for x.");
      xInput.requestFocusInWindow();
      return;
   }

   try {
      String yStr = yInput.getText();
      y = Double.parseDouble(yStr);
   }
   catch (NumberFormatException e) {
         // The string yStr is not a legal number.
      answer.setText("Illegal data for y.");
      yInput.requestFocusInWindow();
      return;
   }

   /* Perform the operation based on the action command from the
    button.  The action command is the text displayed on the
button.
    Note that division by zero produces an error message. */

   String op = evt.getActionCommand();
   if (op.equals("+"))
      answer.setText( "x + y = " + (x+y) );
   else if (op.equals("-"))
      answer.setText( "x - y = " + (x-y) );
   else if (op.equals("*"))
      answer.setText( "x * y = " + (x*y) );
   else if (op.equals("/")) {
      if (y == 0)
         answer.setText("Can't divide by zero!");
      else
         answer.setText( "x / y = " + (x/y) );
```

```
    }

} // end actionPerformed()
```

The complete source code for this example can be found in *SimpleCalc.java*.

6.6.5 Using a null Layout

As mentioned above, it is possible to do without a layout manager altogether. For our next example, we'll look at a panel that does not use a layout manager. If you set the layout manager of a container to be `null`, by calling `container.setLayout(null)`, then you assume complete responsibility for positioning and sizing the components in that container.

If `comp` is any component, then the statement

```
comp.setBounds(x, y, width, height);
```

puts the top left corner of the component at the point `(x,y)`, measured in the coordinate system of the container that contains the component, and it sets the width and height of the component to the specified values. You should only set the bounds of a component if the container that contains it has a null layout manager. In a container that has a non-null layout manager, the layout manager is responsible for setting the bounds, and you should not interfere with its job.

Assuming that you have set the layout manager to `null`, you can call the `setBounds()` method any time you like. (You can even make a component that moves or changes size while the user is watching.) If you are writing a panel that has a known, fixed size, then you can set the bounds of each component in the panel's constructor. Note that you must also add the components to the panel, using the panel's `add(component)` instance method; otherwise, the component will not appear on the screen.

Our example contains four components: two buttons, a label, and a panel that displays a checkerboard pattern:

This is just an example of using a null layout; it doesn't do anything, except that clicking the buttons changes the text of the label. (We will use this example in Section 7.5 as a starting point for a checkers game.)

The panel in this program is defined by the class *NullLayoutDemo*, which is created as a subclass of *JPanel*. The four components are created and added to the panel in the constructor. Then the `setBounds()` method of each component is called to set the size and position of the component:

```
public NullLayoutDemo() {

    setLayout(null);  // I will do the layout myself!

    setBackground(new Color(0,120,0));  // A dark green background.

    setBorder( BorderFactory.createEtchedBorder() );

    setPreferredSize( new Dimension(350,240) );

    /* Create the components and add them to the content pane.  If 
you
         don't add them to a container, they won't appear, even if
         you set their bounds! */

    board = new Checkerboard();
        // (Checkerboard is a subclass of JPanel, defined below as 
a static
        //  nested class inside the main class.)
    add(board);

    newGameButton = new JButton("New Game");
    newGameButton.addActionListener(this);
    add(newGameButton);

    resignButton = new JButton("Resign");
    resignButton.addActionListener(this);
    add(resignButton);
```

```
    message = new JLabel("Click \"New Game\" to begin.");
    message.setForeground( new Color(100,255,100) ); // Light
green.
    message.setFont(new Font("Serif", Font.BOLD, 14));
    add(message);

    /* Set the position and size of each component by calling
         its setBounds() method. */

    board.setBounds(20,20,164,164);
    newGameButton.setBounds(210, 60, 120, 30);
    resignButton.setBounds(210, 120, 120, 30);
    message.setBounds(20, 200, 330, 30);

} // end constructor
```

It's fairly easy in this case to get a reasonable layout. It's much more difficult to do your own layout if you want to allow for changes of size. In that case, you have to respond to changes in the container's size by recomputing the sizes and positions of all the components that it contains. If you want to respond to changes in a container's size, you can register an appropriate listener with the container. Any component generates an event of type *ComponentEvent* when its size changes (and also when it is moved, hidden, or shown). You can register a *ComponentListener* with the container and respond to resize events by recomputing the sizes and positions of all the components in the container. Consult a Java reference for more information about *ComponentEvents*. However, my real advice is that if you want to allow for changes in the container's size, try to find a layout manager to do the work for you.

The complete source code for this example is in *NullLayoutDemo.java*.

6.6.6 A Little Card Game

For a final example, let's look at something a little more interesting as a program. The example is a simple card game in which you look at a playing card and try to predict whether the next card will be higher or lower in value. (Aces have the lowest value in this game.) You've seen a text-oriented version of the same game in Subsection 5.4.3. Section 5.4 also introduced *Deck*, *Hand*, and *Card* classes that are used by the program. In this GUI version of the game, you click on a button to make your prediction. If you predict wrong, you lose. If you make three correct predictions, you win. After completing one game, you can click the "New Game" button to start a new game. Here is what the program looks like in the middle of a game:

The complete source code for the panel can be found in the file *HighLowGUI.java*. I encourage you to compile and run it. Remember that you also need *Card.java*, *Deck.java*, and *Hand.java*, since they define classes that are used in the program.

The overall structure of the main panel in this example should be reasonably clear: It has three buttons in a subpanel at the bottom of the main panel and a large drawing surface that displays the cards and a message. (The cards and message are not components in this example; they are drawn using the graphics context in the panel's `paintComponent()` method.) The main panel uses a *BorderLayout*. The drawing surface occupies the `CENTER` position of the border layout. The subpanel that contains the buttons occupies the `SOUTH` position of the border layout, and the other three positions of the borderlayout are empty.

The drawing surface is defined by a nested class named *CardPanel*, which is subclass of *JPanel*. I have chosen to let the drawing surface object do most of the work of the game: It listens for events from the three buttons and responds by taking the appropriate actions. The main panel is defined by *HighLowGUI* itself, which is also a subclass of *JPanel*. The constructor of the *HighLowGUI* class creates all the other components, sets up event handling, and lays out the components:

```
public HighLowGUI() {   // The constructor.

   setBackground( new Color(130,50,40) );

   setLayout( new BorderLayout(3,3) );  // BorderLayout with 3-
pixel gaps.

   CardPanel board = new CardPanel();  // Where the cards are
drawn.
   add(board, BorderLayout.CENTER);

   JPanel buttonPanel = new JPanel();  // The subpanel that holds
the buttons.
   buttonPanel.setBackground( new Color(220,200,180) );
   add(buttonPanel, BorderLayout.SOUTH);

   JButton higher = new JButton( "Higher" );
   higher.addActionListener(board);   // The CardPanel listens for
events.
```

```
   buttonPanel.add(higher);

   JButton lower = new JButton( "Lower" );
   lower.addActionListener(board);
   buttonPanel.add(lower);

   JButton newGame = new JButton( "New Game" );
   newGame.addActionListener(board);
   buttonPanel.add(newGame);

   setBorder(BorderFactory.createLineBorder( new Color(130,50,40),
3) );

}  // end constructor
```

The programming of the drawing surface class, *CardPanel*, is a nice example of thinking in terms of a state machine. (See Subsection 6.4.4.) It is important to think in terms of the states that the game can be in, how the state can change, and how the response to events can depend on the state. The approach that produced the original, text-oriented game in Subsection 5.4.3 is not appropriate here. Trying to think about the game in terms of a process that goes step-by-step from beginning to end is more likely to confuse you than to help you.

The state of the game includes the cards and the message. The cards are stored in an object of type *Hand*. The message is a *String*. These values are stored in instance variables. There is also another, less obvious aspect of the state: Sometimes a game is in progress, and the user is supposed to make a prediction about the next card. Sometimes we are between games, and the user is supposed to click the "New Game" button. It's a good idea to keep track of this basic difference in state. The *CardPanel* class uses a boolean instance variable named `gameInProgress` for this purpose.

The state of the game can change whenever the user clicks on a button. The *CardPanel* class implements the *ActionListener* interface and defines an `actionPerformed()` method to respond to the user's clicks. This method simply calls one of three other methods, `doHigher()`, `doLower()`, or `newGame()`, depending on which button was pressed. It's in these three event-handling methods that the action of the game takes place.

We don't want to let the user start a new game if a game is currently in progress. That would be cheating. So, the response in the `newGame()` method is different depending on whether the state variable `gameInProgress` is true or false. If a game is in progress, the `message` instance variable should be set to be an error message. If a game is not in progress, then all the state variables should be set to appropriate values for the beginning of a new game. In any case, the board must be repainted so that the user can see that the state has changed. The complete `newGame()` method is as follows:

```
/**
 * Called by the CardPanel constructor, and called by
actionPerformed() if
 * the user clicks the "New Game" button.  Start a new game.
 */
void doNewGame() {
```

```
   if (gameInProgress) {
         // If the current game is not over, it is an error to try
         // to start a new game.
      message = "You still have to finish this game!";
      repaint();
      return;
   }
   deck = new Deck();   // Create the deck and hand to use for this
game.
   hand = new Hand();
   deck.shuffle();
   hand.addCard( deck.dealCard() );  // Deal the first card into
the hand.
   message = "Is the next card higher or lower?";
   gameInProgress = true;
   repaint();
} // end doNewGame()
```

The `doHigher()` and `doLower()` methods are almost identical to each other (and could probably have been combined into one method with a parameter, if I were more clever). Let's look at the `doHigher()` routine. This is called when the user clicks the "Higher" button. This only makes sense if a game is in progress, so the first thing `doHigher()` should do is check the value of the state variable `gameInProgress`. If the value is `false`, then `doHigher()` should just set up an error message. If a game is in progress, a new card should be added to the hand and the user's prediction should be tested. The user might win or lose at this time. If so, the value of the state variable `gameInProgress` must be set to `false` because the game is over. In any case, the board is repainted to show the new state. Here is the `doHigher()` method:

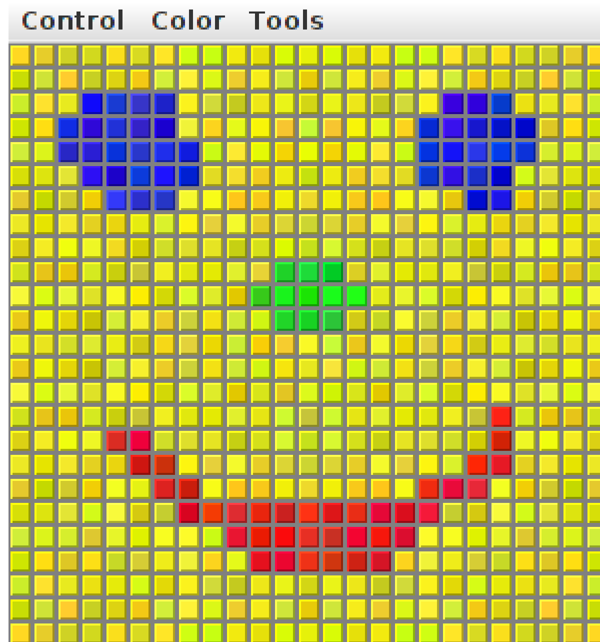
```
/**
 * Called by actionPerformed() when user clicks the "Higher"
button.
 * Check the user's prediction.  Game ends if user guessed
 * wrong or if the user has made three correct predictions.
 */
void doHigher() {
   if (gameInProgress == false) {
         // If the game has ended, it was an error to click
"Higher",
         // So set up an error message and abort processing.
      message = "Click \"New Game\" to start a new game!";
      repaint();
      return;
   }
   hand.addCard( deck.dealCard() );     // Deal a card to the hand.
   int cardCt = hand.getCardCount();
   Card thisCard = hand.getCard( cardCt - 1 );  // Card just dealt.
   Card prevCard = hand.getCard( cardCt - 2 );  // The previous
card.
   if ( thisCard.getValue() < prevCard.getValue() ) {
      gameInProgress = false;
      message = "Too bad! You lose.";
   }
   else if ( thisCard.getValue() == prevCard.getValue() ) {
      gameInProgress = false;
```

```
      message = "Too bad!  You lose on ties.";
   }
   else if ( cardCt == 4) {  // The hand is full, after three
correct guesses.
      gameInProgress = false;
      message = "You win!  You made three correct guesses.";
   }
   else {
      message = "Got it right!  Try for " + cardCt + ".";
   }
   repaint();
} // end doHigher()
```

The `paintComponent()` method of the *CardPanel* class uses the values in the state variables to decide what to show. It displays the string stored in the `message` variable. It draws each of the cards in the `hand`. There is one little tricky bit: If a game is in progress, it draws an extra face-down card, which is not in the hand, to represent the next card in the deck. Drawing the cards requires some care and computation. I wrote a method, "`void drawCard(Graphics g, Card card, int x, int y)`", which draws a card with its upper left corner at the point `(x,y)`. The `paintComponent()` routine decides where to draw each card and calls this routine to do the drawing. You can check out all the details in the source code, *HighLowGUI.java*. (The playing cards used in this program are not very impressive

Menus and Dialogs

WE HAVE ALREADY ENCOUNTERED many of the basic aspects of GUI programming, but professional programs use many additional features. We will cover some of the advanced features of Java GUI programming in Chapter 13, but in this section we look briefly at a few more features that are essential for writing GUI programs. I will discuss these features in the context of a "MosaicDraw" program that is shown in this picture:



The source code for the program is in the file *MosaicDraw.java*. The program also requires *MosaicPanel.java* and *MosaicDrawController.java*. You will want to try it out!

As the user clicks-and-drags the mouse in the large drawing area of this program, it leaves a trail of little colored squares. There is some random variation in the color of the squares. (This is meant to make the picture look a little more like a real mosaic, which is a picture made out of small colored stones in which there would be some natural color variation.) There is a menu bar above the drawing area. The "Control" menu contains commands for filling and clearing the drawing area, along with a few options that affect the appearance of the picture. The "Color" menu lets the user select the color that will be used when the user draws. The "Tools" menu affects the behavior of the mouse. Using the default "Draw" tool, the mouse leaves a trail of single squares. Using the "Draw 3x3" tool, the mouse leaves a swath of colored squares that is three squares wide. There are also "Erase" tools, which let the user set squares back to their default black color.

The drawing area of the program is a panel that belongs to the *MosaicPanel* class, a subclass of *JPanel* that is defined in *MosaicPanel.java*. *MosaicPanel* is a highly reusable class for representing mosaics of colored rectangles. It was also used behind the scenes in the sample program in Subsection 4.6.3. The *MosaicPanel* class does not directly support drawing on the mosaic, but it does support setting the color of each individual square. The MosaicDraw program installs a mouse listener on the panel; the mouse listener responds to mousePressed and mouseDragged events on the panel by setting the color of the square that contains the mouse. This is a nice example of applying a listener to an object to do something that was not programmed into the object itself.

The file *MosaicDraw.java* is a simple class that contains only the `main()` routine for the program. Most of the programming for MosaicDraw can be found in *MosaicDrawController.java*. (It might have gone into the *MosaicPanel* class, if I had not

decided to use that pre-existing class in unmodified form.) It is the *MosaicDrawController* class that creates a *MosaicPanel* object and adds a mouse listener to it. It also creates the menu bar that is shown at the top of the program, and it implements all the commands in the menu bar. It has an instance method `getMosaicPanel()` that returns a reference to the mosaic panel that it has created, and it has another instance method `getMenuBar()` that returns a menu bar for the program. These methods are used to obtain the panel and menu bar so that they can be added to the program's window.

I urge you to study MosaicDrawController.java and MosaicDraw.java. I will not be discussing all aspects of the code here, but you should be able to understand it all after reading this section. As for MosaicPanel.java, it uses some techniques that you would not understand at this point, but I encourage you to at least read the comments in this file to learn about the API for mosaic panels.

6.7.1 Menus and Menubars

MosaicDraw is the first example that we have seen that uses a menu bar. Fortunately, menus are very easy to use in Java. The items in a menu are represented by the class *JMenuItem* (this class and other menu-related classes are in package `javax.swing`). Menu items are used in almost exactly the same way as buttons. In fact, *JMenuItem* and *JButton* are both subclasses of a class, *AbstractButton*, that defines their common behavior. In particular, a *JMenuItem* is created using a constructor that specifies the text of the menu item, such as:

```
JMenuItem fillCommand = new JMenuItem("Fill");
```

You can add an *ActionListener* to a *JMenuItem* by calling the menu item's `addActionListener()` method. The `actionPerformed()` method of the action listener is called when the user selects the item from the menu. You can change the text of the item by calling its `setText(String)` method, and you can enable it and disable it using the `setEnabled(boolean)` method. All this works in exactly the same way as for a *JButton*.

The main difference between a menu item and a button, of course, is that a menu item is meant to appear in a menu rather than in a panel. A menu in Java is represented by the class *JMenu*. A *JMenu* has a name, which is specified in the constructor, and it has an `add(JMenuItem)` method that can be used to add a *JMenuItem* to the menu. For example, the "Tools" menu in the MosaicDraw program could be created as follows, where `listener` is a variable of type `ActionListener`:

```
JMenu toolsMenu = new JMenu("Tools");  // Create a menu with name
"Tools"

JMenuItem drawCommand = new JMenuItem("Draw");   // Create a menu
item.
drawCommand.addActionListener(listener);         // Add listener to
menu item.
```

```
toolsMenu.add(drawCommand);                        // Add menu item
to menu.

JMenuItem eraseCommand = new JMenuItem("Erase"); // Create a menu
item.
eraseCommand.addActionListener(listener);          // Add listener to
menu item.
toolsMenu.add(eraseCommand);                       // Add menu item
to menu.
   .
   .  // Create and add other menu items.
   .
```

Once a menu has been created, it must be added to a menu bar. A menu bar is represented by the class *JMenuBar*. A menu bar is just a container for menus. It does not have a name, and its constructor does not have any parameters. It has an `add(JMenu)` method that can be used to add menus to the menu bar. The name of the menu then appears in the menu bar. For example, the MosaicDraw program uses three menus, `controlMenu`, `colorMenu`, and `toolsMenu`. We could create a menu bar and add the menus to it with the statements:

```
JMenuBar menuBar = new JMenuBar();
menuBar.add(controlMenu);
menuBar.add(colorMenu);
menuBar.add(toolsMenu);
```

The final step in using menus is to use the menu bar in a window such as a *JFrame*. We have already seen that a frame has a "content pane." The menu bar is another component of the frame, not contained inside the content pane. The *JFrame* class has an instance method `setMenuBar(JMenuBar)` that can be used to set the menu bar. (There can only be one, so this is a "set" method rather than an "add" method.) In the MosaicDraw program, the menu bar is created by a *MosaicDrawController* object and can be obtained by calling that object's `getMenuBar()` method. The `main()` routine in MosaicDraw.java gets the menu bar from the controller and adds it to the window. Here is the basic code that is used (in somewhat modified form) to set up the interface:

```
MosaicDrawController controller = new MosaicDrawController();

MosaicPanel content = controller.getMosaicPanel();
window.setContentPane( content );  // Use panel from controller as
content pane.

JMenuBar menuBar = controller.getMenuBar();
window.setJMenuBar( menuBar );    // Use the menu bar from the
controller.
```

Using menus always follows the same general pattern: Create a menu bar. Create menus and add them to the menu bar. Create menu items and add them to the menus (and set up listening to handle action events from the menu items). Use the menu bar in a window by calling the window's `setJMenuBar()` method.

There are other kinds of menu items, defined by subclasses of *JMenuItem*, that can be added to menus. One of these is *JCheckBoxMenuItem*, which represents menu items that can be in one of two states, selected or not selected. A *JCheckBoxMenuItem* has the same functionality and is used in the same way as a *JCheckBox* (see Subsection 6.5.3). Three *JCheckBoxMenuItems* are used in the "Control" menu of the MosaicDraw program. One is used to turn the random color variation of the squares on and off. Another turns a symmetry feature on and off; when symmetry is turned on, the user's drawing is reflected horizontally and vertically to produce a symmetric pattern. And the third checkbox menu item shows and hides the "grouting" in the mosaic; the grouting is the gray lines that are drawn around each of the little squares in the mosaic. The menu item that corresponds to the "Use Randomness" option in the "Control" menu could be set up with the statements:

```
JMenuItem useRandomnessToggle = new JCheckBoxMenuItem("Use
Randomness");
useRandomnessToggle.addActionListener(listener);  // Set up a
listener.
useRandomnessToggle.setSelected(true);  // Randomness is initially
turned on.
controlMenu.add(useRandomnessToggle);   // Add the menu item to the
menu.
```

In my program, the "Use Randomness" *JCheckBoxMenuItem* corresponds to a boolean-valued instance variable named `useRandomness` in the *MosaicDrawController* class. This variable is part of the state of the controller object. Its value is tested whenever the user draws one of the squares, to decide whether or not to add a random variation to the color of the square. When the user selects the "Use Randomness" command from the menu, the state of the *JCheckBoxMenuItem* is reversed, from selected to not-selected or from not-selected to selected. The *ActionListener* for the menu item checks whether the menu item is selected or not, and it changes the value of `useRandomness` to match. Note that selecting the menu command does not have any immediate effect on the picture that is shown in the window. It just changes the state of the program so that future drawing operations on the part of the user will have a different effect. The "Use Symmetry" option in the "Control" menu works in much the same way. The "Show Grouting" option is a little different. Selecting the "Show Grouting" option does have an immediate effect: The picture is redrawn with or without the grouting, depending on the state of the menu item.

My program uses a single *ActionListener* to respond to all of the menu items in all the menus. This is not a particularly good design, but it is easy to implement for a small program like this one. The `actionPerformed()` method of the listener object uses the statement

```
String command = evt.getActionCommand();
```

to get the action command of the source of the event; this will be the text of the menu item. The listener tests the value of `command` to determine which menu item was selected by the user. If the menu item is a *JCheckBoxMenuItem*, the listener must check the state of the menu item. The menu item is the source of the event that is being processed. The listener can get its hands on the menu item object by calling `evt.getSource()`. Since the return value of `getSource()` is

of type *Object*, the return value must be type-cast to the correct type. Here, for example, is the code that handles the "Use Randomness" command:

```
if (command.equals("Use Randomness")) {
        // Set the value of useRandomness depending on the menu
item's state.
   JCheckBoxMenuItem toggle = (JCheckBoxMenuItem)evt.getSource();
   useRandomness = toggle.isSelected();
}
```

(The `actionPerformed()` method uses a rather long `if..then..else` statement to check all the possible action commands. It might be more natural and efficient use a `switch` statement with `command` as the selector and all the possible action commands as cases.)

In addition to menu items, a menu can contain lines that separate the menu items into groups. In the MosaicDraw program, the "Control" menu contains such a separator. A *JMenu* has an instance method `addSeparator()` that can be used to add a separator to the menu. For example, the separator in the "Control" menu was created with the statement:

```
controlMenu.addSeparator();
```

A menu can also contain a submenu. The name of the submenu appears as an item in the main menu. When the user moves the mouse over the submenu name, the submenu pops up. (There is no example of this in the MosaicDraw program.) It is very easy to do this in Java: You can add one *JMenu* to another *JMenu* using a statement such as `mainMenu.add(submenu)`, and it becomes a submenu.

6.7.2 Dialogs

One of the commands in the "Color" menu of the MosaicDraw program is "Custom Color...". When the user selects this command, a new window appears where the user can select a color. This window is an example of a dialog or dialog box. A dialog is a type of window that is generally used for short, single purpose interactions with the user. For example, a dialog box can be used to display a message to the user, to ask the user a question, to let the user select a file to be opened, or to let the user select a color. In Swing, a dialog box is represented by an object belonging to the class *JDialog* or to a subclass.

The *JDialog* class is very similar to *JFrame* and is used in much the same way. Like a frame, a dialog box is a separate window. Unlike a frame, however, a dialog is not completely independent. Every dialog is associated with a frame (or another dialog), which is called its parent window. The dialog box is dependent on its parent. For example, if the parent is closed, the dialog box will also be closed. It is possible to create a dialog box without specifying a parent, but in that case an invisible frame is created by the system to serve as the parent.

Dialog boxes can be either modal or modeless. When a modal dialog is created, its parent frame is blocked. That is, the user will not be able to interact with the parent until the dialog box is closed. Modeless dialog boxes do not block their parents in the same way, so they seem a lot more like independent windows. In practice, modal dialog boxes are easier to use and are much more common than modeless dialogs. All the examples we will look at are modal.

Aside from having a parent, a *JDialog* can be created and used in the same way as a *JFrame*. However, I will not give any examples here of using *JDialog* directly. Swing has many convenient methods for creating common types of dialog boxes. For example, the color choice dialog that appears when the user selects the "Custom Color" command in the MosaicDraw program belongs to the class *JColorChooser*, which is a subclass of *JDialog*. The *JColorChooser* class has a static method that makes color choice dialogs very easy to use:

```
Color JColorChooser.showDialog(Component parentComp,
                                        String title, Color
initialColor)
```

When you call this method, a dialog box appears that allows the user to select a color. The first parameter specifies the parent of the dialog; the parent window of the dialog will be the window (if any) that contains `parentComp`; this parameter can be `null` and it can itself be a frame or dialog object. The second parameter is a string that appears in the title bar of the dialog box. And the third parameter, `initialColor`, specifies the color that is selected when the color choice dialog first appears. The dialog has a sophisticated interface that allows the user to select a color. When the user presses an "OK" button, the dialog box closes and the selected color is returned as the value of the method. The user can also click a "Cancel" button or close the dialog box in some other way; in that case, `null` is returned as the value of the method. This is a modal dialog, and `showDialog()` does not return until the user dismisses the dialog box in some way. By using this predefined color chooser dialog, you can write one line of code that will let the user select an arbitrary color. Swing also has a *JFileChooser* class that makes it almost as easy to show a dialog box that lets the user select a file to be opened or saved.

The *JOptionPane* class includes a variety of methods for making simple dialog boxes that are variations on three basic types: a "message" dialog, a "confirm" dialog, and an "input" dialog. (The variations allow you to provide a title for the dialog box, to specify the icon that appears in the dialog, and to add other components to the dialog box. I will only cover the most basic forms here.)

A message dialog simply displays a message string to the user. The user (hopefully) reads the message and dismisses the dialog by clicking the "OK" button. A message dialog can be shown by calling the static method:

```
void JOptionPane.showMessageDialog(Component parentComp, String
message)
```

The message can be more than one line long. Lines in the message should be separated by newline characters, `\n`. New lines will not be inserted automatically, even if the message is very long. For example, assuming that the special variable `this` refers to a *Component*:

```
JOptionPane.showMessageDialog( this, "This program is about to
crash!\n"
                                       + "Sorry about that.");
```

An input dialog displays a question or request and lets the user type in a string as a response. You can show an input dialog by calling:

```
String JOptionPane.showInputDialog(Component parentComp, String
question)
```

Again, `parentComp` can be `null`, and the question can include newline characters. The dialog box will contain an input box, an "OK" button, and a "Cancel" button. If the user clicks "Cancel", or closes the dialog box in some other way, then the return value of the method is `null`. If the user clicks "OK", then the return value is the string that was entered by the user. Note that the return value can be an empty string (which is not the same as a `null` value), if the user clicks "OK" without typing anything in the input box. If you want to use an input dialog to get a numerical value from the user, you will have to convert the return value into a number (see Subsection 3.7.2). As an example,

```
String name;
name = JOptionPanel.showInputDialog(null, "Hi!  What's your
name?");
if (name == null)
   JOptionPane.showMessageDialog(null, "Well, I'll call you
Grumpy.");
else
   JOptionPane.showMessageDialog(null, "Pleased to meet you, " +
name);
```

Finally, a confirm dialog presents a question and three response buttons: "Yes", "No", and "Cancel". A confirm dialog can be shown by calling:

```
int JOptionPane.showConfirmDialog(Component parentComp, String
question)
```

The return value tells you the user's response. It is one of the following constants:

- `JOptionPane.YES_OPTION` -- the user clicked the "Yes" button
- `JOptionPane.NO_OPTION` -- the user clicked the "No" button
- `JOptionPane.CANCEL_OPTION` -- the user clicked the "Cancel" button
- `JOptionPane.CLOSE_OPTION` -- the dialog was closed in some other way.

By the way, it is possible to omit the Cancel button from a confirm dialog by calling one of the other methods in the `JOptionPane` class. Just call:

```
JOptionPane.showConfirmDialog(
                parent, question, title, JOptionPane.YES_NO_OPTION
)
```

The final parameter is a constant which specifies that only a "Yes" button and a "No" button should be used. The third parameter is a string that will be displayed as the title of the dialog box window.

A small demo program, *SimpleDialogDemo.java* is available to demonstrate JColorChooser and several JOptionPane dialogs.

6.7.3 Fine Points of Frames

In previous sections, whenever I used a frame, I created a *JFrame* object in a `main()` routine and installed a panel as the content pane of that frame. This works fine, but a more object-oriented approach is to define a subclass of *JFrame* and to set up the contents of the frame in the constructor of that class. This is what I did in the case of the MosaicDraw program. *MosaicDraw* is defined as a subclass of *JFrame*. The definition of this class is very short, but it illustrates several new features of frames that I want to discuss:

```
public class MosaicDraw extends JFrame {

   public static void main(String[] args) {
      JFrame window = new MosaicDraw();
      window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      window.setVisible(true);
   }

   public MosaicDraw() {
      super("Mosaic Draw");
      MosaicDrawController controller = new MosaicDrawController();
      setContentPane( controller.getMosaicPanel() );
      setJMenuBar( controller.getMenuBar() );
      pack();
      Dimension screensize = 
Toolkit.getDefaultToolkit().getScreenSize();
      setLocation( (screensize.width - getWidth())/2,
                                  (screensize.height - 
getHeight())/2 );
   }

}
```

The constructor in this class begins with the statement `super("Mosaic Draw")`, which calls the constructor in the superclass, *JFrame*. The parameter specifies a title that will appear in the title bar of the window. The next three lines of the constructor set up the contents of the window; a *MosaicDrawController* is created, and the content pane and menu bar of the window are obtained from the controller. The next line is something new. If `window` is a variable of type

JFrame (or *JDialog*), then the statement `window.pack()` will resize the window so that its size matches the preferred size of its contents. (In this case, of course, "`pack()`" is equivalent to "`this.pack()`"; that is, it refers to the window that is being created by the constructor.) The `pack()` method is usually the best way to set the size of a window. Note that it will only work correctly if every component in the window has a correct preferred size. This is only a problem in two cases: when a panel is used as a drawing surface and when a panel is used as a container with a `null` layout manager. In both these cases there is no way for the system to determine the correct preferred size automatically, and you should set a preferred size by hand. For example:

```
panel.setPreferredSize( new Dimension(400, 250) );
```

The last two lines in the constructor position the window so that it is exactly centered on the screen. The line

```
Dimension screensize = Toolkit.getDefaultToolkit().getScreenSize();
```

determines the size of the screen. The size of the screen is `screensize.width` pixels in the horizontal direction and `screensize.height` pixels in the vertical direction. The `setLocation()` method of the frame sets the position of the upper left corner of the frame on the screen. The expression "`screensize.width - getWidth()`" is the amount of horizontal space left on the screen after subtracting the width of the window. This is divided by 2 so that half of the empty space will be to the left of the window, leaving the other half of the space to the right of the window. Similarly, half of the extra vertical space is above the window, and half is below.

Note that the constructor has created the window and set its size and position, but that at the end of the constructor, the window is not yet visible on the screen. (More exactly, the constructor has created the window *object*, but the visual representation of that object on the screen has not yet been created.) To show the window on the screen, it will be necessary to call its instance method, `window.setVisible(true)`.

In addition to the constructor, the *MosaicDraw* class includes a `main()` routine. This makes it possible to run *MosaicDraw* as a stand-alone application. (The `main()` routine, as a `static` method, has nothing to do with the function of a *MosaicDraw* object, and it could (and perhaps should) be in a separate class.) The `main()` routine creates a *MosaicDraw* and makes it visible on the screen. It also calls

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

which means that the program will end when the user closes the window. Note that this is not done in the constructor because doing it there would make *MosaicDraw* less flexible. It is possible, for example, to write a program that lets the user open multiple MosaicDraw windows. In that case, we don't want to shut down the whole program just because the user has closed *one* of the windows. There are other possible values for the default close operation of a window:

- `JFrame.DO_NOTHING_ON_CLOSE` -- the user's attempts to close the window by clicking its close box will be ignored, except that it will generate a *WindowEvent*. A program can listen for this event and take any action it wants when the user attempts to close the window.
- `JFrame.HIDE_ON_CLOSE` -- when the user clicks its close box, the window will be hidden just as if `window.setVisible(false)` were called. The window can be made visible again by calling `window.setVisible(true)`. This is the value that is used if you do not specify another value by calling `setDefaultCloseOperation`.
- `JFrame.DISPOSE_ON_CLOSE` -- the window is closed and any operating system resources used by the window are released. It is not possible to make the window visible again. (This is the proper way to permanently get rid of a window without ending the program. You can accomplish the same thing programmatically by calling the instance method `window.dispose()`.)

6.7.4 Creating Jar Files

As the final topic for this chapter, we look again at jar files. Recall that a jar file is a "java archive" that can contain a number of class files. When creating a program that uses more than one class, it's usually a good idea to place all the classes that are required by the program into a jar file. If that is done, then a user will only need that one file to run the program. In fact, it is possible to make a so-called executable jar file. A user can run an executable jar file in much the same way as any other application, usually by double-clicking the icon of the jar file. (The user's computer must have a correct version of Java installed, and the computer must be configured correctly for this to work. The configuration is usually done automatically when Java is installed, at least on Windows and Mac OS.)

The question, then, is how to create a jar file. The answer depends on what programming environment you are using. The two basic types of programming environment -- command line and IDE -- were discussed in Section 2.6. Any IDE (Integrated Programming Environment) for Java should have a command for creating jar files. In the Eclipse IDE, for example, it can be done as follows: In the Package Explorer pane, select the programming project (or just all the individual source code files that you need). Right-click on the selection, and choose "Export" from the menu that pops up. In the window that appears, select "JAR file" and click "Next". In the window that appears next, enter a name for the jar file in the box labeled "JAR file". (Click the "Browse" button next to this box to select the file name using a file dialog box.) The name of the file should end with ".jar". If you are creating a regular jar file, not an executable one, you can hit "Finish" at this point, and the jar file will be created. To create an executable file, hit the "Next" button *twice* to get to the "Jar Manifest Specification" screen. At the bottom of this screen is an input box labeled "Main class". You have to enter the name of the class that contains the `main()` routine that will be run when the jar file is executed. If you hit the "Browse" button next to the "Main class" box, you can select the class from a list of classes that contain `main()` routines. Once you've selected the main class, you can click the "Finish" button to create the executable jar file. (Note that newer versions of Eclipse also have an option for exporting an executable Jar file in fewer steps.)

It is also possible to create jar files on the command line. The Java Development Kit includes a command-line program named `jar` that can be used to create jar files. If all your classes are in the default package (like most of the examples in this book), then the `jar` command is easy to use. To create a non-executable jar file on the command line, change to the directory that contains the class files that you want to include in the jar. Then give the command

```
jar  cf  JarFileName.jar  *.class
```

where `JarFileName` can be any name that you want to use for the jar file. The "`*`" in "`*.class`" is a wildcard that makes `*.class` match every class file in the current directory. This means that all the class files in the directory will be included in the jar file. If you want to include only certain class files, you can name them individually, separated by spaces. (Things get more complicated if your classes are not in the default package. In that case, the class files must be in subdirectories of the directory in which you issue the `jar` command. See Subsection 2.6.6.)

Making an executable jar file on the command line is more complicated. There has to be some way of specifying which class contains the `main()` routine. This is done by creating a manifest file. The manifest file can be a plain text file containing a single line of the form

```
Main-Class: ClassName
```

where `ClassName` should be replaced by the name of the class that contains the `main()` routine. For example, if the `main()` routine is in the class *MosaicDraw*, then the manifest file should read "`Main-Class: MosaicDraw`". You can give the manifest file any name you like. Put it in the same directory where you will issue the `jar` command, and use a command of the form

```
jar  cmf  ManifestFileName  JarFileName.jar  *.class
```

to create the jar file. (The `jar` command is capable of performing a variety of different operations. The first parameter to the command, such as "`cf`" or "`cmf`", tells it which operation to perform.)

By the way, if you have successfully created an executable jar file, you can run it on the command line using the command "`java -jar`". For example:

```
java  -jar  JarFileName.jar
```