Chapter 6

Graphical User Interfaces in JAVA

Contents

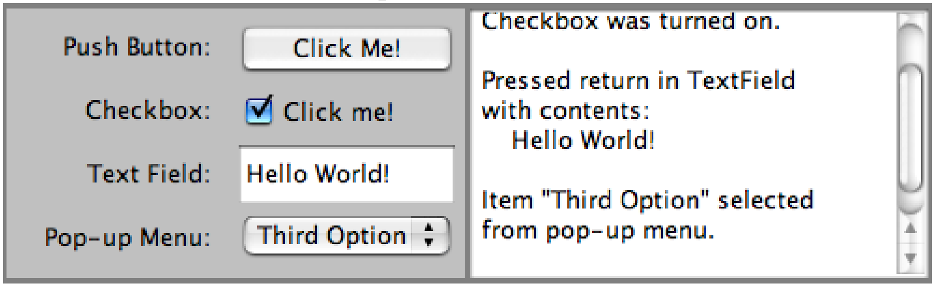
6.1 Introduction: The Modern User Interface

When computers were first introduced, ordinary people,including most programmers, couldn't get near them. They were locked up in rooms with white-coated attendants who would take your programs and data, feed them to the computer, and return the computer's response some time later. When timesharing – where the computer switches its attention rapidly from one person to another – was invented in the 1960s, it became possible for several people to interact directly with the computer at the same time. On a timesharing system, users sit at "*terminals*" where they type commands to the computer, and the computer types back its response. Early personal computers also used typed commands and responses, except that there was only one person involved at a time. This type of interaction between a user and a computer is called a *command-line interface*.

Today most people interact with computers in a completely different way. They use a *Graphical User Interface*, or GUI. The computer draws interface components on the screen. The components include things like windows, scroll bars, menus, buttons, and icons. Usually, a mouse is used to manipulate such components.

A lot of GUI interface components have become fairly standard. That is, they have similar appearance and behavior on many different computer platforms including Macintosh, Windows, and Linux. Java programs, which are supposed to run on many different platforms without modification to the program, can use all the standard GUI components. They might vary a little in appearance from platform to platform, but their functionality should be identical on any computer on which the program runs.

Below is a very simple Java program–actually an "applet,"–that shows a few standard GUI interface components. There are four components that the user can interact with: a button, a checkbox, a text field, and a pop-up menu. These components are labeled. There are a few other components in the applet. The labels themselves are components (even though you can't interact with them). The right half of the applet is a text area component, which can display multiple lines of text, and a scrollbar component appears alongside the text area when the number of lines of text becomes larger than will fit in the text area. And in fact, in Java terminology, the whole applet is itself considered to be a "component."



Checkbox: ☑ Click me!

Text Field: Hello World!

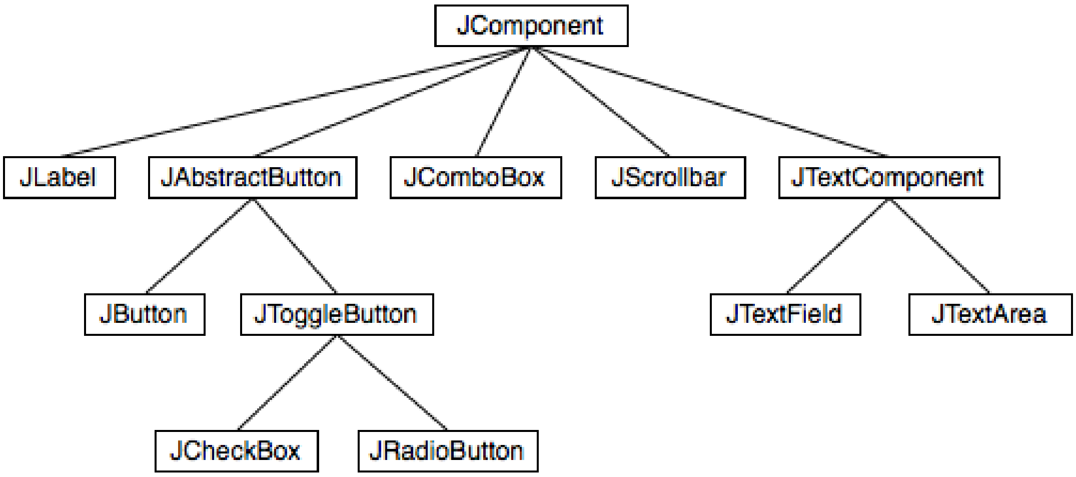
Pop-up Menu: Third Option

```
Checkbox was turned on.

Pressed return in TextField
with contents:
   Hello World!

Item "Third Option" selected
from pop-up menu.
```

Java actually has two complete sets of GUI components. One of these, the *AWT* or Abstract Windowing Toolkit, was available in the original version of Java. The other, which is known as Swing, is included in Java version 1.2 or later, and is used in preference to the AWT in most modern JAVA programs. The applet that is shown above uses components that are part of Swing.

When a user interacts with the GUI components in this applet, an "event" is generated. For example, clicking a push button generates an event, and pressing return while typing in a text field generates an event. Each time an event is generated, a message is sent to the applet telling it that the event has occurred, and the applet responds according to its program. In fact, the program consists mainly of "event handlers" that tell the applet how to respond to various types of events. In this example, the applet has been programmed to respond to each event by displaying a message in the text area.

The use of the term "message" here is deliberate. Messages are sent to objects. In fact, JAVA GUI components are implemented as objects. JAVA includes many predefined classes that represent various types of GUI components. Some of these classes are subclasses of others. Here is a diagram showing some of Swing's GUI classes and their relationships:



Note that all GUI classes are subclasses, directly or indirectly, of a class called `JComponent`, which represents general properties that are shared by all Swing components. Two of the direct subclasses of `JComponent` themselves have subclasses. The classes `JTextArea` and `JTextField`, which have certain behaviors in common, are grouped together as subclasses of `JTextComponent`. Also, `JButton` and `JToggleButton` are subclasses of `JAbstractButton`, which represents properties common to both buttons and checkboxes.

Just from this brief discussion, perhaps you can see how GUI programming can make effective use of object-oriented design. In fact, GUI's, with their "visible objects," are probably a major factor contributing to the popularity of OOP.

6.2 The Basic GUI Application

THERE ARE TWO BASIC TYPES of GUI program in JAVA: *stand-alone applications* and *applets*. An applet is a program that runs in a rectangular area on a Web page. Applets are generally small programs, meant to do fairly simple things, although there is nothing to stop them from being very complex. Applets were responsible for a lot of the initial excitement about JAVA when it was introduced, since they could do things that could not otherwise be done on Web pages. However, there are now easier ways to do many of the more basic things that can be done with applets, and they are no longer the main focus of interest in JAVA. Nevertheless, there are still some things that can be done best with applets, and they are still fairly common on the Web.

A stand-alone application is a program that runs on its own, without depending on a Web browser. You've been writing stand-alone applications all along. Any class that has a main() method defines a stand-alone application; running the program just means executing this main() method. However, the programs that you've seen up till now have been "command-line" programs, where the user and computer interact by typing things back and forth to each other. 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. The main method of a GUI program creates one or more such components and displays them on the computer screen. Very often, that's all it does. Once a GUI component has been created, it follows its **own** programming—programming that tells it how to draw itself 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. By the way, this program can be placed in a file named HelloWorldGUI1.java, compiled, and run 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: the source code "imports" the class javax.swing.JOptionPane to make it possible to refer to the JOptionPane class using its simple name.)

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. I will explain how it works below, but it will take the rest of the chapter before you will really understand completely.

```
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);

   }

}
```

6.2.1 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 large blank area. 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 in the constructor, "GUI Test", 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() method does end. However, the the window is still on the screen and the program as a whole does not end until the user clicks the OK button.

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. displayPanel is a variable of type HelloWorldDisplay, which is a nested static class inside the HelloWorldGUI2 class. 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. 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 nested 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.

6.2.2 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. Before these components can appear on the screen, they must be added to a container. In this program, the variable named content refers to a JPanel that is used as a container, and 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 the component 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 or applet. 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...

6.2.3 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.

Event listeners are the most common technique for handling events in JAVA. 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. 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 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 some other object 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. (We will return to events and listeners in much more detail in later sections.)

6.3 Applets and HTML

ALTHOUGH STAND-ALONE APPLICATIONS are probably more important than applets at this point in the history of JAVA, applets are still widely used. They can do things on Web pages that can't easily be done with other technologies. It is easy to distribute applets to users: The user just has to open a Web page, and the applet is there, with no special installation required (although the user must have an appropriate version of JAVA installed on their computer). And of course, applets are fun; now that the Web has become such a common part of life, it's nice to be able to see your work running on a web page.

The good news is that writing applets is not much different from writing stand-alone applications. The structure of an applet is essentially the same as the structure of the `JFrame`s that were introduced in the previously, and events are handled in the same way in both types of program. So, most of what you learn about applications applies to applets, and *vice versa*.

Of course, one difference is that an applet is dependent on a Web page, so to use applets effectively, you have to learn at least a little about creating Web pages. Web pages are written using a language called HTML (HyperText Markup Language).

6.3.1 JApplet

The `JApplet` class (in package `javax.swing`) can be used as a basis for writing applets in the same way that `JFrame` is used for writing stand-alone applications. The basic `JApplet` class represents a blank rectangular area. Since an applet is not a stand-alone application, this area must appear on a Web page, or in some other environment that knows how to display an applet. Like a `JFrame`, a `JApplet` contains a content pane (and can contain a menu bar). You can add content to an applet either by adding content to its content pane or by replacing the content pane with another component. In my examples, I will generally create a `JPanel` and use it as a replacement for the applet's content pane.

To create an applet, you will write a subclass of `JApplet`. The `JApplet` class defines several instance methods that are unique to applets. These methods are called by the applet's environment at certain points during the applet's "life cycle." In the `JApplet` class itself, these methods do nothing; you can override these methods in a subclass. The most important of these special applet methods is **`public void`** `init()`.

An applet's `init()` method is called when the applet is created. You can use the `init()` method as a place where you can set up the physical structure of the applet and the event handling that will determine its behavior. (You can also do some initialization in the constructor for your class, but there are certain aspects of the applet's environment that are set up after its constructor is called but before the init() method is called, so there are a few operations that will work in the init() method but will not work in the constructor.) The other applet life-cycle methods are start(), stop(), and destroy(). I will not use these methods for the time being and will not discuss them here except to mention that destroy() is called at the end of the applet's lifetime and can be used as a place to do any necessary cleanup, such as closing any windows that were opened by the applet.

With this in mind, we can look at our first example of a JApplet. It is, of course, an applet that says "Hello World!". To make it a little more interesting, I have added a button that changes the text of the message, and a state variable, currentMessage that holds the text of the current message. This example is very similar to the stand-alone application HelloWorldGUI2 from the previous section. It uses an event-handling class to respond when the user clicks the button, a panel to display the message, and another panel that serves as a container for the message panel and the button. The second panel becomes the content pane of the applet. Here is the source code for the applet; again, you are not expected to understand all the details at this time:

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

/**
 * A simple applet that can display the messages "Hello World"
 * and "Goodbye World".  The applet contains a button, and it
 * switches from one message to the other when the button is
 * clicked.
 */
public class HelloWorldApplet extends JApplet {

   private String currentMessage = "Hello World!";
   private MessageDisplay displayPanel;

   private class MessageDisplay extends JPanel {//Defines the display panel.
      public void paintComponent(Graphics g) {
         super.paintComponent(g);
         g.drawString(currentMessage, 20, 30);
      }
   }

   private class ButtonHandler implements ActionListener {//The event listener.
      public void actionPerformed(ActionEvent e) {
         if (currentMessage.equals("Hello World!"))
            currentMessage = "Goodbye World!";
         else
            currentMessage = "Hello World!";
         displayPanel.repaint(); // Paint display panel with new message.
      }
   }
   /**
    * The applet's init() method creates the button and display panel and
    * adds them to the applet, and it sets up a listener to respond to
    * clicks on the button.
    */
   public void init() {

      displayPanel = new MessageDisplay();
      JButton changeMessageButton = new JButton("Change Message");
      ButtonHandler listener = new ButtonHandler();
      changeMessageButton.addActionListener(listener);

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

      setContentPane(content);
   }

}
```

You should compare this class with HelloWorldGUI2.java from the previous section. One subtle difference that you will notice is that the member variables and nested classes in this example are non-static. Remember that an applet is an object. A single class can be used to make several applets, and each of those applets will need its own copy of the applet data, so the member variables in which the data is stored must be non-static instance variables. Since the variables are non-static, the two nested classes, which use those variables, must also be non-static. (Static nested classes cannot access non-static member variables in the containing class) Remember the basic rule for deciding whether to make a nested class static: If it needs access to any instance variable or instance method in the containing class, the nested class must be non-static; otherwise, it can be declared to be **static**.

You can try out the applet itself. Click the "Change Message" button to switch the message back and forth between "Hello World!" and "Goodbye World!":

6.3.2 Reusing Your JPanels

Both applets and frames can be programmed in the same way: Design a JPanel, and use it to replace the default content pane in the applet or frame. This makes it very easy to write two versions of a program, one which runs as an applet and one which runs as a frame. The idea is to create a subclass of JPanel that represents the content pane for your program; all the hard programming work is done in this panel class. An object of this class can then be used as the content pane either in a frame or in an applet. Only a very simple main() program is needed to show your panel in a frame, and only a very simple applet class is needed to show your panel in an applet, so it's easy to make both versions.

As an example, we can rewrite HelloWorldApplet by writing a subclass of JPanel. That class can then be reused to make a frame in a standalone application. This class is very similar to HelloWorldApplet, but now the initialization is done in a constructor instead of in an init() method:

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

public class HelloWorldPanel extends JPanel {

   private String currentMessage = "Hello World!";
   private MessageDisplay displayPanel;

   private class MessageDisplay extends JPanel {//Defines the display panel.
      public void paintComponent(Graphics g) {
         super.paintComponent(g);
         g.drawString(currentMessage, 20, 30);
      }
   }

   private class ButtonHandler implements ActionListener {//The event listener.
      public void actionPerformed(ActionEvent e) {
         if (currentMessage.equals("Hello World!"))
            currentMessage = "Goodbye World!";
         else
            currentMessage = "Hello World!";
         displayPanel.repaint(); // Paint display panel with new message.
      }
   }

   /**
    * The constructor creates the components that will be contained inside this
    * panel, and then adds those components to this panel.
    */
   public HelloWorldPanel() {

      displayPanel = new MessageDisplay();  // Create the display subpanel.

      JButton changeMessageButton = new JButton("Change Message"); // The button.
      ButtonHandler listener = new ButtonHandler();
      changeMessageButton.addActionListener(listener);

      setLayout(new BorderLayout());  // Set the layout manager for this panel.
      add(displayPanel, BorderLayout.CENTER);  // Add the display panel.
      add(changeMessageButton, BorderLayout.SOUTH);  // Add the button.

   }

}
```

Once this class exists, it can be used in an applet. The applet class only has to create an object of type HelloWorldPanel and use that object as its content pane:

```
import javax.swing.JApplet;

public class HelloWorldApplet2 extends JApplet {
   public void init() {
      HelloWorldPanel content = new HelloWorldPanel();
      setContentPane(content);
   }
}
```

Similarly, its easy to make a frame that uses an object of type HelloWorldPanel as its content pane:

```
import javax.swing.JFrame;

public class HelloWorldGUI3 {

   public static void main(String[] args) {
      JFrame window = new JFrame("GUI Test");
      HelloWorldPanel content = new HelloWorldPanel();
      window.setContentPane(content);
      window.setSize(250,100);
      window.setLocation(100,100);
      window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      window.setVisible(true);
   }

}
```

One new feature of this example is the line

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

This says that when the user closes the window by clicking the close box in the title bar of the window, the program should be terminated. This is necessary because no other way is provided to end the program. Without this line, the default close operation of the window would simply hide the window when the user clicks the close box, leaving the program running. This brings up one of the difficulties of reusing the same panel class both in an applet and in a frame: There are some things that a stand-alone application can do that an applet can't do. Terminating the program is one of those things. If an applet calls System.exit() , it has no effect except to generate an error.

Nevertheless, in spite of occasional minor difficulties, many of the GUI examples in this book will be written as subclasses of JPanel that can be used either in an applet or in a frame.

6.3.3 Applets on Web Pages

The <applet> tag can be used to add a JAVA applet to a Web page. This tag must have a matching </applet>. A required modifier named code gives the name of the compiled class file that contains the applet class. The modifiers height and width are required to specify the size of the applet, in pixels. If you want the applet to be centered on the page, you can put the applet in a paragraph with center alignment So, an applet tag to display an applet named HelloWorldApplet centered on a Web page would look like this:

```
<p align=center>
<applet code="HelloWorldApplet.class" height=100 width=250>
</applet>
</p>
```

This assumes that the file HelloWorldApplet.**class** is located in the same directory with the HTML document. If this is not the case, you can use another modifier, codebase, to give the URL of the directory that contains the class file. The value of code itself is always just a class, not a URL.

If the applet uses other classes in addition to the applet class itself, then those class files must be in the same directory as the applet class (always assuming that your classes are all in the "default package"; see Subection2.6.4). If an applet requires more than one or two class files, it's a good idea to collect all the class files into a single jar file. Jar files are "archive files" which hold a number of smaller files. If your class files are in a jar archive, then you have to specify the name of the jar file in an archive modifier in the <applet> tag, as in

```
<applet code="HelloWorldApplet.class" archive="HelloWorld.jar"
height=50...
```

Applets can use applet parameters to customize their behavior. Applet parameters are specified by using <param> tags, which can only occur between an <applet> tag and the closing </applet>. The param tag has required modifiers named name and value, and it takes the form

```
<param name= ‘‘param–name’’  value=‘‘param–value’’>
```

The parameters are available to the applet when it runs. An applet can use the predefined method getParameter() to check for parameters specified in param tags. The getParameter() method has the following interface:

```
String getParameter(String paramName)
```

The parameter paramName corresponds to the param–name in a param tag. If the specified paramName occurs in one of the param tags, then getParameter(paramName) returns the associated param–value. If the specified paramName does not occur in any param tag, then getParameter(paramName) returns the value **null**. Parameter names are case-sensitive, so you cannot use "size" in the param tag and ask for "Size" in getParameter. The getParameter() method is often called in the applet's init() method. It will not work correctly in the applet's constructor, since it depends on information about the applet's environment that is not available when the constructor is called.

Here is an example of an applet tag with several params:

```
<applet code="ShowMessage.class" width=200 height=50>
    <param name="message" value="Goodbye World!">
    <param name="font" value="Serif">
    <param name="size" value="36">
</applet>
```

The ShowMessage applet would presumably read these parameters in its init() method, which could go something like this:

```
String message;  // Instance variable: message to be displayed.
String fontName; // Instance variable: font to use for display.
int fontSize;    // Instance variable: size of the display font.

public void init() {
    String value;
    value = getParameter("message"); // Get message param, if any.
    if (value == null)
       message = "Hello World!";  // Default value, if no param is present.
    else
       message = value;  // Value from PARAM tag.
    value = getParameter("font");
    if (value == null)
       fontName = "SansSerif";  // Default value, if no param is present.
    else
       fontName = value;
    value = getParameter("size");
    try {
       fontSize = Integer.parseInt(value);  // Convert string to number.
    }
    catch (NumberFormatException e) {
       fontSize = 20; // Default value, if no param is present, or if
    }                 //   the parameter value is not a legal integer.
     .
     .
     .
```

Elsewhere in the applet, the instance variables message, fontName, and fontSize would be used to determine the message displayed by the applet and the appearance of that message. Note that the value returned by getParameter() is always a String. If the param represents a numerical value, the string must be converted into a number, as is done here for the size parameter.

6.4 Graphics and Painting

EVERTHING 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.

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 – although not top-level components like JApplet and 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 applet or frame. 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 MessageDisplay class in the example HelloWorldApplet.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 applet or frame. 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 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 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 avoid it for the most part. But I have found it convenient to use getGraphics() in a few cases.

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 almost always 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.
}
```

Most components do, in fact, do all drawing operations in their paintComponent() methods. 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; this method is meant to be called only by the system. 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. 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.

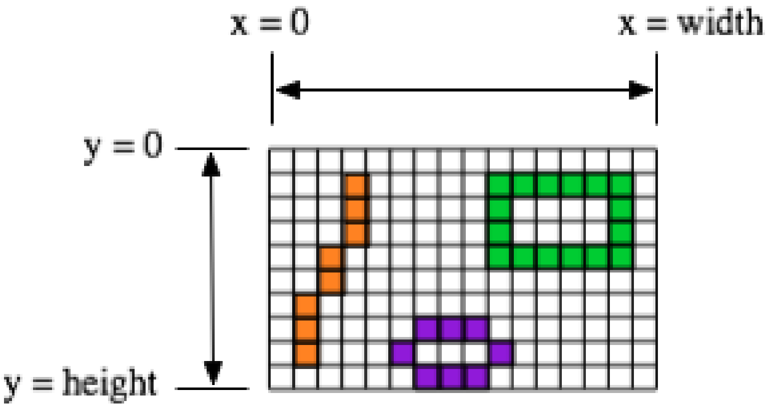
Note that the system can also call paintComponent() for other reasons. It is called when the component first appears on the screen. It will also be called if the component is resized or if it is covered up by another window and then uncovered. The system does not save a copy of the component's contents when it is covered. When it is uncovered, the component is responsible for redrawing itself. (As you will see, 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 about the state of the component in its instance variables. 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 did make modifications directly, you would still have to make the paintComponent() method aware of them in some way so that it will be able 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.

6.4.1 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 16-by-10 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.

6.4.2 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 method 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 method. 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. In the following applet, you can use the scroll bars to control the RGB and HSB values of a color. A sample of the color is shown on the right side of the applet.

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 method 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.

Foreground and background colors can be set by the instance methods setForeground(c) and setBackground(c), 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.

6.4.3 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 10 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 has an associated font that can be set with the setFont(font) instance method, 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.

6.4.4 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.

Here is a list of some of the most important drawing methods. 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.

- 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 position 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.

- `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 hangs 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)`.

- `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);`", where `g` is a graphics context for the component. If you use
  "`g.drawRect(0, 0, getWidth(), getHeight());`", then the right and bottom edges of the rectangle will be drawn *outside* the component.

- `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.

- `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.

- `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.

- `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`.

- `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());`"

- `fillOval(int x, int y, int width, int height)`
  Draws a filled-in oval.

- `fillRoundRect(int x, int y, int width, int height, int xdiam, int ydiam)`
  Draws a filled-in rounded rectangle.

- `fill3DRect(int x, int y, int width, int height, boolean raised)`
  Draws a filled-in three-dimensional rectangle.

- `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.

6.4.5 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. The panel can then be used in either an applet or a frame. 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. Here's an applet that uses the panel as its content pane:

The source 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 calls `g.setFont()` to select that font for drawing the text. 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, "`−50 + (int)(Math.random() * (width+40))`" 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 of the applet.

Here is the complete source code for the `RandomStringsPanel`

```
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
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.
 * <p>This panel is meant to be used as the content pane in
 * either an applet or a frame.
 */
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.

      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
```

This class defines a panel, which is not something that can stand on its own. To see it on the screen, we have to use it in an applet or a frame. Here is a simple applet class that uses a RandomStringsPanel as its content pane:

```
import javax.swing.JApplet;

/**
 * A RandomStringsApplet displays 25 copies of a string, using random colors,
 * fonts, and positions for the copies.  The message can be specified as the
 * value of an applet param with name "message."  If no param with name
 * "message" is present, then the default message "Java!" is displayed.
 * The actual content of the applet is an object of type RandomStringsPanel.
 */
public class RandomStringsApplet extends JApplet {

   public void init() {
      String message = getParameter("message");
      RandomStringsPanel content = new RandomStringsPanel(message);
      setContentPane(content);
   }

}
```

Note that the message to be displayed in the applet can be set using an applet parameter when the applet is added to an HTML document. Remember that to use the applet on a Web page, include both the panel class file, RandomStringsPanel.**class**, and the applet class file, RandomStringsApplet.**class**, in the same directory as the HTML document (or, alternatively, bundle the two class files into a jar file, and put the jar file in the document directory).

Instead of writing an applet, of course, we could use the panel in the window of a stand-alone application. You can find the source code for a main program that does this in the file RandomStringsApp.java.

6.5 Mouse Events

EVENTS ARE CENTRAL TO PROGRAMMING for a graphical user interface. A GUI program doesn't have a main() method 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, 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 is 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 method 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 your GUI program doesn't have a `main()` method, there is a sort of main method running somewhere that executes a loop of the form

```
while the program is still running:
   Wait for the next event to occur
   Call a method 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 method that runs an 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.

6.5.1 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 previously. (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 MyListener implements MouseListener
OR
class MyApplet extends JApplet implements MouseListener
```

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, for example MouseListener, are defined in the package java.awt.event. This means that if you want to work with events, you 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 from the interface;

4. Register the listener object 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. You will see all of these patterns in examples in this textbook.

6.5.2 MouseEvent and MouseListener

The MouseListener interface specifies five different 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 quickly, without moving the mouse. (When the user does this, all three methods – 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 an 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. Here is an applet version of the ClickableRandomStrings program for you to try; when you click the applet, a new set of random strings is displayed:

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 five event-handling methods have empty defintions. We really only want to define a response to mousePressed events, but in order to implement the MouseListener interface, a class **must** define all five methods.)

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:

```
// Create MouseListener object.
RepaintOnClick listener = new RepaintOnClick();

// Create MouseListener object.
panel.addMouseListener(listener);
```

Once this is done, the listener object will be notified of mouse events on the panel. Whenever 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 the RepaintOnClick class was written for use with the 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, RandomStringsPanel 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.

To make this clearer, consider the application version of ClickableRandomStrings. I have included RepaintOnClick as a nested subclass, although it could just as easily be a separate class. The main point is that this program uses the same RandomStringsPanel class that was used in the original program, which did not respond to mouse clicks. The mouse handling has been "bolted on" to an existing class, without having to make any changes at all to that 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 ClickableRandomStringsApp {

   public static void main(String[] args) {
      JFrame window = new JFrame("Random Strings");
      RandomStringsPanel content = new RandomStringsPanel();
      content.addMouseListener( new RepaintOnClick() ); // Register mouse listener.
      window.setContentPane(content);
      window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      window.setLocation(100,75);
      window.setSize(300,240);
      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) { }

   }
}
```

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).

6.5.3 Anonymous Event Handlers

As I mentioned above, it is a fairly common practice to use anonymous nested classes to define listener objects. 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 un-named class and creates an object that belongs to that class. To use the object as a mouse listener, it should 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 for the MouseListener interface is named MouseAdapter. For 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 application: ClickableRandomStringsApp. This version uses an anonymous class based on MouseAdapter to handle mouse events:

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

```
public class ClickableRandomStringsApp {

   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 nested class. As an example, I present HelloWorldGUI4.java. This version is a variation of HelloWorldGUI2.java that uses anonymous nested classes where the original program uses ordinary, named nested classes:

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

/**
 * A simple GUI program that creates and opens a JFrame containing
 * the message "Hello World" and an "OK" button.  When the user clicks
 * the OK button, the program ends.  This version uses anonymous
 * classes to define the message display panel and the action listener
 * object.  Compare to HelloWorldGUI2, which uses nested classes.
 */
public class HelloWorldGUI4 {
   /**
    * The main program creates a window containing a HelloWorldDisplay
    * and a button that will end the program when the user clicks it.
    */
   public static void main(String[] args) {

      JPanel displayPanel = new JPanel() {
            // An anonymous subclass of JPanel that displays "Hello World!".
         public void paintComponent(Graphics g) {
            super.paintComponent(g);
            g.drawString( "Hello World!", 20, 30 );
         }
      };

      JButton okButton = new JButton("OK");

      okButton.addActionListener( new ActionListener() {
            // An anonymous class that defines the listener object.
         public void actionPerformed(ActionEvent e) {
            System.exit(0);
         }
      } );

      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);
   }
}
```

6.6 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 `Button` 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 methods 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. The boolean-valued method, `comp.isEnabled()` can be called 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 Subection6.3.3 for a discussion of fonts.

- `comp.setBackground(color)` and `comp.setForeground(color)` set the background and foreground colors for the component.

- `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 might be a good idea to set a preferred size for it.

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.

6.6.1 JButton

An object of class JButton is a push button that the user can click to trigger some action. You've already seen buttons, 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 constructing the JButton with stopGoButton = **new** JButton(''Go'') 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, "pubic **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. For example, `stopGoButton.setText(‘‘Stop′′)` changes the text displayed on the button to “Stop”. And `stopGoButton.setActionCommand(‘‘sgb′′)` changes the action command associated to this button for action events.

Of course, `JButtons` have all the general `Component` methods, such as `setEnabled()` and `setFont()`. The `setEnabled()` and `setText()` methods of a button 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!”

6.6.2 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("Goodby World!");
```

Since `JLabel` is a subclass of `JComponent`, you can use `JComponent` methods such as `setForeground()` with labels. If you want the background color to have any effect, 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.
```

6.6.3 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 **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 in your program using its setSelected(**boolean**) method. 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, 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 even 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 = ((JCheckBox)cb1).isSelected();
               ... // respond to the change of state
            }
            else if (source == cb2) {
               boolean newState = ((JCheckBox)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.

6.6.4 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 setText() instance method, 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 requestFocus() method. For example, when I discover an error in the user's input, I usually call requestFocus() on the text field that contains the error. This helps the user see where the error occurred and let's the user start typing the correction immediately.

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 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 isn't 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 already inherited from JTextComponent e.g. 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.

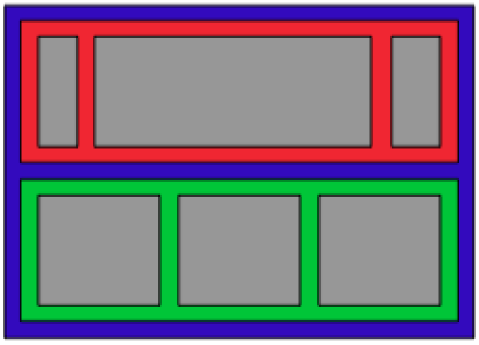
When the user is typing in a JTextField and presses return, an ActionEvent is generated. If you want to respond to such events, you can register an ActionListener with the text field, using the text field's addActionListener() method. (Since a JTextArea can contain multiple lines of text, pressing return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

6.7 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 JApplet or JFrame is an example of a container. The standard class JPanel, which we have mostly used as a drawing surface up till 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:



Three panels, shown in color,
containing six other components,
shown in gray.

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 ordinarily layout is 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 an instance method, setLayout(), that takes a parameter of type LayoutManager and that is used to specify the layout manager that will be responsible for laying out any components that are added to 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.

6.7.1 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.

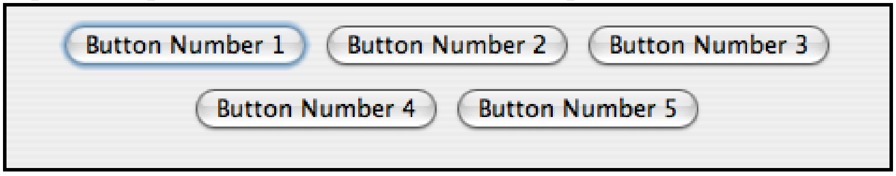
The components in a given row can be either left-aligned, right-aligned, or centered within that row, and there can be horizontal and vertical gaps between components. If the default constructor, "**new** FlowLayout()", is used, then the components on each row will be centered and both the horizontal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

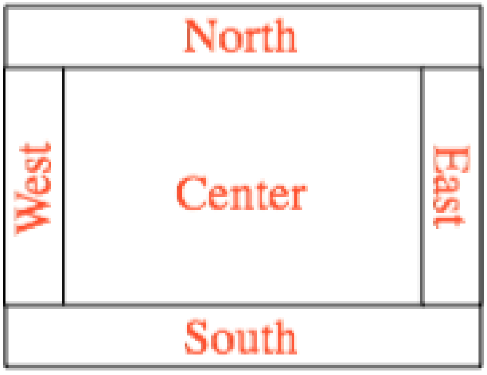
```
panel.add(button1);
panel.add(button2);
panel.add(button3);
panel.add(button4);
panel.add(button5);
```

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 along 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 compompontnts, so that not all five of the possible positions need to be filled.

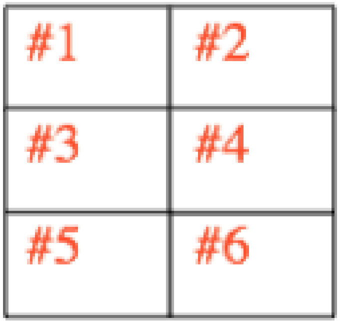
A BorderLayout selects the sizes of its components as follows: The NORTH and SOUTH components (if present) are shown at their preferred heights, but their width is set equal to the full width of the container. The EAST and WEST components are shown at their preferred widths, but their height is set to 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 completely ignored. 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 or JApplet 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 of equal sized rectangles. This illustration shows how the components would be arranged in a grid layout with 3 rows and 2 columns:



#1	#2
#3	#4
#5	#6

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, then you need to 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.

6.7.2 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:

Like the previous example, this example uses a main 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 gray border around the edges of the panel is added with the statement setBorder(BorderFactory.createEmptyBorder(5,5,5,5));.

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 only have one component in each position. In this case, that component 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 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
mainPanel.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 label 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.requestFocus();
      return;
   }


   try {
      String yStr = yInput.getText();
      y = Double.parseDouble(yStr);
   }
```

```
      catch (NumberFormatException e) {
            // The string xStr is not a legal number.
         answer.setText("Illegal data for y.");
         yInput.requestFocus();
         return;
      }

      /* Perfrom 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.)

6.7.3 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 previously have also seen Deck, Hand, and Card classes that are used in the game 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. Try it! See what happens if you click on one of the buttons at a time when it doesn't make sense to do so.

The complete source code for this example is in the file HighLowGUI.java.

The overall structure of the main panel in this example should be 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 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 layout are empty.

The drawing surface is defined by a nested class named CardPanel, which is a 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 another 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 Subection6.5.4.) 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 Subection5.4.3 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. CardPanel 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 show 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() method. 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 actionPerformmed() when user clicks "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) {
       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() method decides where to draw each card and calls this method to do the drawing. You can check out all the details in the source code, HighLowGUI.java.

One further note on the programming of this example: The source code defines HighLowGUI as a subclass of JPanel. The class contains a main() method so that it can be run as a stand-alone application; the main() method simply opens a window that uses a panel of type JPanel as its content pane. In addition, I decided to write an applet version of the program as a static nested class named Applet inside the HighLowGUI class. Since this is a nested class, its full name is HighLowGUI.Applet and the class file produced when the code is compiled is HighLowGUI\$Applet.**class**. This class is used for the applet version of the program shown above. The <applet> tag lists the class file for the applet as code=′′HighLowGUI\$Applet.**class**′′. This is admittedly an unusual way to organize the program, and it is probably more natural to have the panel, applet, and stand-alone program defined in separate classes. However, writing the program in this way does show the flexibility of JAVA classes.

Simple dialogs are created by static methods in the class JOptionPane. This class includes many methods for making dialog boxes, but they are all variations on the three basic types shown here: 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.

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, 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.

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:

```
    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.

If you would like to see how dialogs are created and used in the sample applet, you can find the source code in the file SimpleDialogDemo.java.

6.8 Images and Resources

WE HAVE SEEN HOW TO USE THE GRAPHICS class to draw on a GUI component that is visible on the computer’s screen. Often, however, it is useful to be able to create a drawing **off-screen** , in the computer’s memory. It is also important to be able to work with images that are stored in files.

To a computer, an image is just a set of numbers. The numbers specify the color of each pixel in the image. The numbers that represent the image on the computer’s screen are stored in a part of memory called a frame buffer. Many times each second, the computer’s video card reads the data in the frame buffer and colors each pixel on the screen according to that data. Whenever the computer needs to make some change to the screen, it writes some new numbers to the frame buffer, and the change appears on the screen a fraction of a second later, the next time the screen is redrawn by the video card.

Since it's just a set of numbers, the data for an image doesn't have to be stored in a frame buffer. It can be stored elsewhere in the computer's memory. It can be stored in a file on the computer's hard disk. Just like any other data file, an image file can be downloaded over the Internet. Java includes standard classes and methods that can be used to copy image data from one part of memory to another and to get data from an image file and use it to display the image on the screen.

6.8.1 Images

The class java.awt.Image represents an image stored in the computer's memory. There are two fundamentally different types of Image. One kind represents an image read from a source outside the program, such as from a file on the computer's hard disk or over a network connection. The second type is an image created by the program. I refer to this second type as an off-screen canvas. An off-screen canvas is region of the computer's memory that can be used as a drawing surface. It is possible to draw to an offscreen image using the same Graphics class that is used for drawing on the screen.

An Image of either type can be copied onto the screen (or onto an off-screen canvas) using methods that are defined in the Graphics class. This is most commonly done in the paintComponent() method of a JComponent. Suppose that g is the Graphics object that is provided as a parameter to the paintComponent() method, and that img is of type Image. Then the statement "`g.drawImage(img, x, y, this);`" will draw the image img in a rectangular area in the component. The integer-valued parameters x and y give the position of the upper-left corner of the rectangle in which the image is displayed, and the rectangle is just large enough to hold the image. The fourth parameter, this, is the special variable that refers to the JComponent itself. This parameter is there for technical reasons having to do with the funny way Java treats image files. For most applications, you don't need to understand this, but here is how it works: g.drawImage() does not actually draw the image in all cases. It is possible that the complete image is not available when this method is called; this can happen, for example, if the image has to be read from a file. In that case, g.drawImage() merely **initiates** the drawing of the image and returns immediately. Pieces of the image are drawn later, asynchronously, as they become available. The question is, **how** do they get drawn? That's where the fourth parameter to the drawImage method comes in. The fourth parameter is something called an ImageObserver. When a piece of the image becomes available to be drawn, the system will inform the ImageObserver, and that piece of the image will appear on the screen. Any JComponent object can act as an ImageObserver. The drawImage method returns a boolean value to indicate whether the image has actually been drawn or not when the method returns. When drawing an image that you have created in the computer's memory, or one that you are sure has already been completely loaded, you can set the ImageObserver parameter to null.

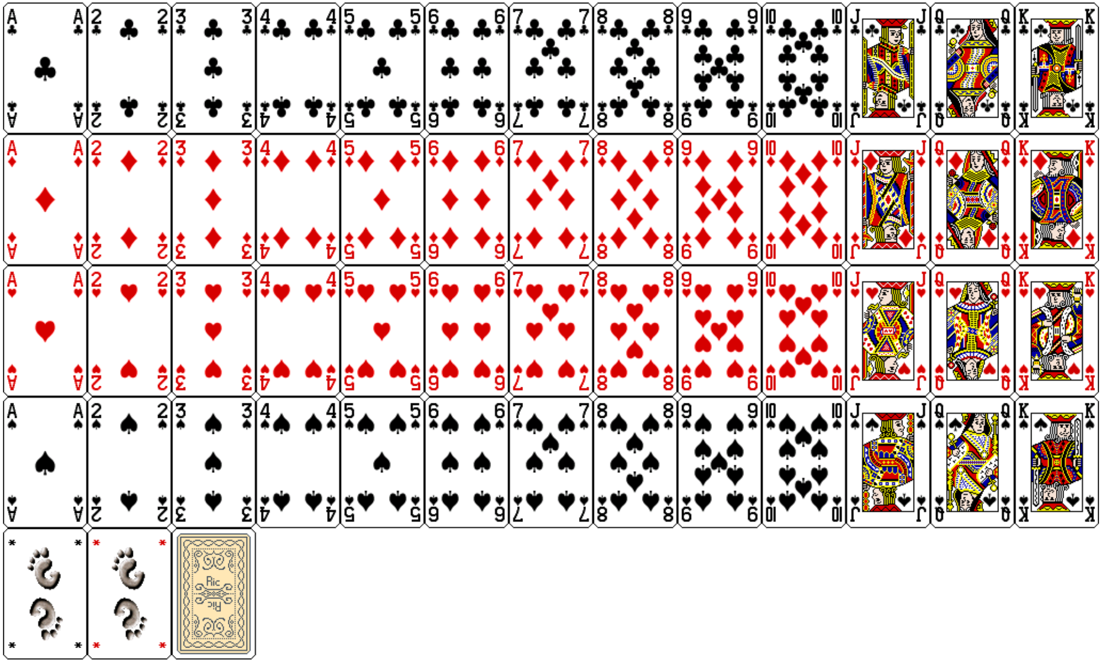
There are a few useful variations of the drawImage() method. For example, it is possible to scale the image as it is drawn to a specified width and height. This is done with the command

```
g.drawImage(img, x, y, width, height, imageObserver);
```

The parameters width and height give the size of the rectangle in which the image is displayed. Another version makes it possible to draw just part of the image. In the command:

```
g.drawImage(img, dest_x1,   dest_y1,   dest_x2,   dest_y2,
            source_x1, source_y1, source_x2, source_y2, imageObserver);
```

the integers source_x1, source_y1, source_x2, and source_y2 specify the top-left and bottom-right corners of a rectangular region in the source image. The integers dest_x1, dest_y1, dest_x2, and dest_y2 specify the corners of a region in the destination graphics context. The specified rectangle in the image is drawn, with scaling if necessary, to the specified rectangle in the graphics context. For an example in which this is useful, consider a card game that needs to display 52 different cards. Dealing with 52 image files can be cumbersome and inefficient, especially for downloading over the Internet. So, all the cards might be put into a single image:



(This image is from the Gnome desktop project, `http://www.gnome.org`, and is shown here much smaller than its actual size.) Now, only one Image object is needed. Drawing one card means drawing a rectangular region from the image. This technique is used in a variation of the sample program HighLowGUI.java. In the original version, the cards are represented by textual descriptions such as “King of Hearts.” In the new version, HighLowWithImages.java, the cards are shown as images. Here is an applet version of the program:

In the program, the cards are drawn using the following method. The instance variable cardImages is a variable of type Image that represents the image that is shown above, containing 52 cards, plus two Jokers and a face-down card. Each card is 79 by 123 pixels. These numbers are used, together with the suit and value of the card, to compute the corners of the source rectangle for the drawImage() command:

```
/**
 * Draws a card in a 79x123 pixel rectangle with its
 * upper left corner at a specified point (x,y).  Drawing the card
 * requires the image file "cards.png".
 * @param g The graphics context used for drawing the card.
 * @param card The card that is to be drawn.  If the value is null, then a
 * face-down card is drawn.
 * @param x the x-coord of the upper left corner of the card
 * @param y the y-coord of the upper left corner of the card
 */
public void drawCard(Graphics g, Card card, int x, int y) {
   int cx;    // x-coord of upper left corner of the card inside cardsImage
   int cy;    // y-coord of upper left corner of the card inside cardsImage
   if (card == null) {
      cy = 4*123;   // coords for a face-down card.
      cx = 2*79;
   }
   else {
      cx = (card.getValue()-1)*79;
      switch (card.getSuit()) {
      case Card.CLUBS:
         cy = 0;
         break;
      case Card.DIAMONDS:
         cy = 123;
         break;
      case Card.HEARTS:
         cy = 2*123;
         break;
      default:  // spades
         cy = 3*123;
         break;
      }
   }
   g.drawImage(cardImages,x,y,x+79,y+123,cx,cy,cx+79,cy+123,this);
}
```

I will tell you later in this section how the image file, cards.png, can be loaded into the program.

6.8.2 Image File I/O

The class javax.imageio.ImageIO makes it easy to save images from a program into files and to read images from files into a program. This would be useful in a program such as PaintWithOffScreenCanvas, so that the users would be able to save their work and to open and edit existing images. (See Exercise12.1.)

There are many ways that the data for an image could be stored in a file. Many standard formats have been created for doing this. Java supports at least three standard image formats: PNG, JPEG, and GIF. (Individual implementations of Java might support more.) The JPEG format is "lossy," which means that the picture that you get when you read a JPEG file is only an approximation of the picture that was saved. Some information in the picture has been lost. Allowing some information to be lost makes it possible to compress the image into a lot fewer bits than would otherwise be necessary. Usually, the approximation is quite good. It works best for photographic images and worst for simple line drawings. The PNG format, on the other hand is "lossless," meaning that the picture in the file is an exact duplicate of the picture that was saved. A PNG file is compressed, but not in a way that loses information. The compression works best for images made up mostly of large blocks of uniform color; it works worst for photographic images. GIF is an older format that is limited to just 256 colors in an image; it has mostly been superseded by PNG.

Suppose that image is a BufferedImage. The image can be saved to a file simply by calling ImageIO.write(image, format, file) where format is a String that specifies the image format of the file and file is a File that specifies the file that is to be written. The format string should ordinarily be either "PNG" or "JPEG", although other formats might be supported.

ImageIO.write() is a static method in the ImageIO class. It returns a boolean value that is false if the image format is not supported. That is, if the specified image format is not supported, then the image is **not** saved, but no exception is thrown. This means that you should always check the return value! For example:

```
boolean hasFormat = ImageIO.write(OSC,format,selectedFile);
if ( ! hasFormat )
   throw new Exception(format + " format is not available.");
```

If the image format **is** recognized, it is still possible that that an IOExcption might be thrown when the attempt is made to send the data to the file.

The ImageIO class also has a static read() method for reading an image from a file into a program. The method ImageIO.read(inputFile) takes a variable of type File as a parameter and returns a BufferedImage. The return value is null if the file does not contain an image that is stored in a supported format. Again, no exception is thrown in this case, so you should always be careful to check the return value. It is also possible for an IOException to occur when the attempt is made to read the file. There is another version of the read() method that takes an InputStream instead of a file as its parameter, and a third version that takes a URL.

Earlier in this section, we encountered another method for reading an image from a URL, the createImage() method from the Toolkit class. The difference is that ImageIO.read() reads the image data completely and stores the result in a BufferedImage. On the other hand, createImage() does not actually read the data; it really just stores the image location and the data won't be read until later, when the image is used. This has the advantage that the createImage() method itself can complete very quickly. ImageIO.read(), on the other hand, can take some time to execute.

Chapter 7

A Solitaire Game - Klondike

In this chapter will build a version of the Solitaire game. We'll use the case study investigate the object-oriented concepts of encapsulation, inheritance, and polymorphism. The game is inspired by Timothy Budd's version in his book AN INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING.

7.1 Klondike Solitaire

The most popular solitare game is called **klondike**. It can be described as follows:

The layout of the game is shown in the figure below. A single standard pack of 52 cards is used. (i.e. 4 suits (spades ♠, diamonds ♢, hearts ♡, clubs ♣) and 13 cards (13 ranks) in each suit.).

The tableau, or playing table, consists of 28 cards in 7 piles. The first pile has 1 card, the second 2, the third 3, and so on up to 7. The top card of each pile is initially face up; all other cards are face down.

The suit piles (sometimes called foundations) are built up from aces to kings in suits. They are constructed above the tableau as the cards become available. The object of the game is to build all 52 cards into the suit piles.

The cards that are not part of the tableau are initially all in the deck. Cards in the deck are face down, and are drawn one by one from the deck and placed, face up, on the discard pile. From there, they can be moved onto either a tableau pile or a foundation. Cards are drawn from the deck until the pile is empty; at this point, the game is over if no further moves can be made.

Cards can be placed on a tableau pile only on a card of next-higher rank and opposite color. They can be placed on a foundation only if they are the same suit and next higher card or if the foundation is empty and the card is an ace. Spaces in the tableau that arise during play can be filled only by kings.

The topmost card of each tableau pile and the topmost card of the discard pile are always available for play. The only time more than one card is moved is when an entire collection of face-up cards from a tableau (called a build) is moved to another tableau pile. This can be done if the bottommost card of the build can be legally played on the topmost card of the destination. Our initial game will not support the transfer of a build. The topmost card of a tableau is always face up. If a card is moved

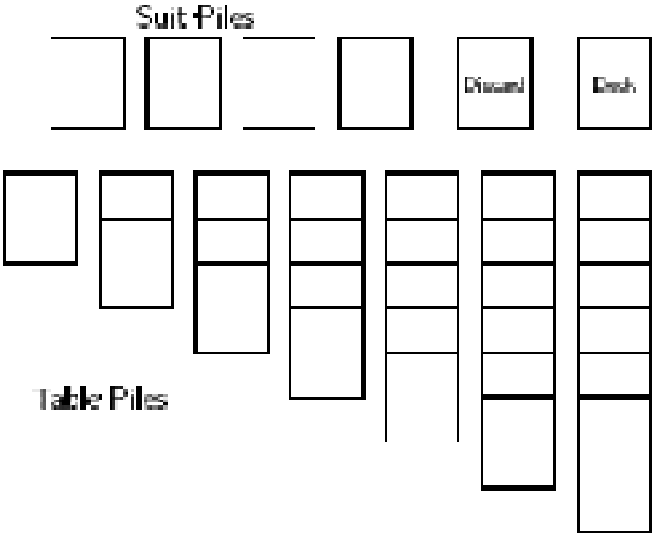


Figure 7.1: Layout of the Solitaire Game

from a tableau, leaving a face-down card on the top, the latter card can be turned face up.

7.2 Card Games

In this section and the next we will explore games that employ playing cards, and use them to build our simplified game of Klondike Solitaire.

To start off we will program two classes, a Card class and a Deck class. These two classes will be useful in almost all card games. Create and new project (CardGames is good name) and write these classes in a package called cardGames.

The Card class

The aim is to build an ABSTRACTION of a playing card. Objects of type Card represent a single playing card. The class has the following responsibilites:

Know its suit, rank and whether it is black or red

Create a card specified by rank and suit

Know if it is face down or up

Display itself (face up or down)

Flip itself (change from face down to face up and vice versa)

Your tasks is to design the Card class and program it. It is also necessary to test your class.

Using Images

In order to program the class, we need to use images of cards.

There are several ways to work with images. Heres a quick how-to describing one way...

(a) Copy the images folder into the project folder. It should be copied into the top level of the CardGames folder.

(b) Using an image is a three step process:

* Declare a variable of type Image e.g. Image backImage;
* Read an image into this variable: (This must be done within a **try/catch** block and assumes the images are stored in the images folder in the project.)

```
try{
  backImage = ImageIO.read(new File("images/b1fv.gif"));

}
catch (IOException i){
  System.err.println("Image load error");
}
```

* Draw the image (Off course, you draw method will be different since you have to worry about whether the card is face up and face down and the image you draw depends on the particular card.):

```
public void draw(Graphics g, int x, int y) {
  g.drawImage(backImage,x,y,null); }
```

(c) The naming convention of the image files is straight forward: 'xnn.gif' is the format were 'x' is a letter of the suit (s=spades ♠, d=diamonds ♢, h=hearts ♡, c=clubs ♣) and 'nn' is a one or two digit number representing the card's rank (1=ACE, 2-10=cards 2 to 10, 11=JACK, 12=QUEEN, 13=KING). e.g. c12 is the Queen of clubs; d1 is the Ace of Diamonds; h8=8 of hearts. There are two images of the back of a card (b1fv.gif and b2fv.gif).

The testing of the Card class can be done by setting up a test harness. This could simply be a main method in the Card class like this one. You will off course make changes to this to do various tests.:

```
public static void main(String[] args) {

  class Panel extends JPanel { //a method local inner class
    Card c;
    Panel(){ c = new Card(1,13); }

    public void PanelTest(){ //method to test Cards
      repaint();      c.flip();      repaint();
    }
    public void paintComponent(Graphics g){
       super.paintComponent(g);
       c.draw(g,20,10);
    }
  } \\end of class Panel
```

```
    JFrame frame = new JFrame();
    frame.setSize(new Dimension(500,500));
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Panel p = new Panel();
    frame.setContentPane(p);
    frame.show();
    p.PanelTest();
  }\\end of main method
```

7.2.1 The CardNames Interface

The CardNames class is an interface defining names.

```
public interface CardNames {
        public static final int heart = 0;
        public static final int diamond = 1;
        public static final int club = 2;
        public static final int spade = 3;
        public static final int ace = 1;
        public static final int jack = 11;
        public static final int queen = 12;
        public static final int king = 13;
        public static final int red = 0;
        public static final int black = 1;
}
```

Its a convenience class that allows us to use these names in a consistent manner. Thus, we can use the name CardNames.ace throughout the program consistently (i. e. Different parts of the program will mean the same thing when they say CardNames.ace).

7.2.2 The Deck class

This class is meant to represent a deck of 52 cards. (A Deck is composed of 52 Cards). Its responsibilities are:

Create a deck of 52 cards

Know the cards in the deck

Shuffle a deck

Deal a card from the deck

Know how many cards are in the deck

Design, write and test the Deck class.

7.3 Implementation of Klondike

To program the game, we notice that we basically need to keep track of several piles of cards. The piles have similar functionality, so inheritance is strongly suggested. What we do is write all the common functionality in a base class called CardPile. We then specialise this class to create the concrete classes for each pile.

A class diagram for this application is shown above:

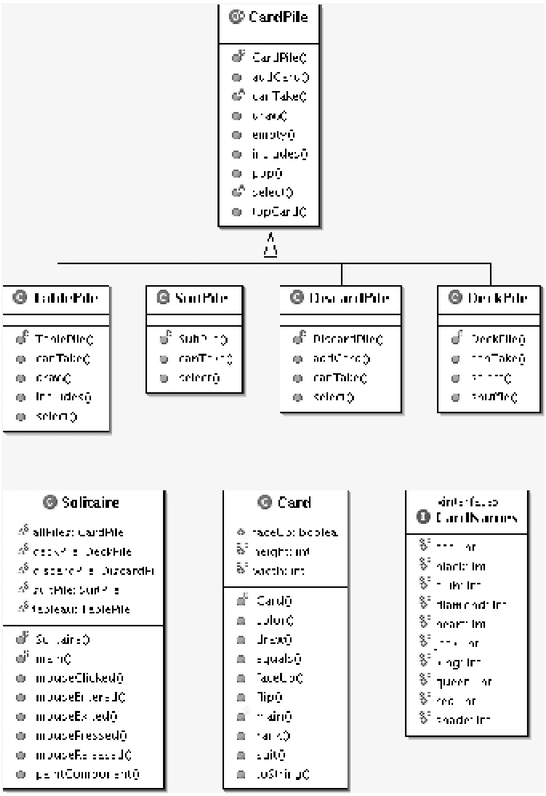


Figure 7.2: Class diagram for the Solitaire app

7.3.1 The CardPile class (the base class)

```
package solitaire;

import java.awt.Graphics;
import java.util.LinkedList;
import java.util.List;

public abstract class CardPile {

        protected List pile;
        protected int x;
        protected int y;

        /** * Make an Empty Pile          */
        public CardPile(int x, int y) {
                pile = new LinkedList();
                this.x = x;
                this.y = y;
        }

        public boolean empty(){
           return pile.isEmpty();
        }
        public Card topCard() {
           if (!empty())
                return (Card)pile.get(pile.size()-1);
           else
                return null;
        }

        public Card pop() {
           if (!empty())
              return (Card)pile.remove(pile.size()-1);
           else
              return null;
        }

        public boolean includes(int tx, int ty) {
           return x<=tx && tx <= x + Card.width
                         && y <= ty && ty <= y + Card.height;
        }

        public void addCard(Card aCard){
                pile.add(aCard);
        }

        public void draw (Graphics g){
           if (empty()) {
                g.drawRect(x,y,Card.width,Card.height);
           }
           else
                topCard().draw(g,x,y);
        }

        public abstract boolean canTake(Card aCard);

        public abstract void select ();
}
```

Notice that this class is abstract. It has three protected attributes (What does protected mean?). The x and y are coordinates of this pile on some drawing surface and the pile attribute is Collection of Cards. Most of the methods are self explanatory ;).

* The includes method is given a point (a coordinate) and returns true if this point is contained within the space occupied by the cards in the pile. We intend to use this method to tell us if the user has clicked on this particular pile of cards. The idea is to get the coordinates of the point the user has clicked on and then ask each pile if this coordinate falls within the space it occupies.

* The canTake abstract method should tell us whether a particular pile of cards can accept a card. Different piles will have different criteria for accepting a Card. For example, suit piles will accept a card if it is the same suit as all others in the pile and if its rank is one more that its topCard. The table piles will accept a card if its suit is opposite in color and its rank is one less than the pile's topCard.

* The select abstract method is the action this pile takes if it can accept a Card. Usually, this means adding it to its pile and making the new Card the topCard.

7.3.2 The Solitaire class

The Solitaire class is the one that runs. It creates and maintains the different piles of cards. Notice that most of its attributes are static and visible to other classes in the package. Study it carefully and make sure you understand it fully (FULLY!) before you continue.

```
package solitaire;

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

public class Solitaire extends JPanel implements MouseListener {

        static DeckPile deckPile;
        static DiscardPile discardPile;
        static TablePile tableau[];
        static SuitPile suitPile[];
        static CardPile allPiles[];

        public Solitaire(){
                setBackground(Color.green);
                addMouseListener(this);
                allPiles = new CardPile[13];
                suitPile = new SuitPile[4];
                tableau = new TablePile[7];

                int deckPos = 600;
                int suitPos = 15;
                allPiles[0] = deckPile = new DeckPile(deckPos, 5);
                allPiles[1] = discardPile =
                         new DiscardPile(deckPos - Card.width - 10, 5);
                for (int i = 0; i < 4; i++)
                        allPiles[2+i] = suitPile[i] =
                         new SuitPile(suitPos + (Card.width + 10) * i, 5);
                for (int i = 0; i < 7; i++)
                        allPiles[6+i] = tableau[i] =
                          new TablePile(suitPos + (Card.width + 10) * i,
                                             Card.height + 20, i+1);
                repaint();
        }

        public void paintComponent(Graphics g) {
                super.paintComponent(g);
                for (int i = 0; i < 13; i++)
                        allPiles[i].draw(g);
        }
        public static void main(String[] args) {
                JFrame frame = new JFrame();
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
                frame.setSize(800,600);
                frame.setTitle("Solitaire");

                Solitaire s = new Solitaire();
                frame.add(s);
                frame.validate();
                s.repaint();
        }

        public void mouseClicked(MouseEvent e) {
                int x = e.getX();
                int y = e.getY();
                for (int i = 0; i < 12; i++)
                        if (allPiles[i].includes(x, y)) {
                                allPiles[i].select();
                                repaint();
                        }
        }

        public void mousePressed(MouseEvent e) { }

        public void mouseReleased(MouseEvent e) { }

        public void mouseEntered(MouseEvent e) { }

        public void mouseExited(MouseEvent e) { }
}
```

7.3.3 Completing the Implementation

Write the classes TablePile, SuitPile, DiscardPile, DeckPile. I suggest that you create all the classes first and then work with them one at a time. They all extend the CardPile class. You must take care to consider situations when the pile is empty. The following will guide you in writing these classes:

* **the** DeckPile **Class** This class extends the CardPile class. It must create a full deck of cards (stored in its super class's pile attribute.) The cards should be shuffled after creation (use Collections.shuffle(...)). You never add cards to the DeckPile so its canTake method always returns false. The select method removes a card from the deckPile and adds it to the discardPile (In the Solitaire class).

* **The** DiscardPile **Class** This maintains a pile of cards that do not go into any of the other piles. Override the addCard method to check first if the card is faceUp and flip it if its not. Then add the card to the pile. You never add cards to the DiscardPile so its canTake method always returns false. The select method requires careful thought. Remember that this method runs when the user selects this pile. Now, what happens when the user clicks on the topCard in the discardPile? We must check if any SuitPile (4 of them) or any TablePile

(7 of them) (all in the Solitaire class) can take the card. If any of these piles can take the card we add the Card to that pile. If not, we leave it on the discardPile.

* **The** `SuitPile` **Class** The `select` method is empty (Cards are never removed from this pile). The `canTake` method should return true if the Card is the same suit as all others in the pile and if its rank is one more that its topCard.

* **The** `TablePile` **Class** Write the constructor to initialize the table pile. The constructor accepts three parameters, the x and y coordinates of the pile, and an integer that tell it how many cards it contains. (remember that the first tablePile contains 1 card, the second 2 Cards etc.). It takes Cards from the deckPile. The table pile is displayed differently from the other piles (the cards overlap). We thus need to override the includes the method and the draw method. The `canTake` method is also different. The table piles will accept a card if its suit is opposite in color and its rank is one less than the pile's topCard. The `select` method is similar to the one in `DiscardPile`. We must check if any SuitPile (4 of them) or any TablePile (7 of them) (all in the Solitaire class) can take the card. If any of these piles can take the card we add the Card to that pile otherwise we leave it in this tabePile.

Chapter 8

Generic Programming

Contents

A DATA STRUCTURE IS A COLLECTION OF DATA ITEMS, considered as a unit. For example, a list is a data structure that consists simply of a sequence of items. Data structures play an important part in programming. Various standard data structures have been developed including lists, sets, and trees. Most programming libraries provide built-in data structures that may be used with very little effort from the programmer. Java has the **Collection Framework** that provides standard data structures for use by programmers.

Generic programming refers to writing code that will work for many types of data. The source code presented there for working with dynamic arrays of integers works only for data of type int. But the source code for dynamic arrays of double, String, JButton, or any other type would be almost identical, except for the substitution of one type name for another. It seems silly to write essentially the same code over and over. Java goes some distance towards solving this problem by providing the `ArrayList` class. An `ArrayList` is essentially a dynamic array of values of type Object. Since every class is a subclass of Object, objects of any type can be stored in an `ArrayList`. Java goes even further by providing "parameterized types." The `ArrayList` type can be parameterized, as in "`ArrayList<String>`", to limit the values that can be stored in the list to objects of a specified type. Parameterized types extend Java's basic philosophy of type-safe programming to generic programming.

8.1 Generic Programming in Java

JAVA'S GENERIC PROGRAMMING FEATURES are represented by group of generic classes and interfaces as a group are known as the **Java Collection Framework**. These classes represents various data structure designed to hold Objects can be used with objects of any type. Unfortunately the result is a category of errors that show up only at run time, rather than at compile time. If a programmer assumes that all the items in a data structure are strings and tries to process those items as strings, a run-time error will occur if other types of data have inadvertently been added to the data structure. In JAVA, the error will most likely occur when the program retrieves an Object from the data structure and tries to type-cast it to type `String`. If the object is not actually of type `String`, the illegal type-cast will throw an error of type `ClassCastException`.

JAVA 5.0 introduced parameterized types, such as `ArrayList<String>`. This made it possible to create generic data structures that can be type-checked at compile time rather than at run time. With these data structures, type-casting is not necessary, so `ClassCastExceptions` are avoided. The compiler will detect any attempt to add an object of the wrong type to the data structure; it will report a syntax error and will refuse to compile the program. In Java 5.0, all of the classes and interfaces in the Collection Framework, and even some classes that are not part of that framework, have been parameterized. In this chapter, I will use the parameterized types almost exclusively, but you should remember that their use is not mandatory. It is still legal to use a parameterized class as a non-parameterized type, such as a plain `ArrayList`.

With a Java parameterized class, there is only one compiled class file. For example, there is only one compiled class file, `ArrayList`.class, for the parameterized class `ArrayList`. The parameterized types `ArrayList<String>` and `ArrayList<Integer>` both use the some compiled class file, as does the plain `ArrayList` type. The type parameter—`String` or `Integer`—just tells the compiler to limit the type of object that can be stored in the data structure. The type parameter has no effect at run time and is not even known at run time. The type information is said to be "erased" at run time. This type erasure introduces a certain amount of weirdness. For example, you can't test "**if** (list **instanceof** {ArrayList<String>)" because the **instanceof** operator is evaluated at run time, and at run time only the plain `ArrayList` exists. Even worse, you can't create an array that has base type `ArrayList<String>` using the new operator, as in "**new** `ArrayList<String>(N)`". This is because the new operator is evaluated at run time, and at run time there is no such thing as "`ArrayList<String>`"; only the non-parameterized type `ArrayList` exists at run time.

Fortunately, most programmers don't have to deal with such problems, since they turn up only in fairly advanced programming. Most people who use the **Java Collection Framework** will not encounter them, and they will get the benefits of type-safe generic programming with little difficulty.

8.2 ArrayLists

IN THIS SECTION we discuss `ArrayLists` that are part of the Collection Framework.

Arrays in JAVA have two disadvantages: they have a fixed size and their type must be must be specified when they are created.

The size of an array is fixed when it is created. In many cases, however, the number of data items that are actually stored in the array varies with time. Consider the following examples: An array that stores the lines of text in a word-processing program. An array that holds the list of computers that are currently downloading a page from a Web site. An array that contains the shapes that have been added to the screen by the user of a drawing program. Clearly, we need some way to deal with cases where the number of data items in an array is not fixed.

Specifying the type when arrays are created means that one can only put primitives or objects of the specified into the array—for example, an array of **int** can only hold integers. One way to work around this is to declare `Object` as the type of an array. In this case one can place anything into the array because, in JAVA, every class is a subclass of the class named `Object`. This means that every object can be assigned to a variable of type `Object`. Any object can be put into an array of type `Object[ ]`.

An `ArrayList` serves much the same pupose as arrays do. It allows you to store objects of any type. The `ArrayList` class is in the package `java.util`, so if you want to use it in a program, you should put the directive "**import** `java.util.ArrayList;`" at the beginning of your source code file.

The `ArrayList` class always has a definite size, and it is illegal to refer to a position in the `ArrayList` that lies outside its size. In this, an `ArrayList` is more like a regular array. However, the size of an `ArrayList` can be increased at will. The `ArrayList` class defines many instance methods. I'll describe some of the most useful. Suppose that list is a variable of type `ArrayList`. Then we have:

- `list.size()`–This method returns the current size of the `ArrayList`. The only valid positions in the list are numbers in the range 0 to `list.size()`−1. Note that the size can be zero. A call to the default constructor new `ArrayList()` creates an `ArrayList` of size zero.

- `list.add(obj)`–Adds an object onto the end of the list, increasing the size by 1. The parameter, `obj`, can refer to an object of any type, or it can be **null**.

- `list.get(N)`–returns the value stored at position `N` in the `ArrayList`. `N` must be an integer in the range 0 to `list.size()`−1. If N is outside this range, an error of type `IndexOutOfBoundsException` occurs. Calling this method is similar to referring to `A[N]` for an array, `A`, except you can't use `list.get(N)` on the left side of an assignment statement.

- `list.set(N, obj)`–Assigns the object, obj, to position N in the `ArrayList`, replacing the item previously stored at position N. The integer N must be in the range from 0 to `list.size()`−1. A call to this method is equivalent to the command `A[N] = obj` for an array A.

- `list.remove(obj)`–If the specified object occurs somewhere in the `ArrayList`, it is removed from the list. Any items in the list that come after the removed item are moved down one position. The size of the `ArrayList` decreases by 1. If `obj` occurs more than once in the list, only the first copy is removed.

- `list.remove(N)`–For an integer, N, this removes the N-th item in the `ArrayList`. N must be in the range 0 to `list.size()`−1. Any items in the list that come after the removed item are moved down one position. The size of the `ArrayList` decreases by 1.

- `list.indexOf(obj)`–A method that searches for the object, obj, in the `ArrayList`. If the object is found in the list, then the position number where it is found is returned. If the object is not found, then −1 is returned.

For example, suppose that players in a game are represented by objects of type Player. The players currently in the game could be stored in an ArrayList named players. This variable would be declared as ArrayList players; and initialized to refer to a new, empty ArrayList object with players = **new** ArrayList();. If newPlayer is a variable that refers to a Player object, the new player would be added to the ArrayList and to the game by saying players.add(newPlayer); and if player number i leaves the game, it is only necessary to say players.remove(i);. Or, if player is a variable that refers to the Player that is to be removed, you could say players.remove(player);.

All this works very nicely. The only slight difficulty arises when you use the method players.get(i) to get the value stored at position i in the ArrayList. The return type of this method is Object. In this case the object that is returned by the method is actually of type Player. In order to do anything useful with the returned value, it's usually necessary to type-cast it to type Player by saying:
Player plr = (Player)players.get(i);.

For example, if the Player class includes an instance method makeMove() that is called to allow a player to make a move in the game, then the code for letting every player make a move is

```
for (int i = 0;  i < players.size();  i++) {
    Player plr = (Player)players.get(i);
    plr.makeMove();
}
```

The two lines inside the for loop can be combined to a single line:
((Player)players.get(i)).makeMove();.
This gets an item from the list, type-casts it, and then calls the makeMove() method on the resulting Player. The parentheses around "(Player)players.get(i)" are required because of Java's precedence rules. The parentheses force the type-cast to be performed before the makeMove() method is called.

for−each loops work for ArrayLists just as they do for arrays. But note that since the items in an ArrayList are only known to be Objects, the type of the loop control variable must be Object. For example, the for loop used above to let each Player make a move could be written as the **for**−each loop

```
for ( Object plrObj : players ) {
   Player plr = (Player)plrObj;
   plr.makeMove();
}
```

In the body of the loop, the value of the loop control variable, plrObj, is one of the objects from the list, players. This object must be type-cast to type Player before it can be used.

8.3 Parameterized Types

THE MAIN DIFFERENCE BETWEEN true generic programming and the ArrayList examples in the previous subsection is the use of the type Object as the basic type for objects that are stored in a list. This has at least two unfortunate consequences: First, it makes it necessary to use type-casting in almost every case when an element is retrieved from that list. Second, since any type of object can legally be added to the list, there is no way for the compiler to detect an attempt to add the wrong type of object to the list; the error will be detected only at run time when the object is retrieved from the list and the attempt to type-cast the object fails. Compare this to arrays. An array of type BaseType[ ] can **only** hold objects of type BaseType. An attempt to store an object of the wrong type in the array will be detected by the compiler, and there is no need to type-cast items that are retrieved from the array back to type BaseType.

To address this problem, Java 5.0 introduced parameterized types. ArrayList is an example: Instead of using the plain "ArrayList" type, it is possible to use ArrayList<BaseType>, where BaseType is any object type, that is, the name of a class or of an interface. (BaseType **cannot** be one of the primitive types.)

ArrayList<BaseType> can be used to create lists that can hold only objects of type BaseType. For example, ArrayList<ColoredRect> rects;. declares a variable named rects of type ArrayList<ColoredRect>, and
rects = **new** ArrayList<ColoredRect>();
sets rects to refer to a newly created list that can only hold objects belonging to the class ColoredRect (or to a subclass). The funny-looking "ArrayList<ColoredRect>" is being used here in the same way as an ordinary class name–don't let the "<ColoredRect>" confuse you; it's just part of the name of the type. When a statements such as rects.add(x); occurs in the program, the compiler can check whether x is in fact of type ColoredRect. If not, the compiler will report a syntax error. When an object is retrieved from the list, the compiler knows that the object must be of type ColoredRect, so no type-cast is necessary. You can say simply:
ColoredRect rect = rects.get(i).

You can even refer directly to an instance variable in the object, such as rects.get(i).color. This makes using ArrayList<ColoredRect> very similar to using ColoredRect[ ] with the added advantage that the list can grow to any size. Note that if a for-each loop is used to process the items in rects, the type of the loop control variable can be ColoredRect, and no type-cast is necessary. For example, when using ArrayList<ColoredRect> as the type for the list rects, the code for drawing all the rectangles in the list could be rewritten as:

```
for ( ColoredRect rect : rects ) {
   g.setColor( rect.color );
   g.fillRect( rect.x, rect.y, rect.width, rect.height);
   g.setColor( Color.BLACK );
   g.drawRect( rect.x, rect.y, rect.width − 1, rect.height − 1);
}
```

You can use ArrayList<ColoredRect> anyplace where you could use a normal type: to declare variables, as the type of a formal parameter in a method, or as the return type of a method. ArrayList<ColoredRect> is not considered to be a separate class from ArrayList. An object of type ArrayList<ColoredRect> actually belongs to the class ArrayList, but the compiler restricts the type of objects that can be added to the list.)

The only drawback to using parameterized types is that the base type cannot be a primitive type. For example, there is no such thing as "ArrayList<**int**>". However, this is not such a big drawback as it might seem at first, because of the "*wrapper types*" and "*autoboxing*". A wrapper type such as Double or Integer can be used as a base type for a parameterized type. An object of type ArrayList<Double> can hold objects of type Double. Since each object of type Double holds a value of type double, it's almost like having a list of doubles. If numlist is declared to be of type

ArrayList<Double> and if x is of type double, then the value of x can be added to the list by saying: numlist.add(**new** Double(x));.

Furthermore, because of autoboxing, the compiler will automatically do double-to-Double and Double-to-double type conversions when necessary. This means that the compiler will treat "numlist.add(x)" as begin equivalent to the statement "numlist.add(**new** Double(x))". So, behind the scenes, "numlist.add(x)" is actually adding an object to the list, but it looks a lot as if you are working with a list of doubles.

The ArrayList class is just one of several standard classes that are used for generic programming in Java. We will spend the next few sections looking at these classes and how they are used, and we'll see that there are also generic methods and generic interfaces. All the classes and interfaces discussed in these sections are defined in the package java.util, and you will need an import statement at the beginning of your program to get access to them. (Before you start putting "importăjava.util.*" at the beginning of every program, you should know that some things in java.util have names that are the same as things in other packages. For example, both java.util.List and java.awt.List exist, so it is often better to import the individual classes that you need.)

8.4 The Java Collection Framework

JAVA'S GENERIC DATA STRUCTURES can be divided into two categories: collections and maps. A collection is more or less what it sound like: a collection of objects. An ArrayList is an example of a collection. A map associates objects in one set with objects in another set in the way that a dictionary associates definitions with words or a phone book associates phone numbers with names. In Java, collections and maps are represented by the parameterized interfaces Collection<T> and Map<T,S>. Here, "T" and "S" stand for any type except for the primitive types.

We will discuss only collections in this course.

There are two types of collections: lists and sets. A list is a collection in which the objects are arranged in a linear sequence. A list has a first item, a second item, and so on. For any item in the list, except the last, there is an item that directly follows it. The defining property of a set is that no object can occur more than once in a set; the elements of a set are not necessarily thought of as being in any particular order. The ideas of lists and sets are represented as parameterized interfaces List<T> and Set<T>. These are sub–interfaces of \code{Collection<T>. That is, any object that implements the interface List<T> or Set<T> automatically implements Collection<T> as well. The interface Collection<T> specifies general operations that can be applied to any collection at all. List<T> and Set<T> add additional operations that are appropriate for lists and sets respectively.

Of course, any actual object that is a collection, list, or set must belong to a concrete class that implements the corresponding interface. For example, the class ArrayList<T> implements the interface List<T> and therefore also implements Collection<T>. This means that all the methods that are defined in the list and collection interfaces can be used with, for example, an ArrayList<String> object. We will look at various classes that implement the list and set interfaces in the next section. But before we do that, we'll look briefly at some of the general operations that are available for all collections.

The interface Collection<T> specifies methods for performing some basic operations on any collection of objects. Since "collection" is a very general concept, operations that can be applied to all collections are also very general. They are generic operations in the sense that they can be applied to various types of collections containing various types of objects. Suppose that `coll` is an object that implements the interface `Collection<T>` (for some specific non-primitive type T). Then the following operations, which are specified in the interface `Collection<T>`, are defined for `coll`:

- `coll.size()`–returns an **int** that gives the number of objects in the collection.
- `coll.isEmpty()`–returns a **boolean** value which is true if the size of the collection is 0.
- `coll.clear()`–removes all objects from the collection.
- `coll.add(tobject)`–adds `tobject` to the collection. The parameter must be of type T; if not, a syntax error occurs at compile time. This method returns a boolean value which tells you whether the operation actually modified the collection. For example, adding an object to a Set has no effect if that object was already in the set.
- `coll.contains(object)`–returns a boolean value that is true if object is in the collection. Note that object is **not** required to be of type T, since it makes sense to check whether object is in the collection, no matter what type object has. (For testing equality, null is considered to be equal to itself. The criterion for testing non-null objects for equality can differ from one kind of collection to another.)
- `coll.remove(object)`–removes `object` from the collection, if it occurs in the collection, and returns a boolean value that tells you whether the object was found. Again, `object` is not required to be of type T.
- `coll.containsAll(coll2)`–returns a boolean value that is true if every object in `coll2` is also in the `coll`. The parameter can be any collection.
- `coll.addAll(coll2)`–adds all the objects in `coll2` to `coll`. The parameter, `coll2`, can be any collection of type `Collection<T>`. However, it can also be more general. For example, if T is a class and S is a sub-class of T, then coll2 can be of type `Collection<S>`. This makes sense because any object of type S is automatically of type T and so can legally be added to coll.
- `coll.removeAll(coll2)`–removes every object from `coll` that also occurs in the collection `coll2`. `coll2` can be any collection.
- `coll.retainAll(coll2)`–removes every object from `coll` that **does not occur** in the collection `coll2`. It "retains" only the objects that do occur in `coll2`. `coll2` can be any collection.
- `coll.toArray()`–returns an array of type `Object[ ]` that contains all the items in the collection. The return value can be type-cast to another array type, if appropriate. Note that the return type is `Object[ ]`, not `T[ ]`! However, you can type-cast the return value to a more specific type. For example, if you know that all the items in coll are of type `String`, then `String[])coll.toArray()` gives you an array of Strings containing all the strings in the collection.

Since these methods are part of the Collection<T> interface, they must be defined for every object that implements that interface. There is a problem with this, however. For example, the size of some kinds of collection cannot be changed after they are created. Methods that add or remove objects don't make sense for these collections. While it is still legal to call the methods, an exception will be thrown when the call is evaluated at run time. The type of the exception thrown is UnsupportedOperationException. Furthermore, since Collection<T> is only an interface, not a concrete class, the actual implementation of the method is left to the classes that implement the interface. This means that the semantics of the methods, as described above, are not guaranteed to be valid for all collection objects; they are valid, however, for classes in the Java Collection Framework.

There is also the question of efficiency. Even when an operation is defined for several types of collections, it might not be equally efficient in all cases. Even a method as simple as size() can vary greatly in efficiency. For some collections, computing the size() might involve counting the items in the collection. The number of steps in this process is equal to the number of items. Other collections might have instance variables to keep track of the size, so evaluating size() just means returning the value of a variable. In this case, the computation takes only one step, no matter how many items there are. When working with collections, it's good to have some idea of how efficient operations are and to choose a collection for which the operations that you need can be implemented most efficiently. We'll see specific examples of this in the next two sections.

8.5 Iterators and for-each Loops

THE INTERFACE Collection<T> defines a few basic generic algorithms, but suppose you want to write your own generic algorithms. Suppose, for example, you want to do something as simple as printing out every item in a collection. To do this in a generic way, you need some way of going through an arbitrary collection, accessing each item in turn. We have seen how to do this for specific data structures: For an array, you can use a **for** loop to iterate through all the array indices. For a linked list, you can use a while loop in which you advance a pointer along the list.

Collections can be represented in any of these forms and many others besides. With such a variety of traversal mechanisms, how can we even hope to come up with a single generic method that will work for collections that are stored in wildly different forms? This problem is solved by iterators. An iterator is an object that can be used to traverse a collection. Different types of collections have iterators that are implemented in different ways, but all iterators are **used** in the same way. An algorithm that uses an iterator to traverse a collection is generic, because the same technique can be applied to any type of collection. Iterators can seem rather strange to someone who is encountering generic programming for the first time, but you should understand that they solve a difficult problem in an elegant way.

The interface Collection<T> defines a method that can be used to obtain an iterator for any collection. If coll is a collection, then coll.iterator() returns an iterator that can be used to traverse the collection. You should think of the iterator as a kind of generalized pointer that starts at the beginning of the collection and can move along the collection from one item to the next. Iterators are defined by a parameterized interface named Iterator<T>. If coll implements the interface Collection<T> for some specific type T, then coll.iterator() returns an iterator of type Iterator<T> , with the same type T as its type parameter. The interface Iterator<T> defines just three methods. If iter refers to an object that implements Iterator<T>, then we have:

- iter.next()–returns the next item, and advances the iterator. The return value is of type T. This method lets you look at one of the items in the collection. Note that there is no way to look at an item without advancing the iterator past that item. If this method is called when no items remain, it will throw a NoSuchElementException.
- iter.hasNext()–returns a boolean value telling you whether there are more items to be processed. In general, you should test this before calling iter.next().
- iter.remove()–if you call this after calling iter.next(), it will remove the item that you just saw from the collection. Note that this method has **no parameter** . It removes the item that was most recently returned by iter.next(). This might produce an UnsupportedOperationException, if the collection does not support removal of items.

Using iterators, we can write code for printing all the items in **any** collection. Suppose, for example, that coll is of type Collection<String>. In that case, the value returned by coll.iterator() is of type Iterator<String>, and we can say:

```
Iterator<String> iter;          // Declare the iterater variable.
iter = coll.iterator();         // Get an iterator for the collection.
while ( iter.hasNext() ) {
   String item = iter.next();   // Get the next item.
   System.out.println(item);
}
```

The same general form will work for other types of processing. For example, the following code will remove all **null** values from any collection of type Collection<JButton> (as long as that collection supports removal of values):

```
Iterator<JButton> iter = coll.iterator():
while ( iter.hasNext() ) {
    JButton item = iter.next();
    if (item == null)
       iter.remove();
}
```

(Note, by the way, that when Collection<T>, Iterator<T>, or any other parameterized type is used in actual code, they are always used with actual types such as String or JButton in place of the “formal type parameter” T. An iterator of type Iterator<String> is used to iterate through a collection of Strings; an iterator of type Iterator<JButton> is used to iterate through a collection of JButtons; and so on.)

An iterator is often used to apply the same operation to all the elements in a collection. In many cases, it’s possible to avoid the use of iterators for this purpose by using a **for**−each loop. A **for**−each loop can also be used to iterate through any collection. For a collection coll of type Collection<T>, a **for**−each loop takes the form:

```
for ( T x : coll ) { // "for each object x, of type T, in coll"
   // process x
}
```

Here, x is the loop control variable. Each object in coll will be assigned to x in turn, and the body of the loop will be executed for each object. Since objects in coll are of type T, x is declared to be of type T. For example, if namelist is of type Collection<String>, we can print out all the names in the collection with:

```
for ( String name : namelist ) {
   System.out.println( name );
}
```

This for-each loop could, of course, be written as a while loop using an iterator, but the for-each loop is much easier to follow.

8.6 Equality and Comparison

THERE ARE SEVERAL METHODS in the collection interface that test objects for equality. For example, the methods coll.contains(object) and coll.remove(object) look for an item in the collection that is equal to object. However, equality is not such a simple matter. The obvious technique for testing equality–using the == operator– does not usually give a reasonable answer when applied to objects. The == operator tests whether two objects are identical in the sense that they share the same location in memory. Usually, however, we want to consider two objects to be equal if they represent the same value, which is a very different thing. Two values of type String should be considered equal if they contain the same sequence of characters. The question of whether those characters are stored in the same location in memory is irrelevant. Two values of type Date should be considered equal if they represent the same time.

The Object class defines the boolean-valued method equals(Object) for testing whether one object is equal to another. This method is used by many, but not by all, collection classes for deciding whether two objects are to be considered the same. In the Object class, obj1.equals(obj2) is defined to be the same as obj1 == obj2. However, for most sub-classes of Object, this definition is not reasonable, and it should be overridden. The String class, for example, overrides equals() so that for a String str, str.equals(obj) if obj is also a String and obj contains the same sequence of characters as str.

If you write your own class, you might want to define an equals() method in that class to get the correct behavior when objects are tested for equality. For example, a Card class that will work correctly when used in collections could be defined as shown below. Without the equals() method in this class, methods such as contains() and remove() in the interface Collection<Card> will not work as expected.

```
public class Card {  // Class to represent playing cards.

   int suit;  // Number from 0 to 3 that codes for the suit --
              // spades, diamonds, clubs or hearts.
   int value; // Number from 1 to 13 that represents the value.

   public boolean equals(Object obj) {
       try {
          Card other = (Card)obj;  // Type-cast obj to a Card.
          if (suit == other.suit && value == other.value) {
                // The other card has the same suit and value as
                // this card, so they should be considered equal.
             return true;
          }
          else
             return false;
       }
       catch (Exception e) {
              // This will catch the NullPointerException that occurs if obj
              // is null and the ClassCastException that occurs if obj is
              // not of type Card.  In these cases, obj is not equal to
              // this Card, so return false.
           return false;
       }
    }

    .
    . // other methods and constructors
    .
}
```

A similar concern arises when items in a collection are sorted. Sorting refers to arranging a sequence of items in ascending order, according to some criterion. The problem is that there is no natural notion of ascending order for arbitrary objects. Before objects can be sorted, some method must be defined for comparing them. Objects that are meant to be compared should implement the interface `java.lang.Comparable`. In fact, Comparable is defined as a parameterized interface, `Comparable<T>`, which represents the ability to be compared to an object of type T. The interface `Comparable<T>` defines one method: `public int compareTo( T obj )`.

The value returned by `obj1.compareTo(obj2)` should be negative if and only if `obj1` comes before `obj2`, when the objects are arranged in ascending order. It should be positive if and only if `obj1` comes after `obj2`. A return value of zero means that the objects are considered to be the same for the purposes of this comparison. This does not necessarily mean that the objects are equal in the sense that `obj1.equals(obj2)` is true. For example, if the objects are of type `Address`, representing mailing addresses, it might be useful to sort the objects by zip code. Two Addresses are considered the same for the purposes of the sort if they have the same zip code–but clearly that would not mean that they are the same address.

The `String` class implements the interface `Comparable<String>` and define `compareTo` in a reasonable way (and in this case, the return value of `compareTo` is zero if and only if the two strings that are being compared are equal). If you define your own class and want to be able to sort objects belonging to that class, you should do the same. For example:

```
/**
 * Represents a full name consisting of a first name and a last name.
 */
public class FullName implements Comparable<FullName> {

   private String firstName, lastName;  // Non-null first and last names.

   public FullName(String first, String last) {  // Constructor.
      if (first == null || last == null)
         throw new IllegalArgumentException("Names must be non-null.");
      firstName = first;
      lastName = last;
   }

   public boolean equals(Object obj) {
      try {
         FullName other = (FullName)obj;  // Type-cast obj to type FullName
         return firstName.equals(other.firstName)
                              && lastName.equals(other.lastName);
      }
      catch (Exception e) {
         return false;  // if obj is null or is not of type FirstName
      }
   }

   public int compareTo( FullName other ) {
      if ( lastName.compareTo(other.lastName) < 0 ) {
              // If lastName comes before the last name of
              // the other object, then this FullName comes
              // before the other FullName.  Return a negative
              // value to indicate this.
         return -1;
      }
      if ( lastName.compareTo(other.lastName) > 0 ) {
              // If lastName comes after the last name of
              // the other object, then this FullName comes
              // after the other FullName.  Return a positive
              // value to indicate this.
         return 1;
      }
      else {
              // Last names are the same, so base the comparison on
              // the first names, using compareTo from class String.
         return firstName.compareTo(other.firstName);
      }
   }

   .
   . // other methods
   .
}
```

(Its odd to declare the class as “classFullName **implements** Comparable<FullName>”, with “FullName” repeated as a type parameter in the name of the interface. However, it does make sense. It means that we are going to compare objects that belong to the class FullName to other objects **of the same type**. Even though this is the only reasonable thing to do, that fact is not obvious to the Java compiler – and the type parameter in `Comparable<FullName>` is there for the compiler.)

There is another way to allow for comparison of objects in Java, and that is to provide a separate object that is capable of making the comparison. The object must implement the interface `Comparator<T>`, where `T` is the type of the objects that are to be compared. The interface `Comparator<T>` defines the method: **`public int`** `compare( T obj1, T obj2 )`.

This method compares two objects of type T and returns a value that is negative, or positive, or zero, depending on whether `obj1` comes before `obj2`, or comes after `obj2`, or is considered to be the same as `obj2` for the purposes of this comparison. Comparators are useful for comparing objects that do not implement the `Comparable` interface and for defining several different orderings on the same collection of objects.

In the next two sections, we'll see how Comparable and Comparator are used in the context of collections and maps.

8.7 Generics and Wrapper Classes

AS NOTED ABOVE, JAVA'S GENERIC PROGRAMMING does not apply to the primitive types, since generic data structures can only hold objects, while values of primitive type are not objects. However, the "*wrapper classes*" make it possible to get around this restriction to a great extent.

Recall that each primitive type has an associated wrapper class: class `Integer` for type **`int`**, class `Boolean` for type **boolean**, class `Character` for type **char**, and so on.

An object of type `Integer` contains a value of type int. The object serves as a "wrapper" for the primitive type value, which allows it to be used in contexts where objects are required, such as in generic data structures. For example, a list of Integers can be stored in a variable of type `ArrayList<Integer>`, and interfaces such as `Collection<Integer>` and `Set<Integer>` are defined. Furthermore, class `Integer` defines `equals()`, `compareTo()`, and `toString()` methods that do what you would expect (that is, that compare and write out the corresponding primitive type values in the usual way). Similar remarks apply for all the wrapper classes.

Recall also that Java does automatic conversions between a primitive type and the corresponding wrapper type. (These conversions, are called *autoboxing* and *unboxing*) This means that once you have created a generic data structure to hold objects belonging to one of the wrapper classes, you can use the data structure pretty much as if it actually contained primitive type values. For example, if `numbers` is a variable of type `Collection<Integer>`, it is legal to call `numbers.add(17)` or `numbers.remove(42)`. You can't literally add the primitive type value 17 to numbers, but Java will automatically convert the 17 to the corresponding wrapper object, **new** `Integer(17)`, and the wrapper object will be added to the collection. (The creation of the object does add some time and memory overhead to the operation, and you should keep that in mind in situations where efficiency is important. An array of int is more efficient than an `ArrayList<Integer>`)

8.8 Lists

IN THE PREVIOUS SECTION, we looked at the general properties of collection classes in Java. In this section, we look at a few specific collection classes (lists in particular) and how to use them. A list consists of a sequence of items arranged in a linear order. A list has a definite order, but is not necessarily sorted into ascending order.

ArrayList and LinkedList

There are two obvious ways to represent a list: as a dynamic array and as a linked list. Both of these options are available in generic form as the collection classes `java.util.ArrayList` and `java.util.LinkedList`. These classes are part of the Java Collection Framework. Each implements the interface `List<T>`, and therefor the interface `Collection<T>`. An object of type `ArrayList<T>` represents an ordered sequence of objects of type T, stored in an array that will grow in size whenever necessary as new items are added. An object of type `LinkedList<T>` also represents an ordered sequence of objects of type T, but the objects are stored in nodes that are linked together with pointers.

Both list classes support the basic list operations that are defined in the interface `List<T>`, and an abstract data type is defined by its operations, not by its representation. So why two classes? Why not a single List class with a single representation? The problem is that there **is** no single representation of lists for which all list operations are efficient. For some operations, linked lists are more efficient than arrays. For others, arrays are more efficient. In a particular application of lists, it's likely that only a few operations will be used frequently. You want to choose the representation for which the frequently used operations will be as efficient as possible.

Broadly speaking, the `LinkedList` class is more efficient in applications where items will often be added or removed at the beginning of the list or in the middle of the list. In an array, these operations require moving a large number of items up or down one position in the array, to make a space for a new item or to fill in the hole left by the removal of an item.

On the other hand, the ArrayList class is more efficient when random access to items is required. Random access means accessing the k-th item in the list, for any integer k. Random access is used when you get or change the value stored at a specified position in the list. This is trivial for an array. But for a linked list it means starting at the beginning of the list and moving from node to node along the list for k steps.

Operations that can be done efficiently for both types of lists include sorting and adding an item at the end of the list.

All lists implement the methods from interface `Collection<T>` that were discussed in previously. These methods include `size()`, `isEmpty()`, `remove(Object)`, `add(T)`, and `clear()`. The `add(T)` method adds the object at the end of the list. The `remove(Object)` method involves first finding the object, which is not very efficient for any list since it involves going through the items in the list from beginning to end until the object is found. The interface `List<T>` adds some methods for accessing list items according to their numerical positions in the list. Suppose that list is an object of type `List<T>`. Then we have the methods:

- `list.get(index)`–returns the object of type T that is at position index in the list, where index is an integer. Items are numbered 0, 1, 2, ..., `list.size()`−1. The parameter must be in this range, or an `IndexOutOfBoundsException` is thrown.

- `list.set(index,obj)`–stores the object `obj` at position number index in the list, replacing the object that was there previously. The object `obj` must be of type T. This does not change the number of elements in the list or move any of the other elements.

- list.add(index,obj)–inserts an object obj into the list at position number index, where obj must be of type T. The number of items in the list increases by one, and items that come after position index move up one position to make room for the new item. The value of index must be in the range 0 to list.size(), inclusive. If index is equal to list.size(), then obj is added at the end of the list.

- list.remove(index)–removes the object at position number index, and returns that object as the return value of the method. Items after this position move up one space in the list to fill the hole, and the size of the list decreases by one. The value of index must be in the range 0 to list.size()−1.

- list.indexOf(obj)–returns an **int** that gives the position of obj in the list, if it occurs. If it does not occur, the return value is −1. The object obj can be of any type, not just of type T. If obj occurs more than once in the list, the index of the first occurrence is returned.

These methods are defined both in class ArrayList<T> and in class LinkedList<T>, although some of them–get and set–are only efficient for ArrayLists. The class LinkedList<T> adds a few additional methods, which are not defined for an ArrayList. If linkedlist is an object of type LinkedList<T>, then we have

- linkedlist.getFirst()–returns the object of type T that is the first item in the list. The list is not modified. If the list is empty when the method is called, an exception of type NoSuchElementException is thrown (the same is true for the next three methods as well).

- linkedlist.getLast()–returns the object of type T that is the last item in the list. The list is not modified.

- linkedlist.removeFirst()–removes the first item from the list, and returns that object of type T as its return value.

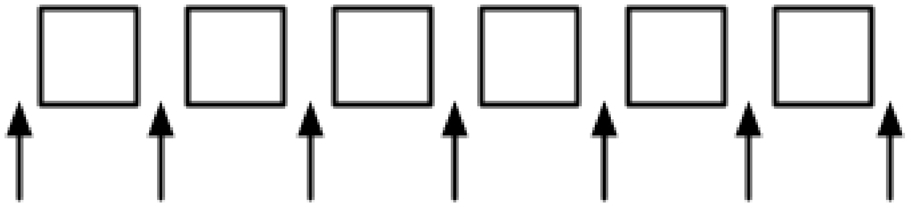
- linkedlist.removeLast()–removes the last item from the list, and returns that object of type T as its return value.

- linkedlist.addFirst(obj)–adds the obj, which must be of type T, to the beginning of the list.

- linkedlist.addLast(obj)–adds the object obj, which must be of type T, to the end of the list. (This is exactly the same as linkedlist.add(obj) and is apparently defined just to keep the naming consistent.)

If list is an object of type List<T>, then the method list.iterator(), defined in the interface Collection<T>, returns an Iterator that can be used to traverse the list from beginning to end. However, for Lists, there is a special type of Iterator, called a ListIterator, which offers additional capabilities. ListIterator<T> is an interface that extends the interface Iterator<T>. The method list.listIterator() returns an object of type ListIterator<T>.

A ListIterator has the usual Iterator methods, hasNext(), next(), and remove(), but it also has methods hasPrevious(), previous(), and add(obj) that make it possible to move backwards in the list and to add an item at the current position of the iterator. To understand how these work, its best to think of an iterator as pointing to a position **between** two list elements, or at the beginning or end of the list. In this diagram, the items in a list are represented by squares, and arrows indicate the possible positions of an iterator:



If iter is of type ListIterator<T>, then iter.next() moves the iterator one space to the right along the list and returns the item that the iterator passes as it moves. The method iter.previous() moves the iterator one space to the left along the list and returns the item that it passes. The method iter.remove() removes an item from the list; the item that is removed is the item that the iterator passed most recently in a call to either iter.next() or iter.previous(). There is also a method iter.add(obj) that adds the specified object to the list at the current position of the iterator (where obj must be of type T). This can be between two existing items or at the beginning of the list or at the end of the list.

As an example of using a ListIterator, suppose that we want to maintain a list of items that is always sorted into increasing order. When adding an item to the list, we can use a ListIterator to find the position in the list where the item should be added. Once the position has been found, we use the same list iterator to place the item in that position. The idea is to start at the beginning of the list and to move the iterator forward past all the items that are smaller than the item that is being inserted. At that point, the iterator's add() method can be used to insert the item. To be more definite, suppose that stringList is a variable of type List<String>. Assume that that the strings that are already in the list are stored in ascending order and that newItem is a string that we would like to insert into the list. The following code will place newItem in the list in its correct position, so that the modified list is still in ascending order:

```
ListIterator<String> iter = stringList.listIterator();
// Move the iterator so that it points to the position where
// newItem should be inserted into the list.  If newItem is
// bigger than all the items in the list, then the while loop
// will end when iter.hasNext() becomes false, that is, when
// the iterator has reached the end of the list.
while (iter.hasNext()) {
   String item = iter.next();
   if (newItem.compareTo(item) <= 0) {
         // newItem should come BEFORE item in the list.
         // Move the iterator back one space so that
         // it points to the correct insertion point,
         // and end the loop.
      iter.previous();
      break;
   }
}
iter.add(newItem);
```

Here, stringList may be of type ArrayList<String> or of type LinkedList<String>. The algorithm that is used to insert newItem into the list will be about equally efficient for both types of lists, and it will even work for other classes that implement the interface List<String>. You would probably find it easier to design an insertion algorithm that uses array-like indexing with the methods get(index) and add(index,obj). However, that algorithm would be inefficient for LinkedLists because random access is so inefficient for linked lists. (By the way, the insertion algorithm works when the list is empty. It might be useful for you to think about why this is true.)

Sorting

Sorting a list is a fairly common operation, and there should really be a sorting method in the List interface. There is not, presumably because it only makes sense to sort lists of certain types of objects, but methods for sorting lists are available as static methods in the class java.util.Collections. This class contains a variety of static utility methods for working with collections. The methods are generic; that is, they will work for collections of objects of various types. Suppose that list is of type List<T>. The command Collections.sort(list); can be used to sort the list into ascending order. The items in the list should implement the interface Comparable<T>. The method Collections.sort() will work, for example, for lists of String and for lists of any of the wrapper classes such as Integer and Double. There is also a sorting method that takes a Comparator as its second argument: Collections.sort(list,comparator);.

In this method, the comparator will be used to compare the items in the list. As mentioned in the previous section, a Comparator is an object that defines a compare() method that can be used to compare two objects.

The sorting method that is used by Collections.sort() is the so-called "merge sort" algorithm.

The Collections class has at least two other useful methods for modifying lists. Collections.shuffle(list) will rearrange the elements of the list into a random order. Collections.reverse(list) will reverse the order of the elements, so that the last element is moved to the beginning of the list, the next-to-last element to the second position, and so on.

Since an efficient sorting method is provided for Lists, there is no need to write one yourself. You might be wondering whether there is an equally convenient method for standard arrays. The answer is yes. Array-sorting methods are available as static methods in the class java.util.Arrays. The statement Arrays.sort(A); will sort an array, A, provided either that the base type of A is one of the primitive types (except **boolean**) or that A is an array of Objects that implement the Comparable interface. You can also sort part of an array. This is important since arrays are often only "partially filled." The command: Arrays.sort(A,fromIndex,toIndex); sorts the elements A[fromIndex], A[fromIndex+1], ..., A[toIndex−1] into ascending order. You can use Arrays.sort(A,0,N−1) to sort a partially filled array which has elements in the first N positions.

Java does not support generic programming for primitive types. In order to implement the command Arrays.sort(A), the Arrays class contains eight methods: one method for arrays of Objects and one method for each of the primitive types **byte**, **short**, **int**, **long**, **float**, **double**, and **char**.

Chapter 9

Correctness and Robustness

Contents

A PROGRAM IS CORRECT if it accomplishes the task that it was designed to perform. It is robust if it can handle illegal inputs and other unexpected situations in a reasonable way. For example, consider a program that is designed to read some numbers from the user and then print the same numbers in sorted order. The program is correct if it works for any set of input numbers. It is robust if it can also deal with non-numeric input by, for example, printing an error message and ignoring the bad input. A non-robust program might crash or give nonsensical output in the same circumstance.

Every program should be correct. (A sorting program that doesn't sort correctly is pretty useless.) It's not the case that every program needs to be completely robust. It depends on who will use it and how it will be used. For example, a small utility program that you write for your own use doesn't have to be particularly robust.

The question of correctness is actually more subtle than it might appear. A programmer works from a specification of what the program is supposed to do. The programmer's work is correct if the program meets its specification. But does that mean that the program itself is correct? What if the specification is incorrect or incomplete? A correct program should be a correct implementation of a complete and correct specification. The question is whether the specification correctly expresses the intention and desires of the people for whom the program is being written. This is a question that lies largely outside the domain of computer science.

9.1 Introduction

9.1.1 Horror Stories

MOST COMPUTER USERS HAVE PERSONAL EXPERIENCE with programs that don't work or that crash. In many cases, such problems are just annoyances, but even on a personal computer there can be more serious consequences, such as lost work or lost money. When computers are given more important tasks, the consequences of failure can be proportionately more serious.

Just a few years ago, the failure of two multi-million space missions to Mars was prominent in the news. Both failures were probably due to software problems, but in both cases the problem was not with an incorrect program as such. In September 1999, the Mars Climate Orbiter burned up in the Martian atmosphere because data that was expressed in English units of measurement (such as feet and pounds) was entered into a computer program that was designed to use metric units (such as centimeters and grams). A few months later, the Mars Polar Lander probably crashed because its software turned off its landing engines too soon. The program was supposed to detect the bump when the spacecraft landed and turn off the engines then. It has been determined that deployment of the landing gear might have jarred the spacecraft enough to activate the program, causing it to turn off the engines when the spacecraft was still in the air. The unpowered spacecraft would then have fallen to the Martian surface. A more robust system would have checked the altitude before turning off the engines!

There are many equally dramatic stories of problems caused by incorrect or poorly written software. Let's look at a few incidents recounted in the book *Computer Ethics* by Tom Forester and Perry Morrison. (This book covers various ethical issues in computing. It, or something like it, is essential reading for any student of computer science.)

In 1985 and 1986, one person was killed and several were injured by excess radiation, while undergoing radiation treatments by a mis-programmed computerized radiation machine. In another case, over a ten-year period ending in 1992, almost 1,000 cancer patients received radiation dosages that were 30% less than prescribed because of a programming error.

In 1985, a computer at the Bank of New York started destroying records of ongoing security transactions because of an error in a program. It took less than 24 hours to fix the program, but by that time, the bank was out $5,000,000 in overnight interest payments on funds that it had to borrow to cover the problem.

The programming of the inertial guidance system of the F-16 fighter plane would have turned the plane upside-down when it crossed the equator, if the problem had not been discovered in simulation. The Mariner 18 space probe was lost because of an error in one line of a program. The Gemini V space capsule missed its scheduled landing target by a hundred miles, because a programmer forgot to take into account the rotation of the Earth.

In 1990, AT&T's long-distance telephone service was disrupted throughout the United States when a newly loaded computer program proved to contain a bug.

These are just a few examples. Software problems are all too common. As programmers, we need to understand why that is true and what can be done about it.

9.1.2 Java to the Rescue

Part of the problem, according to the inventors of Java, can be traced to programming languages themselves. Java was designed to provide some protection against certain types of errors. How can a language feature help prevent errors? Let's look at a few examples.

Early programming languages did not require variables to be declared. In such languages, when a variable name is used in a program, the variable is created automatically. You might consider this more convenient than having to declare every variable explicitly. But there is an unfortunate consequence: An inadvertent spelling error might introduce an extra variable that you had no intention of creating. This type of error was responsible, according to one famous story, for yet another lost spacecraft. In the FORTRAN programming language, the command "DO 20 I = 1,5" is the first statement of a counting loop. Now, spaces are insignificant in FORTRAN, so this is equivalent to "DO20I=1,5". On the other hand, the command "DO20I=1.5", with a period instead of a comma, is an assignment statement that assigns the value 1.5 to the variable DO20I. Supposedly, the inadvertent substitution of a period for a comma in a statement of this type caused a rocket to blow up on take-off. Because FORTRAN doesn't require variables to be declared, the compiler would be happy to accept the statement "DO20I=1.5." It would just create a new variable named DO20I. If FORTRAN required variables to be declared, the compiler would have complained that the variable DO20I was undeclared.

While most programming languages today do require variables to be declared, there are other features in common programming languages that can cause problems. Java has eliminated some of these features. Some people complain that this makes Java less efficient and less powerful. While there is some justice in this criticism, the increase in security and robustness is probably worth the cost in most circumstances. The best defense against some types of errors is to design a programming language in which the errors are impossible. In other cases, where the error can't be completely eliminated, the language can be designed so that when the error does occur, it will automatically be detected. This will at least prevent the error from causing further harm, and it will alert the programmer that there is a bug that needs fixing. Let's look at a few cases where the designers of Java have taken these approaches.

An array is created with a certain number of locations, numbered from zero up to some specified maximum index. It is an error to try to use an array location that is outside of the specified range. In Java, any attempt to do so is detected automatically by the system. In some other languages, such as C and C++, it's up to the programmer to make sure that the index is within the legal range. Suppose that an array, A, has three locations, A[0], A[1], and A[2]. Then A[3], A[4], and so on refer to memory locations beyond the end of the array. In Java, an attempt to store data in A[3] will be detected. The program will be terminated (unless the error is "caught". In C or C++, the computer will just go ahead and store the data in memory that is not part of the array. Since there is no telling what that memory location is being used for, the result will be unpredictable. The consequences could be much more serious than a terminated program. (See, for example, the discussion of buffer overflow errors later in this section.)

Pointers are a notorious source of programming errors. In Java, a variable of object type holds either a pointer to an object or the special value null. Any attempt to use a null value as if it were a pointer to an actual object will be detected by the system. In some other languages, again, it's up to the programmer to avoid such null pointer errors. In my old Macintosh computer, a null pointer was actually implemented as if it were a pointer to memory location zero. A program could use a null pointer to change values stored in memory near location zero. Unfortunately, the Macintosh stored important system data in those locations. Changing that data could cause the whole system to crash, a consequence more severe than a single failed program.

Another type of pointer error occurs when a pointer value is pointing to an object of the wrong type or to a segment of memory that does not even hold a valid object at all. These types of errors are impossible in Java, which does not allow programmers to manipulate pointers directly. In other languages, it is possible to set a pointer to point, essentially, to any location in memory. If this is done incorrectly, then using the pointer can have unpredictable results.

Another type of error that cannot occur in Java is a memory leak. In Java, once there are no longer any pointers that refer to an object, that object is "garbage collected" so that the memory that it occupied can be reused. In other languages, it is the programmer's responsibility to return unused memory to the system. If the programmer fails to do this, unused memory can build up, leaving less memory for programs and data. There is a story that many common programs for older Windows computers had so many memory leaks that the computer would run out of memory after a few days of use and would have to be restarted.

Many programs have been found to suffer from buffer overflow errors. Buffer overflow errors often make the news because they are responsible for many network security problems. When one computer receives data from another computer over a network, that data is stored in a buffer. The buffer is just a segment of memory that has been allocated by a program to hold data that it expects to receive. A buffer overflow occurs when more data is received than will fit in the buffer. The question is, what happens then? If the error is detected by the program or by the networking software, then the only thing that has happened is a failed network data transmission. The real problem occurs when the software does not properly detect buffer overflows. In that case, the software continues to store data in memory even after the buffer is filled, and the extra data goes into some part of memory that was not allocated by the program as part of the buffer. That memory might be in use for some other purpose. It might contain important data. It might even contain part of the program itself. This is where the real security issues come in. Suppose that a buffer overflow causes part of a program to be replaced with extra data received over a network. When the computer goes to execute the part of the program that was replaced, it's actually executing data that was received from another computer. That data could be anything. It could be a program that crashes the computer or takes it over. A malicious programmer who finds a convenient buffer overflow error in networking software can try to exploit that error to trick other computers into executing his programs.

For software written completely in Java, buffer overflow errors are impossible. The language simply does not provide any way to store data into memory that has not been properly allocated. To do that, you would need a pointer that points to unallocated memory or you would have to refer to an array location that lies outside the range allocated for the array. As explained above, neither of these is possible in Java. (However, there could conceivably still be errors in Java's standard classes, since some of the methods in these classes are actually written in the C programming language rather than in Java.)

It's clear that language design can help prevent errors or detect them when they occur. Doing so involves restricting what a programmer is allowed to do. Or it requires tests, such as checking whether a pointer is null, that take some extra processing time. Some programmers feel that the sacrifice of power and efficiency is too high a price to pay for the extra security. In some applications, this is true. However, there are many situations where safety and security are primary considerations. Java is designed for such situations.

9.1.3 Problems Remain in Java

There is one area where the designers of Java chose not to detect errors automatically: numerical computations. In Java, a value of type int is represented as a 32-bit binary number. With 32 bits, it's possible to represent a little over four billion different values. The values of type int range from -2147483648 to 2147483647. What happens when the result of a computation lies outside this range? For example, what is $2147483647 + 1$? And what is $2000000000 * 2$? The mathematically correct result in each case cannot be represented as a value of type int. These are examples of integer overflow. In most cases, integer overflow should be considered an error. However, Java does not automatically detect such errors. For example, it will compute the value of $2147483647 + 1$ to be the negative number, -2147483648. (What happens is that any extra bits beyond the 32-nd bit in the correct answer are discarded. Values greater than 2147483647 will "wrap around" to negative values. Mathematically speaking, the result is always "correct modulo 232".)

For example, consider the $3N + 1$ program. Starting from a positive integer N, the program computes a certain sequence of integers:

```
while ( N != 1 ) {
   if ( N % 2 == 0 )  // If N is even...
      N = N / 2;
   else
      N = 3 * N + 1;
   System.out.println(N);
}
```

But there is a problem here: If N is too large, then the value of $3 * N + 1$ will not be mathematically correct because of integer overflow. The problem arises whenever $3 * N + 1 > 2147483647$, that is when $N > 2147483646/3$. For a completely correct program, we should check for this possibility **before** computing $3 * N + 1$:

```
while ( N != 1 ) {
   if ( N % 2 == 0 )  // If N is even...
      N = N / 2;
   else {
      if (N > 2147483646/3) {
        System.out.println("Sorry, value of N has become too large!");
        break;
      }
      N = 3 * N + 1;
   }
   System.out.println(N); }
```

The problem here is not that the original algorithm for computing $3N + 1$ sequences was wrong. The problem is that it just can't be correctly implemented using 32-bit integers. Many programs ignore this type of problem. But integer overflow errors have been responsible for their share of serious computer failures, and a completely robust program should take the possibility of integer overflow into account. (The infamous "Y2K" bug was, in fact, just this sort of error.)

For numbers of type double, there are even more problems. There are still overflow errors, which occur when the result of a computation is outside the range of values that can be represented as a value of type double. This range extends up to about 1.7×10^{308}. Numbers beyond this range do not "wrap around" to negative values. Instead, they are represented by special values that have no real numerical equivalent. The special values `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY` represent numbers outside the range of legal values. For example, 20×10^{308} is computed to be `Double.POSITIVE_INFINITY`. Another special value of type double, `Double.NaN`, represents an illegal or undefined result. ("`NaN`" stands for "Not a Number".) For example, the result of dividing by zero or taking the square root of a negative number is `Double.NaN`. You can test whether a number x is this special non-a-number value by calling the boolean-valued method `Double.isNaN(x)`.

For real numbers, there is the added complication that most real numbers can only be represented approximately on a computer. A real number can have an infinite number of digits after the decimal point. A value of type double is only accurate to about 15 digits. The real number $1/3$, for example, is the repeating decimal $0.333333333333...$, and there is no way to represent it exactly using a finite number of digits. Computations with real numbers generally involve a loss of accuracy. In fact, if care is not exercised, the result of a large number of such computations might be completely wrong! There is a whole field of computer science, known as numerical analysis, which is devoted to studying algorithms that manipulate real numbers.

So you see that not all possible errors are avoided or detected automatically in Java. Furthermore, even when an error is detected automatically, the system's default response is to report the error and terminate the program. This is hardly robust behavior! So, a Java programmer still needs to learn techniques for avoiding and dealing with errors. These are the main topics of the rest of this chapter.

9.2 Writing Correct Programs

CORRECT PROGRAMS DON'T JUST HAPPEN. It takes planning and attention to detail to avoid errors in programs. There are some techniques that programmers can use to increase the likelihood that their programs are correct.

9.2.1 Provably Correct Programs

In some cases, it is possible to **prove** that a program is correct. That is, it is possible to demonstrate mathematically that the sequence of computations represented by the program will always produce the correct result. Rigorous proof is difficult enough that in practice it can only be applied to fairly small programs. Furthermore, it depends on the fact that the "correct result" has been specified correctly and completely. As I've already pointed out, a program that correctly meets its specification is not useful if its specification was wrong. Nevertheless, even in everyday programming, we can apply the ideas and techniques that are used in proving that programs are correct.

The fundamental ideas are *process* and *state*. A state consists of all the information relevant to the execution of a program at a given moment during its execution. The state includes, for example, the values of all the variables in the program, the output that has been produced, any input that is waiting to be read, and a record of the position in the program where the computer is working. A process is the sequence of states that the computer goes through as it executes the program. From this point of view, the meaning of a statement in a program can be expressed in terms of the effect that the execution of that statement has on the computer's state. As a simple example, the meaning of the assignment statement "$x = 7$;" is that after this statement is executed, the value of the variable x will be 7. We can be absolutely sure of this fact, so it is something upon which we can build part of a mathematical proof.

In fact, it is often possible to look at a program and deduce that some fact must be true at a given point during the execution of a program. For example, consider the do loop:

```
do {
   Scanner keyboard = new Scanner(System.in);
   System.out.prinln("Enter a positive integer: ");
   N = keyboard.nextInt();
} while (N <= 0);
```

After this loop ends, we can be absolutely sure that the value of the variable N is greater than zero. The loop cannot end until this condition is satisfied. This fact is part of the meaning of the while loop. More generally, if a while loop uses the test "**while** (condition)", then after the loop ends, we can be sure that the condition is false. We can then use this fact to draw further deductions about what happens as the execution of the program continues. (With a loop, by the way, we also have to worry about the question of whether the loop will ever end. This is something that has to be verified separately.)

A fact that can be proven to be true after a given program segment has been executed is called a postcondition of that program segment. Postconditions are known facts upon which we can build further deductions about the behavior of the program. A postcondition of a program as a whole is simply a fact that can be proven to be true after the program has finished executing. A program can be proven to be correct by showing that the postconditions of the program meet the program's specification.

Consider the following program segment, where all the variables are of type double:

```
disc = B*B - 4*A*C;
x = (-B + Math.sqrt(disc)) / (2*A);
```

The quadratic formula (from high-school mathematics) assures us that the value assigned to x is a solution of the equation $Ax^2+Bx+C = 0$, provided that the value of disc is greater than or equal to zero and the value of A is not zero. **If** we can assume or guarantee that $B * B - 4 * A * C \ \ > = \ \ 0$ and that $A! = 0$, then the fact that x is a solution of the equation becomes a postcondition of the program segment. We say that the condition, $B * B - 4 * A * C \ \ > = \ \ 0$ is a precondition of the program segment. The condition that $A \ ! = \ \ 0$ is another precondition. A precondition is defined to be condition that must be true at a given point in the execution of a program in order for the program to continue correctly. A precondition is something that you want to be true. It's something that you have to check or force to be true, if you want your program to be correct.

We've encountered preconditions and postconditions once before. That section introduced preconditions and postconditions as a way of specifying the contract of a method. As the terms are being used here, a precondition of a method is just a precondition of the code that makes up the definition of the method, and the postcondition of a method is a postcondition of the same code. In this section, we have generalized these terms to make them more useful in talking about program correctness.

Let's see how this works by considering a longer program segment:

```
do {
   Scanner keyboard = new Scanner(System.in);
   System.out.println("Enter A, B, and C.  B*B-4*A*C must be >= 0.");
   System.out.print("A = ");
   A = keyboard.nextDouble();
   System.out.print("B = ");
   B = keyboard.nextDouble();
   System.out.print("C = ");
   C = keyboard.nextDouble();
   if (A == 0 || B*B - 4*A*C < 0)
      System.out.println("Your input is illegal.  Try again.");
} while (A == 0 || B*B - 4*A*C < 0);

disc = B*B - 4*A*C;
x = (-B + Math.sqrt(disc)) / (2*A);
```

After the loop ends, we can be sure that $B * B - 4 * A * C\ \ >=\ 0$ and that $A\,! =\ 0$. The preconditions for the last two lines are fulfilled, so the postcondition that x is a solution of the equation $A * x2 + B * x + C = 0$ is also valid. This program segment correctly and provably computes a solution to the equation. (Actually, because of problems with representing numbers on computers, this is not 100% true. The **algorithm** is correct, but the **program** is not a perfect implementation of the algorithm.

Here is another variation, in which the precondition is checked by an if statement. In the first part of the if statement, where a solution is computed and printed, we know that the preconditions are fulfilled. In the other parts, we know that one of the preconditions fails to hold. In any case, the program is correct.

```
Scanner keyboard = new Scanner(System.in);
System.out.println("Enter your values for A, B, and C.");
System.out.print("A = ");
A = keyboard.nextDouble();
System.out.print("B = ");
B = keyboard.nextDouble();
System.out.print("C = ");
C = keyboard.nextDouble();

if (A != 0 && B*B - 4*A*C >= 0) {
   disc = B*B - 4*A*C;
   x = (-B + Math.sqrt(disc)) / (2*A);
   System.out.println("A solution of A*X*X + B*X + C = 0 is " + x);
}
else if (A == 0) {
   System.out.println("The value of A cannot be zero.");
}
else {
   System.out.println("Since B*B - 4*A*C is less than zero, the");
   System.out.println("equation A*X*X + B*X + C = 0 has no solution.");
}
```

Whenever you write a program, it's a good idea to watch out for preconditions and think about how your program handles them. Often, a precondition can offer a clue about how to write the program.

For example, every array reference, such as A[i], has a precondition: The index must be within the range of legal indices for the array. For A[i], the precondition is that $0 \ <= \ i \ < \ A$.length. The computer will check this condition when it evaluates A[i], and if the condition is not satisfied, the program will be terminated. In order to avoid this, you need to make sure that the index has a legal value. (There is actually another precondition, namely that A is not null, but let's leave that aside for the moment.) Consider the following code, which searches for the number x in the array A and sets the value of i to be the index of the array element that contains x:

```
i = 0;
while (A[i] != x) {
   i++;
}
```

As this program segment stands, it has a precondition, namely that x is actually in the array. If this precondition is satisfied, then the loop will end when A[i] == x. That is, the value of i when the loop ends will be the position of x in the array. However, if x is not in the array, then the value of i will just keep increasing until it is equal to A.length. At that time, the reference to A[i] is illegal and the program will be terminated. To avoid this, we can add a test to make sure that the precondition for referring to A[i] is satisfied:

```
i = 0;
while (i < A.length && A[i] != x) {
   i++;
}
```

Now, the loop will definitely end. After it ends, i will satisfy **either** i == A.length or A[i] == x. An **if** statement can be used after the loop to test which of these conditions caused the loop to end:

```
i = 0;
while (i < A.length && A[i] != x) {
   i++;
}

if (i == A.length)
   System.out.println("x is not in the array");
else
   System.out.println("x is in position " + i);
```

9.2.2 Robust Handling of Input

One place where correctness and robustness are important–and especially difficult– is in the processing of input data, whether that data is typed in by the user, read from a file, or received over a network.

Sometimes, it's useful to be able to look ahead at what's coming up in the input without actually reading it. For example, a program might need to know whether the next item in the input is a number or a word. For this purpose, the Scanner class has various hasNext methods. These includes hasNextBoolean(); hasNextInteger(); hasNextLine() and hasNextDouble(). For example the hasNextInteger() method returns true if the input's next token is an integer. Thus, you can check if the expected input is available before actually reading it.

9.3 Exceptions and try..catch

GETTING A PROGRAM TO WORK under ideal circumstances is usually a lot easier than making the program robust. A robust program can survive unusual or "exceptional" circumstances without crashing. One approach to writing robust programs is to anticipate the problems that might arise and to include tests in the program for each possible problem. For example, a program will crash if it tries to use an array element A[i], when i is not within the declared range of indices for the array A. A robust program must anticipate the possibility of a bad index and guard against it. One way to do this is to write the program in a way that ensures that the index is in the legal range. Another way is to test whether the index value is legal before using it in the array. This could be done with an if statement:

```
if (i < 0 || i >= A.length) {
   ...  // Do something to handle the out-of-range index, i
}
else {
   ...  // Process the array element, A[i]
}
```

There are some problems with this approach. It is difficult and sometimes impossible to anticipate all the possible things that might go wrong. It's not always clear what to do when an error is detected. Furthermore, trying to anticipate all the possible problems can turn what would otherwise be a straightforward program into a messy tangle of if statements.

9.3.1 Exceptions and Exception Classes

We have already seen that Java (like its cousin, C++) provides a neater, more structured alternative method for dealing with errors that can occur while a program is running. The method is referred to as exception handling. The word "exception" is meant to be more general than "error." It includes any circumstance that arises as the program is executed which is meant to be treated as an exception to the normal flow of control of the program. An exception might be an error, or it might just be a special case that you would rather not have clutter up your elegant algorithm.

When an exception occurs during the execution of a program, we say that the exception is thrown. When this happens, the normal flow of the program is thrown off-track, and the program is in danger of crashing. However, the crash can be avoided if the exception is caught and handled in some way. An exception can be thrown in one part of a program and caught in a different part. An exception that is not caught will generally cause the program to crash.

By the way, since Java programs are executed by a Java interpreter, having a program crash simply means that it terminates abnormally and prematurely. It doesn't mean that the Java interpreter will crash. In effect, the interpreter catches any exceptions that are not caught by the program. The interpreter responds by terminating the program. In many other programming languages, a crashed program will sometimes crash the entire system and freeze the computer until it is restarted. With

Java, such system crashes should be impossible – which means that when they happen, you have the satisfaction of blaming the system rather than your own program.

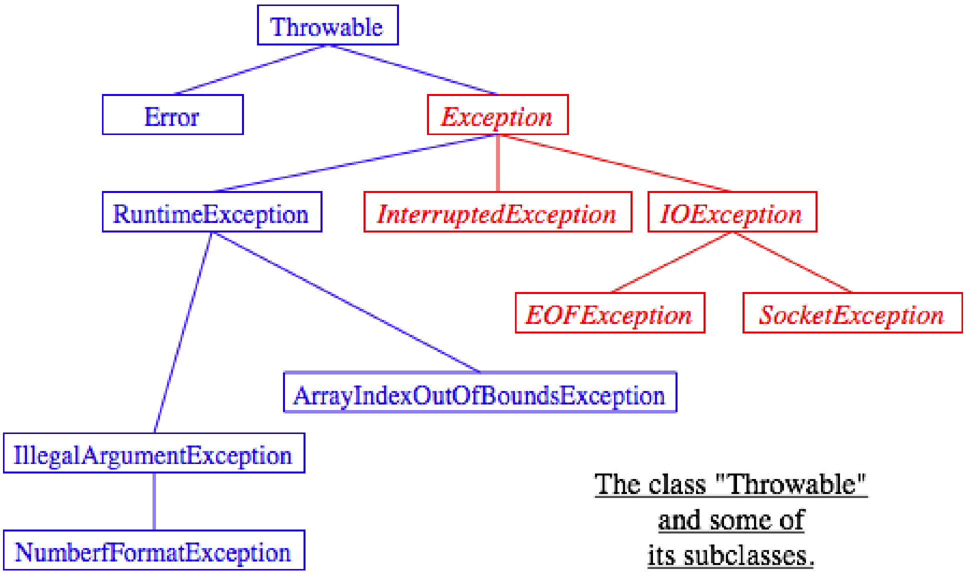
When an exception occurs, the thing that is actually "thrown" is an object. This object can carry information (in its instance variables) from the point where the exception occurs to the point where it is caught and handled. This information always includes the method call stack, which is a list of the methods that were being executed when the exception was thrown. (Since one method can call another, several methods can be active at the same time.) Typically, an exception object also includes an error message describing what happened to cause the exception, and it can contain other data as well. All exception objects must belong to a subclass of the standard class `java.lang.Throwable`. In general, each different type of exception is represented by its own subclass of `Throwable`, and these subclasses are arranged in a fairly complex class hierarchy that shows the relationship among various types of exceptions. `Throwable` has two direct subclasses, `Error` and `Exception`. These two subclasses in turn have many other predefined subclasses. In addition, a programmer can create new exception classes to represent new types of exceptions.

Most of the subclasses of the class `Error` represent serious errors within the Java virtual machine that should ordinarily cause program termination because there is no reasonable way to handle them. In general, you should not try to catch and handle such errors. An example is a `ClassFormatError`, which occurs when the Java virtual machine finds some kind of illegal data in a file that is supposed to contain a compiled Java class. If that class was being loaded as part of the program, then there is really no way for the program to proceed.

On the other hand, subclasses of the class `Exception` represent exceptions that are meant to be caught. In many cases, these are exceptions that might naturally be called "errors," but they are errors in the program or in input data that a programmer can anticipate and possibly respond to in some reasonable way. (However, you should avoid the temptation of saying, "Well, I'll just put a thing here to catch all the errors that might occur, so my program won't crash." If you don't have a reasonable way to respond to the error, it's best just to let the program crash, because trying to go on will probably only lead to worse things down the road – in the worst case, a program that gives an incorrect answer without giving you any indication that the answer might be wrong!)

The class `Exception` has its own subclass, `RuntimeException`. This class groups together many common exceptions, including all those that have been covered in previous sections. For example, `IllegalArgumentException` and `NullPointerException` are subclasses of `RuntimeException`. A `RuntimeException` generally indicates a bug in the program, which the programmer should fix. `RuntimeExceptions` and `Errors` share the property that a program can simply ignore the possibility that they might occur. ("Ignoring" here means that you are content to let your program crash if the exception occurs.) For example, a program does this every time it uses an array reference like `A[i]` without making arrangements to catch a possible `ArrayIndexOutOfBoundsException`. For all other exception classes besides `Error`, `RuntimeException`, and their subclasses, exception-handling is "mandatory" in a sense that I'll discuss below.

The following diagram is a class hierarchy showing the class `Throwable` and just a few of its subclasses. Classes that require mandatory exception-handling are shown in red:



The class "Throwable" and some of its subclasses.

The class Throwable includes several instance methods that can be used with any exception object. If e is of type Throwable (or one of its subclasses), then e.getMessage() is a method that returns a String that describes the exception. The method e.toString(), which is used by the system whenever it needs a string representation of the object, returns a String that contains the name of the class to which the exception belongs as well as the same string that would be returned by e.getMessage(). And e.printStackTrace() writes a stack trace to standard output that tells which methods were active when the exception occurred. A stack trace can be very useful when you are trying to determine the cause of the problem. (Note that if an exception is **not** caught by the program, then the system automatically prints the stack trace to standard output.)

9.3.2 The try Statement

To catch exceptions in a Java program, you need a try statement. The try statements that we have used so far had a syntax similar to the following example:

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( ArrayIndexOutOfBoundsException e ) {
   System.out.println("M is the wrong size to have a determinant.");
   e.printStackTrace();
}
```

Here, the computer tries to execute the block of statements following the word “try”. If no exception occurs during the execution of this block, then the “catch” part of the statement is simply ignored. However, if an exception of type ArrayIndexOutOfBoundsException occurs, then the computer jumps immediately to the catch clause of the try statement. This block of statements is said to be an exception handler for ArrayIndexOutOfBoundsException. By handling the exception in this way, you prevent it from crashing the program. Before the body of the catch clause is executed, the object that represents the exception is assigned to the variable e, which is used in this example to print a stack trace.

However, the full syntax of the try statement allows more than one catch clause. This makes it possible to catch several different types of exceptions with one try statement. In the example above, in addition to the possibility of an ArrayIndexOutOfBoundsException, there is a possible NullPointerException which will occur if the value of M is null. We can handle both exceptions by adding a second catch clause to the try statement:

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( ArrayIndexOutOfBoundsException e ) {
   System.out.println("M is the wrong size to have a determinant.");
}
catch ( NullPointerException e ) {
   System.out.print("Programming error!  M doesn't exist." + );
}
```

Here, the computer tries to execute the statements in the try clause. If no error occurs, both of the catch clauses are skipped. If an ArrayIndexOutOfBoundsException occurs, the computer executes the body of the first catch clause and skips the second one. If a NullPointerException occurs, it jumps to the second catch clause and executes that.

Note that both ArrayIndexOutOfBoundsException and NullPointerException are subclasses of RuntimeException. It's possible to catch all RuntimeExceptions with a single catch clause. For example:

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( RuntimeException err ) {
   System.out.println("Sorry, an error has occurred.");
   System.out.println("The error was: " + err);
}
```

The catch clause in this try statement will catch any exception belonging to class RuntimeException or to any of its subclasses. This shows why exception classes are organized into a class hierarchy. It allows you the option of casting your net narrowly to catch only a specific type of exception. Or you can cast your net widely to catch a wide class of exceptions. Because of subclassing, when there are multiple catch clauses in a try statement, it is possible that a given exception might match several of those catch clauses. For example, an exception of type NullPointerException would match catch clauses for NullPointerException, RuntimeException, Exception, or Throwable. In this case, only the **first** catch clause that matches the exception is executed.

The example I've given here is not particularly realistic. You are not very likely to use exception-handling to guard against null pointers and bad array indices. This is a case where careful programming is better than exception handling: Just be sure that your program assigns a reasonable, non-null value to the array M. You would certainly resent it if the designers of Java forced you to set up a **try..catch** statement every time you wanted to use an array! This is why handling of potential RuntimeExceptions is not mandatory. There are just too many things that might go wrong! (This also shows that exception-handling does not solve the problem of program robustness. It just gives you a tool that will in many cases let you approach the problem in a more organized way.)

I have still not completely specified the syntax of the try statement. There is one additional element: the possibility of a finally clause at the end of a try statement. The complete syntax of the try statement can be described as:

```
try {
   statements
}
optional–catch–clauses
optional–finally–clause
```

Note that the catch clauses are also listed as optional. The try statement can include zero or more catch clauses and, optionally, a **finally** clause. The **try** statement **must** include one or the other. That is, a **try** statement can have either a **finally** clause, or one or more catch clauses, or both. The syntax for a catch clause is

```
catch ( exception–class–name variable–name ) {
   statements
}
```

and the syntax for a finally clause is

```
finally {
   statements
}
```

The semantics of the **finally** clause is that the block of statements in the **finally** clause is guaranteed to be executed as the last step in the execution of the try statement, whether or not any exception occurs and whether or not any exception that does occur is caught and handled. The **finally** clause is meant for doing essential cleanup that under no circumstances should be omitted. One example of this type of cleanup is closing a network connection. Although you don't yet know enough about networking to look at the actual programming in this case, we can consider some pseudocode:

```
try {
   open a network connection
}
catch ( IOException e ) {
   report the error
   return  // Don't continue if connection can't be opened!
}

// At this point, we KNOW that the connection is open.

try {
   communicate over the connection
}
catch ( IOException e ) {
   handle the error
}
finally {
   close the connection
}
```

The **finally** clause in the second **try** statement ensures that the network connection will definitely be closed, whether or not an error occurs during the communication. The first **`try`** statement is there to make sure that we don't even try to communicate over the network unless we have successfully opened a connection. The pseudocode in this example follows a general pattern that can be used to robustly obtain a resource, use the resource, and then release the resource.

9.3.3 Throwing Exceptions

There are times when it makes sense for a program to deliberately throw an exception. This is the case when the program discovers some sort of exceptional or error condition, but there is no reasonable way to handle the error at the point where the problem is discovered. The program can throw an exception in the hope that some other part of the program will catch and handle the exception. This can be done with a throw statement. In this section, we cover the throw statement more fully. The syntax of the throw statement is: **`throw`** `exception−object ;`

The exception-object must be an object belonging to one of the subclasses of `Throwable`. Usually, it will in fact belong to one of the subclasses of `Exception`. In most cases, it will be a newly constructed object created with the new operator. For example: **`throw new`** `ArithmeticException("Division by zero");`

The parameter in the constructor becomes the error message in the exception object; if `e` refers to the object, the error message can be retrieved by calling `e.getMessage()`. (You might find this example a bit odd, because you might expect the system itself to throw an `ArithmeticException` when an attempt is made to divide by zero. So why should a programmer bother to throw the exception? Recalls that if the numbers that are being divided are of type **`int`**, then division by zero will indeed throw an `ArithmeticException`. However, no arithmetic operations with floating-point numbers will ever produce an exception. Instead, the special value `Double.NaN` is used to represent the result of an illegal operation. In some situations, you might prefer to throw an `ArithmeticException` when a real number is divided by zero.)

An exception can be thrown either by the system or by a throw statement. The exception is processed in exactly the same way in either case. Suppose that the exception is thrown inside a try statement. If that try statement has a catch clause that handles that type of exception, then the computer jumps to the catch clause and executes it. The exception has been handled. After handling the exception, the computer executes the finally clause of the try statement, if there is one. It then continues normally with the rest of the program, which follows the try statement. If the exception is not immediately caught and handled, the processing of the exception will continue.

When an exception is thrown during the execution of a method and the exception is not handled in the same method, then that method is terminated (after the execution of any pending finally clauses). Then the method that called that method gets a chance to handle the exception. That is, if the method was called inside a try statement that has an appropriate catch clause, then **that** catch clause will be executed and the program will continue on normally from there. Again, if the second method does not handle the exception, then it also is terminated and the method that called **it** (if any) gets the next shot at the exception. The exception will crash the program only if it passes up through the entire chain of method calls without being handled.

A method that might generate an exception can announce this fact by adding a clause "throws exception-class-name" to the header of the method. For example:

```
/**
 * Returns the larger of the two roots of the quadratic equation
 * A*x*x + B*x + C = 0, provided it has any roots.  If A == 0 or
 * if the discriminant, B*B - 4*A*C, is negative, then an exception
 * of type IllegalArgumentException is thrown.
 */
static public double root( double A, double B, double C )
                              throws IllegalArgumentException {
    if (A == 0) {
      throw new IllegalArgumentException("A can't be zero.");
    }
    else {
       double disc = B*B - 4*A*C;
       if (disc < 0)
          throw new IllegalArgumentException("Discriminant < zero.");
       return  (-B + Math.sqrt(disc)) / (2*A);
    }
}
```

As discussed in the previous section, the computation in this method has the preconditions that $A! = 0$ and $B * B - 4 * A * C \ >= \ 0$. The method throws an exception of type IllegalArgumentException when either of these preconditions is violated. When an illegal condition is found in a method, throwing an exception is often a reasonable response. If the program that called the method knows some good way to handle the error, it can catch the exception. If not, the program will crash – and the programmer will know that the program needs to be fixed.

A throws clause in a method heading can declare several different types of exceptions, separated by commas. For example:

```
void processArray(int[] A) throws NullPointerException,
                                  ArrayIndexOutOfBoundsException { ...
```

9.3.4 Mandatory Exception Handling

In the preceding example, declaring that the method root() can throw an IllegalArgumentException is just a courtesy to potential readers of this method. This is because handling of IllegalArgumentExceptions is not "mandatory". A method can throw an IllegalArgumentException without announcing the possibility. And a program that calls that method is free either to catch or to ignore the exception, just as a programmer can choose either to catch or to ignore an exception of type NullPointerException.

For those exception classes that require mandatory handling, the situation is different. If a method can throw such an exception, that fact **must** be announced in a throws clause in the method definition. Failing to do so is a syntax error that will be reported by the compiler.

On the other hand, suppose that some statement in the body of a method can generate an exception of a type that requires mandatory handling. The statement could be a throw statement, which throws the exception directly, or it could be a call to a method that can throw the exception. In either case, the exception **must** be handled. This can be done in one of two ways: The first way is to place the statement in a try statement that has a catch clause that handles the exception; in this case, the exception is handled within the method, so that any caller of the method will never see the exception. The second way is to declare that the method can throw the exception. This is done by adding a "throws" clause to the method heading, which alerts any callers to the possibility that an exception might be generated when the method is executed. The caller will, in turn, be forced either to handle the exception in a try statement or to declare the exception in a throws clause in its own header.

Exception-handling is mandatory for any exception class that is not a subclass of either Error or RuntimeException. Exceptions that require mandatory handling generally represent conditions that are outside the control of the programmer. For example, they might represent bad input or an illegal action taken by the user. There is no way to **avoid** such errors, so a robust program has to be prepared to handle them. The design of Java makes it impossible for programmers to ignore the possibility of such errors.

Among the exceptions that require mandatory handling are several that can occur when using Java's input/output methods. This means that you can't even use these methods unless you understand something about exception-handling.

9.3.5 Programming with Exceptions

Exceptions can be used to help write robust programs. They provide an organized and structured approach to robustness. Without exceptions, a program can become cluttered with if statements that test for various possible error conditions. With exceptions, it becomes possible to write a clean implementation of an algorithm that will handle all the normal cases. The exceptional cases can be handled elsewhere, in a catch clause of a try statement.

When a program encounters an exceptional condition and has no way of handling it immediately, the program can throw an exception. In some cases, it makes sense to throw an exception belonging to one of Java's predefined classes, such as IllegalArgumentException or IOException. However, if there is no standard class that adequately represents the exceptional condition, the programmer can define a new exception class. The new class must extend the standard class Throwable or one of its subclasses. In general, if the programmer does **not** want to require mandatory exception handling, the new class will extend RuntimeException (or one of its subclasses). To create a new exception class that **does** require mandatory handling, the programmer can extend one of the other subclasses of Exception or can extend Exception itself.

Here, for example, is a class that extends Exception, and therefore requires mandatory exception handling when it is used:

```
public class ParseError extends Exception {
   public ParseError(String message) {
         // Create a ParseError object containing
         // the given message as its error message.
      super(message);
   }
}
```

The class contains only a constructor that makes it possible to create a ParseError object containing a given error message. (The statement "**super**(message)" calls a constructor in the superclass, Exception.) The class inherits the getMessage() and printStackTrace() methods from its superclass, off course. If e refers to an object of type ParseError, then the method call e.getMessage() will retrieve the error message that was specified in the constructor. But the main point of the ParseError class is simply to exist. When an object of type ParseError is thrown, it indicates that a certain type of error has occurred. (Parsing, by the way, refers to figuring out the syntax of a string. A ParseError would indicate, presumably, that some string that is being processed by the program does not have the expected form.)

A throw statement can be used in a program to throw an error of type ParseError. The constructor for the ParseError object must specify an error message. For example:

```
throw new ParseError("Encountered an illegal negative number.");
```

or

```
throw new ParseError("The word '" + word
                              + "' is not a valid file name.");
```

If the throw statement does not occur in a try statement that catches the error, then the method that contains the throw statement must declare that it can throw a ParseError by adding the clause "**throws** ParseError" to the method heading. For example,

```
void getUserData() throws ParseError {
   . . .
}
```

This would not be required if ParseError were defined as a subclass of RuntimeException instead of Exception, since in that case exception handling for ParseErrors would not be mandatory.

A method that wants to handle ParseErrors can use a try statement with a catch clause that catches ParseErrors. For example:

```
try {
   getUserData();
   processUserData();
}
catch (ParseError pe) {
   . . .  // Handle the error
}
```

Note that since ParseError is a subclass of Exception, a catch clause of the form "**catch** (Exception e)" would also catch ParseErrors, along with any other object of type Exception.

Sometimes, it's useful to store extra data in an exception object. For example,

```
class ShipDestroyed extends RuntimeException {
   Ship ship;  // Which ship was destroyed.
   int where_x, where_y;  // Location where ship was destroyed.
   ShipDestroyed(String message, Ship s, int x, int y) {
           // Constructor creates a ShipDestroyed object
           // carrying an error message plus the information
           // that the ship s was destroyed at location (x,y)
           // on the screen.
        super(message);
        ship = s;
        where_x = x;
        where_y = y;
   }
}
```

Here, a ShipDestroyed object contains an error message and some information about a ship that was destroyed. This could be used, for example, in a statement:

```
if ( userShip.isHit() )
   throw new ShipDestroyed("You've been hit!", userShip, xPos, yPos);
```

Note that the condition represented by a ShipDestroyed object might not even be considered an error. It could be just an expected interruption to the normal flow of a game. Exceptions can sometimes be used to handle such interruptions neatly.

The ability to throw exceptions is particularly useful in writing general-purpose methods and classes that are meant to be used in more than one program. In this case, the person writing the method or class often has no reasonable way of handling the error, since that person has no way of knowing exactly how the method or class will be used. In such circumstances, a novice programmer is often tempted to print an error message and forge ahead, but this is almost never satisfactory since it can lead to unpredictable results down the line. Printing an error message and terminating the program is almost as bad, since it gives the program no chance to handle the error.

The program that calls the method or uses the class needs to know that the error has occurred. In languages that do not support exceptions, the only alternative is to return some special value or to set the value of some variable to indicate that an error has occurred. For example, a method may return the value -1 if the user's input is illegal. However, this only does any good if the main program bothers to test the return value. It is very easy to be lazy about checking for special return values every time a method is called. And in this case, using -1 as a signal that an error has occurred makes it impossible to allow negative return values. Exceptions are a cleaner way for a method to react when it encounters an error.

9.4 Assertions

WE END THIS CHAPTER WITH A SHORT SECTION ON ASSERTIONS, another feature of the Java programming language that can be used to aid in the development of correct and robust programs.

Recall that a precondition is a condition that must be true at a certain point in a program, for the execution of the program to continue correctly from that point. In the case where there is a chance that the precondition might not be satisfied – for example, if it depends on input from the user – then it's a good idea to insert an if statement to test it. But then the question arises, What should be done if the precondition does not hold? One option is to throw an exception. This will terminate the program, unless the exception is caught and handled elsewhere in the program.

In many cases, of course, instead of using an if statement to *test* whether a precondition holds, a programmer tries to write the program in a way that will *guarantee* that the precondition holds. In that case, the test should not be necessary, and the if statement can be avoided. The problem is that programmers are not perfect. In spite of the programmer's intention, the program might contain a bug that screws up the precondition. So maybe it's a good idea to check the precondition – at least during the debugging phase of program development.

Similarly, a postcondition is a condition that is true at a certain point in the program as a consequence of the code that has been executed before that point. Assuming that the code is correctly written, a postcondition is guaranteed to be true, but here again testing whether a desired postcondition is **actually** true is a way of checking for a bug that might have screwed up the postcondition. This is somthing that might be desirable during debugging.

The programming languages C and C++ have always had a facility for adding what are called assertions to a program. These assertions take the form "assert(condition)", where condition is a boolean-valued expression. This condition expresses a precondition or postcondition that should hold at that point in the program. When the computer encounters an assertion during the execution of the program, it evaluates the condition. If the condition is false, the program is terminated. Otherwise, the program continues normally. This allows the programmer's belief that the condition is true to be tested; if if it not true, that indicates that the part of the program that preceded the assertion contained a bug. One nice thing about assertions in C and C++ is that they can be "turned off" at compile time. That is, if the program is compiled in one way, then the assertions are included in the compiled code. If the program is compiled in another way, the assertions are not included. During debugging, the first type of compilation is used. The release version of the program is compiled with assertions turned off. The release version will be more efficient, because the computer won't have to evaluate all the assertions.

Although early versions of Java did not have assertions, an assertion facility similar to the one in C/C++ has been available in Java since version 1.4. As with the C/C++ version, Java assertions can be turned on during debugging and turned off during normal execution. In Java, however, assertions are turned on and off at run time rather than at compile time. An assertion in the Java source code is always included in the compiled class file. When the program is run in the normal way, these assertions are ignored; since the condition in the assertion is not evaluated in this case, there is little or no performance penalty for having the assertions in the program. When the program is being debugged, it can be run with assertions enabled, as discussed below, and then the assertions can be a great help in locating and identifying bugs.

An assertion statement in Java takes one of the following two forms: `assert condition ;` or `assert condition : error−message ;` where condition is a boolean-valued expression and error-message is a string or an expression of type `String`. The word "assert" is a reserved word in Java, which cannot be used as an identifier. An assertion statement can be used anyplace in Java where a statement is legal.

If a program is run with assertions disabled, an assertion statement is equivalent to an empty statement and has no effect. When assertions are enabled and an assertion statement is encountered in the program, the condition in the assertion is evaluated. If the value is true, the program proceeds normally. If the value of the condition is false, then an exception of type `java.lang.AssertionError` is thrown, and the program will crash (unless the error is caught by a try statement). If the assert statement includes an error-message, then the error message string becomes the message in the `AssertionError`.

So, the statement "assert condition : error-message;" is similar to

```
if ( condition == false )
    throw new AssertionError( error−message );
```

except that the if statement is executed whenever the program is run, and the assert statement is executed only when the program is run with assertions enabled.

The question is, when to use assertions instead of exceptions? The general rule is to use assertions to test conditions that should definitely be true, if the program is written correctly. Assertions are useful for testing a program to see whether or not it is correct and for finding the errors in an incorrect program. After testing and debugging, when the program is used in the normal way, the assertions in the program will be ignored. However, if a problem turns up later, the assertions are still there in the program to be used to help locate the error. If someone writes to you to say that your program doesn't work when he does such-and-such, you can run the program with assertions enabled, do such-and-such, and hope that the assertions in the program will help you locate the point in the program where it goes wrong.

Consider, for example, the `root()` method that calculates a root of a quadratic equation. If you believe that your program will always call this method with legal arguments, then it would make sense to write the method using assertions instead of exceptions:

```
/**
 * Returns the larger of the two roots of the quadratic equation
 * A*x*x + B*x + C = 0, provided it has any roots.
 * Precondition: A != 0 and B*B - 4*A*C >= 0.
 */
static public double root( double A, double B, double C )  {
   assert A != 0 : "Leading coefficient of quadratic equation cannot be zero.";
   double disc = B*B - 4*A*C;
   assert disc >= 0 : "Discriminant of quadratic equation cannot be negative.";
   return  (-B + Math.sqrt(disc)) / (2*A);
}
```

The assertions are not checked when the program is run in the normal way. If you are correct in your belief that the method is never called with illegal arguments, then checking the conditions in the assertions would be unnecessary. If your belief is not correct, the problem should turn up during testing or debugging, when the program is run with the assertions enabled.

If the `root()` method is part of a software library that you expect other people to use, then the situation is less clear. Sun's Java documentation advises that assertions should **not** be used for checking the contract of public methods: If the caller of a method violates the contract by passing illegal parameters, then an exception should be thrown. This will enforce the contract whether or not assertions are enabled. (However, while it's true that Java programmers *expect* the contract of a method to be enforced with exceptions, there are reasonable arguments for using assertions instead, in some cases.)

On the other hand, it never hurts to use an assertion to check a postcondition of a method. A postcondition is something that is supposed to be true after the method has executed, and it can be tested with an assert statement at the end of the method. If the postcodition is false, there is a bug in the method itself, and that is something that needs to be found during the development of the method.

To have any effect, assertions must be **enabled** when the program is run. How to do this depends on what programming environment you are using. In the usual command line environment, assertions are enabled by adding the `-enableassertions` option to the java command that is used to run the program. For example, if the class that contains the main program is `RootFinder`, then the command

```
java -enableassertions RootFinder
```

will run the program with assertions enabled. The `-enableassertions` option can be abbreviated to `-ea`, so the command can alternatively be written as `java -ea RootFinder`.

In fact, it is possible to enable assertions in just part of a program. An option of the form "-ea:class-name" enables only the assertions in the specified class. Note that there are no spaces between the -ea, the ":", and the name of the class. To enable all the assertions in a package and in its sub-packages, you can use an option of the form "-ea:package-name...". To enable assertions in the "default package" (that is, classes that are not specified to belong to a package, like almost all the classes in this book), use "-ea:...". For example, to run a Java program named "MegaPaint" with assertions enabled for every class in the packages named "paintutils" and "drawing", you would use the command:

```
java  −ea:paintutils...  −ea:drawing...  MegaPaint
```

If you are using the Eclipse integrated development environment, you can specify the -ea option by creating a run configuration. Right-click the name of the main program class in the Package Explorer pane, and select "Run As" from the pop-up menu and then "Run..." from the submenu. This will open a dialog box where you can manage run configurations. The name of the project and of the main class will be already be filled in. Click the "Arguments" tab, and enter -ea in the box under "VM Arguments". The contents of this box are added to the java command that is used to run the program. You can enter other options in this box, including more complicated enableassertions options such as -ea:paintutils.... When you click the "Run" button, the options will be applied. Furthermore, they will be applied whenever you run the program, unless you change the run configuration or add a new configuration. Note that it is possible to make two run configurations for the same class, one with assertions enabled and one with assertions disabled.

Chapter 10

Input and Output

Contents

10.1 Streams, Readers, and Writers

Without the ability to interact with the rest of the world, a program would be useless. The interaction of a program with the rest of the world is referred to as input/output or I/O. Historically, one of the hardest parts of programming language design has been coming up with good facilities for doing input and output. A computer can be connected to many different types of input and output devices. If a programming language had to deal with each type of device as a special case, the complexity would be overwhelming. One of the major achievements in the history of programming has been to come up with good abstractions for representing I/O devices. In Java, the main I/O abstractions are called streams. Other I/O abstractions, such as "files" and "channels" also exist, but in this section we will look only at streams. Every stream represents either a source of input or a destination to which output can be sent.

10.1.1 Character and Byte Streams

When dealing with input/output, you have to keep in mind that there are two broad categories of data: machine-formatted data and human-readable data. Machine-formatted data is represented in binary form, the same way that data is represented inside the computer, that is, as strings of zeros and ones. Human-readable data is in the form of characters. When you read a number such as 3.141592654, you are reading a sequence of characters and interpreting them as a number. The same number would be represented in the computer as a bit-string that you would find unrecognizable.

To deal with the two broad categories of data representation, Java has two broad categories of streams: byte streams for machine-formatted data and character streams for human-readable data. There are many predefined classes that represent streams of each type.

An object that **outputs** data to a byte stream belongs to one of the subclasses of the abstract class OutputStream. Objects that **read** data from a byte stream belong to subclasses of InputStream. If you write numbers to an OutputStream, you won't be able to read the resulting data yourself. But the data can be read back into the computer with an InputStream. The writing and reading of the data will be very efficient, since there is no translation involved: the bits that are used to represent the data inside the computer are simply copied to and from the streams.

For reading and writing human-readable character data, the main classes are the abstract classes Reader and Writer. All character stream classes are subclasses of one of these. If a number is to be written to a Writer stream, the computer must translate it into a human-readable sequence of characters that represents that number. Reading a number from a Reader stream into a numeric variable also involves a translation, from a character sequence into the appropriate bit string. (Even if the data you are working with consists of characters in the first place, such as words from a text editor, there might still be some translation. Characters are stored in the computer as $16-$bit Unicode values. For people who use Western alphabets, character data is generally stored in files in ASCII code, which uses only 8 bits per character. The Reader and Writer classes take care of this translation, and can also handle non-western alphabets in countries that use them.)

Byte streams can be useful for direct machine-to-machine communication, and they can sometimes be useful for storing data in files, especially when large amounts of data need to be stored efficiently, such as in large databases. However, binary data is *fragile* in the sense that its meaning is not self-evident. When faced with a long series of zeros and ones, you have to know what information it is meant to represent and how that information is encoded before you will be able to interpret it. Of course, the same is true to some extent for character data, which is itself coded into binary form. But the binary encoding of character data has been standardized and is well understood, and data expressed in character form can be made meaningful to human readers. The current trend seems to be towards increased use of character data, represented in a way that will make its meaning as self-evident as possible.

I should note that the original version of Java did not have character streams, and that for ASCII-encoded character data, byte streams are largely interchangeable with character streams. In fact, the standard input and output streams, System.in and System.out, are byte streams rather than character streams. However, you should use Readers and Writers rather than InputStreams and OutputStreams when working with character data.

The standard stream classes discussed in this section are defined in the package java.io, along with several supporting classes. You must import the classes from this package if you want to use them in your program. That means either importing individual classes or putting the directive "import java.io.*;" at the beginning of your source file. Streams are necessary for working with files and for doing communication over a network. They can be also used for communication between two concurrently running threads, and there are stream classes for reading and writing data stored in the computer's memory.

The beauty of the stream abstraction is that it is as easy to write data to a file or to send data over a network as it is to print information on the screen.

The basic I/O classes `Reader`, `Writer`, `InputStream`, and `OutputStream` provide only very primitive I/O operations. For example, the `InputStream` class declares the instance method **`public int`** `read()` **`throws`** `IOException` for reading one byte of data, as a number in the range 0 to 255, from an input stream. If the end of the input stream is encountered, the `read()` method will return the value -1 instead. If some error occurs during the input attempt, an exception of type `IOException` is thrown. Since `IOException` is an exception class that requires mandatory exception-handling, this means that you can't use the `read()` method except inside a **`try`** statement or in a method that is itself declared with a "**`throws`** `IOException`" clause.

The `InputStream` class also defines methods for reading several bytes of data in one step into an array of bytes. However, `InputStream` provides no convenient methods for reading other types of data, such as int or double, from a stream. This is not a problem because you'll never use an object of type `InputStream` itself. Instead, you'll use subclasses of `InputStream` that add more convenient input methods to `InputStream`'s rather primitive capabilities. Similarly, the `OutputStream` class defines a primitive output method for writing one byte of data to an output stream. The method is defined as:**`public void`** `write(`**`int`** `b)` **`throws`** `IOException` The parameter is of type int rather than byte, but the parameter value is type-cast to type byte before it is written; this effectively discards all but the eight low order bytes of `b`. Again, in practice, you will almost always use higher-level output operations defined in some subclass of `OutputStream`.

The `Reader` and `Writer` classes provide identical low-level `read` and `write` methods. As in the byte stream classes, the parameter of the `write(c)` method in `Writer` and the return value of the read() method in `Reader` are of type **`int`**, but in these character-oriented classes, the I/O operations read and write characters rather than bytes. The return value of read() is -1 if the end of the input stream has been reached. Otherwise, the return value must be type-cast to type char to obtain the character that was read. In practice, you will ordinarily use higher level I/O operations provided by sub-classes of `Reader` and `Writer`, as discussed below.

10.1.2 PrintWriter

One of the neat things about Java's I/O package is that it lets you add capabilities to a stream by "wrapping" it in another stream object that provides those capabilities. The wrapper object is also a stream, so you can read from or write to it–but you can do so using fancier operations than those available for basic streams.

For example, `PrintWriter` is a subclass of `Writer` that provides convenient methods for outputting human-readable character representations of all of Java's basic data types. If you have an object belonging to the `Writer` class, or any of its subclasses, and you would like to use `PrintWriter` methods to output data to that `Writer`, all you have to do is wrap the `Writer` in a `PrintWriter` object. You do this by constructing a new `PrintWriter` object, using the `Writer` as input to the constructor. For example, if `charSink` is of type `Writer`, then you could say

```
PrintWriter printableCharSink = new PrintWriter(charSink);
```

When you output data to printableCharSink, using the high-level output methods in PrintWriter, that data will go to exactly the same place as data written directly to charSink. You've just provided a better interface to the same output stream. For example, this allows you to use PrintWriter methods to send data to a file or over a network connection.

For the record, if out is a variable of type PrintWriter, then the following methods are defined:

- out.print(x)–prints the value of x, represented in the form of a string of characters, to the output stream; x can be an expression of any type, including both primitive types and object types. An object is converted to string form using its toString() method. A null value is represented by the string "null".
- out.println()–outputs an end-of-line to the output stream.
- out.println(x)–outputs the value of x, followed by an end-of-line; this is equivalent to out.print(x) followed by out.println().
- out.printf(formatString, x1, x2, ...)–does formated output of x1, x2, ... to the output stream. The first parameter is a string that specifies the format of the output. There can be any number of additional parameters, of any type, but the types of the parameters must match the formatting directives in the format string.

Note that none of these methods will ever throw an IOException. Instead, the PrintWriter class includes the method **public boolean** checkError() which will return true if any error has been encountered while writing to the stream. The PrintWriter class catches any IOExceptions internally, and sets the value of an internal error flag if one occurs. The checkError() method can be used to check the error flag. This allows you to use PrintWriter methods without worrying about catching exceptions. On the other hand, to write a fully robust program, you should call checkError() to test for possible errors whenever you used a PrintWriter.

10.1.3 Data Streams

When you use a PrintWriter to output data to a stream, the data is converted into the sequence of characters that represents the data in human-readable form. Suppose you want to output the data in byte-oriented, machine-formatted form? The java.io package includes a byte-stream class, DataOutputStream that can be used for writing data values to streams in internal, binary-number format. DataOutputStream bears the same relationship to OutputStream that PrintWriter bears to Writer. That is, whereas OutputStream only has methods for outputting bytes, DataOutputStream has methods writeDouble(**double** x) for outputting values of type double, writeInt(**int** x) for outputting values of type int, and so on. Furthermore, you can wrap any OutputStream in a DataOutputStream so that you can use the higher level output methods on it. For example, if byteSink is of type classname, you could say

```
DataOutputStream dataSink = new DataOutputStream(byteSink);
```

to wrap byteSink in a DataOutputStream, dataSink.

For input of machine-readable data, such as that created by writing to a DataOutputStream, java.io provides the class DataInputStream. You can wrap any InputStream in aDataInputStream object to provide it with the ability to read data of various types from the byte-stream. The methods in theDataInputStream for reading binary data are called readDouble(), readInt(), and so on. Data written by a DataOutputStream is guaranteed to be in a format that can be read by a DataInputStream. This is true even if the data stream is created on one type of computer and read on another type of computer. The cross-platform compatibility of binary data is a major aspect of Java's platform independence.

In some circumstances, you might need to read character data from an InputStream or write character data to an OutputStream. This is not a problem, since characters, like all data, are represented as binary numbers. However, for character data, it is convenient to use Reader and Writer instead of InputStream and OutputStream. To make this possible, you can **wrap** a byte stream in a character stream. If byteSource is a variable of type InputStream and byteSink is of type OutputStream, then the statements

```
Reader charSource = new InputStreamReader( byteSource );
Writer charSink   = new OutputStreamWriter( byteSink );
```

create character streams that can be used to read character data from and write character data to the byte streams. In particular, the standard input stream System.in, which is of type InputStream for historical reasons, can be wrapped in a Reader to make it easier to read character data from standard input:

```
Reader charIn = new InputStreamReader( System.in );
```

As another application, the input and output streams that are associated with a network connection are byte streams rather than character streams, but the byte streams can be wrapped in character streams to make it easy to send and receive character data over the network.

10.1.4 Reading Text

Still, the fact remains that much I/O is done in the form of human-readable characters. In view of this, it is surprising that Java does **not** provide a standard character input class that can read character data in a manner that is reasonably symmetrical with the character output capabilities of PrintWriter. There is one basic case that is easily handled by a standard class. The BufferedReader class has a method **public** String readLine() **throws** IOException that reads one line of text from its input source. If the end of the stream has been reached, the return value is **null**. When a line of text is read, the end-of-line marker is read from the input stream, but it is not part of the string that is returned. Different input streams use different characters as end-of-line markers, but the readLine method can deal with all the common cases.

Line-by-line processing is very common. Any Reader can be wrapped in a BufferedReader to make it easy to read full lines of text. If reader is of type Reader, then a BufferedReader wrapper can be created for reader with BufferedReader in = **new** BufferedReader(reader);.

This can be combined with the InputStreamReader class that was mentioned above to read lines of text from an InputStream. For example, we can apply this to System.in:

```
BufferedReader in;  // BufferedReader for reading from standard input.
in = new BufferedReader( new InputStreamReader( System.in ) );
try {
   String line = in.readLine();
   while ( line != null && line.length() > 0 ) {
      processOneLineOfInput( line );
      line = in.readLine();
   }
}
catch (IOException e) {
}
```

This code segment reads and processes lines from standard input until either an empty line or an end-of-stream is encountered. (An end-of-stream is possible even for interactive input. For example, on at least some computers, typing a Control-D generates an end-of-stream on the standard input stream.) The **try..catch** statement is necessary because the readLine method can throw an exception of type IOException, which requires mandatory exception handling; an alternative to try..catch would be to declare that the method that contains the code "**throws** IOException". Also, remember that BufferedReader, InputStreamReader, and IOException must be imported from the package java.io.

10.1.5 The Scanner Class

Since its introduction, Java has been notable for its lack of built-in support for basic input, and for its reliance on fairly advanced techniques for the support that it does offer. (This is my opinion, at least.) The Scanner class was introduced in Java 5.0 to make it easier to read basic data types from a character input source. It does not (again, in my opinion) solve the problem completely, but it is a big improvement. The Scanner class is in the package java.util.

Input methods are defined as instance methods in the Scanner class, so to use the class, you need to create a Scanner object. The constructor specifies the source of the characters that the Scanner will read. The scanner acts as a wrapper for the input source. The source can be a Reader, an InputStream, a String, or a File. (If a String is used as the input source, the Scanner will simply read the characters in the string from beginning to end, in the same way that it would process the same sequence of characters from a stream. The File class will be covered in the next section.) For example, you can use a Scanner to read from standard input by saying:

```
Scanner standardInputScanner = new Scanner( System.in );
```

and if charSource is of type Reader, you can create a Scanner for reading from charSource with:

```
Scanner scanner = new Scanner( charSource );
```

When processing input, a scanner usually works with tokens. A token is a meaningful string of characters that cannot, for the purposes at hand, be further broken down into smaller meaningful pieces. A token can, for example, be an individual word or a string of characters that represents a value of type double. In the case of a scanner, tokens must be separated by "delimiters." By default, the delimiters are whitespace characters such as spaces and end-of-line markers. In normal processing, whitespace characters serve simply to separate tokens and are discarded by the scanner. A scanner has instance methods for reading tokens of various types. Suppose that scanner is an object of type Scanner. Then we have:

- scanner.next()–reads the next token from the input source and returns it as a String.

- scanner.nextInt(), scanner.nextDouble(), and so on–reads the next token from the input source and tries to convert it to a value of type int, double, and so on. There are methods for reading values of any of the primitive types.

- scanner.nextLine()–reads an entire line from the input source, up to the next end-of-line and returns the line as a value of type String. The end-of-line marker is read but is not part of the return value. Note that this method is **not** based on tokens. An entire line is read and returned, including any whitespace characters in the line.

All of these methods can generate exceptions. If an attempt is made to read past the end of input, an exception of type NoSuchElementException is thrown. Methods such as scanner.getInt() will throw an exception of type InputMismatchException if the next token in the input does not represent a value of the requested type. The exceptions that can be generated do not require mandatory exception handling.

The Scanner class has very nice look-ahead capabilities. You can query a scanner to determine whether more tokens are available and whether the next token is of a given type. If scanner is of type Scanner:

- scanner.hasNext()–returns a boolean value that is true if there is at least one more token in the input source.

- scanner.hasNextInt(), scanner.hasNextDouble(), and so on–returns a boolean value that is true if there is at least one more token in the input source and that token represents a value of the requested type.

- scanner.hasNextLine()–returns a boolean value that is true if there is at least one more line in the input source.

Although the insistence on defining tokens only in terms of delimiters limits the usability of scanners to some extent, they are easy to use and are suitable for many applications.

10.2 Files

The data and programs in a computer's main memory survive only as long as the power is on. For more permanent storage, computers use files, which are collections of data stored on a hard disk, on a USB memory stick, on a CD-ROM, or on some other type of storage device. Files are organized into directories (sometimes called folders). A directory can hold other directories, as well as files. Both directories and files have names that are used to identify them.

Programs can read data from existing files. They can create new files and can write data to files. In Java, such input and output can be done using streams. Human-readable character data is read from a file using an object belonging to the class FileReader, which is a subclass of Reader. Similarly, data is written to a file in human-readable format through an object of type FileWriter, a subclass of Writer. For files that store data in machine format, the appropriate I/O classes are FileInputStream and FileOutputStream. In this section, I will only discuss character-oriented file I/O using the FileReader and FileWriter classes. However, FileInputStream and FileOutputStream are used in an exactly parallel fashion. All these classes are defined in the java.io package.

It's worth noting right at the start that applets which are downloaded over a network connection are not allowed to access files (unless you have made a very foolish change to your web browser's configuration). This is a security consideration. You can download and run an applet just by visiting a Web page with your browser. If downloaded applets had access to the files on your computer, it would be easy to write an applet that would destroy all the data on a computer that downloads it. To prevent such possibilities, there are a number of things that downloaded applets are not allowed to do. Accessing files is one of those forbidden things. Standalone programs written in Java, however, have the same access to your files as any other program. When you write a standalone Java application, you can use all the file operations described in this section.

10.2.1 Reading and Writing Files

The FileReader class has a constructor which takes the name of a file as a parameter and creates an input stream that can be used for reading from that file. This constructor will throw an exception of type FileNotFoundException if the file doesn't exist. It requires mandatory exception handling, so you have to call the constructor in a **try..catch** statement (or inside a method that is declared to throw the exception). For example, suppose you have a file named "data.txt", and you want your program to read data from that file. You could do the following to create an input stream for the file:

```
FileReader data;    // (Declare the variable before the
                    //   try statement, or else the variable
                    //   is local to the try block and you won't
                    //   be able to use it later in the program.)

try {
   data = new FileReader("data.txt");  // create the stream
}
catch (FileNotFoundException e) {
   ... // do something to handle the error -- maybe, end the program
}
```

The FileNotFoundException class is a subclass of IOException, so it would be acceptable to catch IOExceptions in the above **try...catch** statement. More generally, just about any error that can occur during input/output operations can be caught by a catch clause that handles IOException.

Once you have successfully created a FileReader, you can start reading data from it. But since FileReaders have only the primitive input methods inherited from the basic Reader class, you will probably want to wrap your FileReader in a Scanner, or in some other wrapper class.

Working with output files is no more difficult than this. You simply create an object belonging to the class FileWriter. You will probably want to wrap this output stream in an object of type PrintWriter. For example, suppose you want to write data to a file named "result.dat". Since the constructor for FileWriter can throw an exception of type IOException, you should use a **try..catch** statement:

```
PrintWriter result;

try {
   result = new PrintWriter(new FileWriter("result.dat"));
}
catch (IOException e) {
   ... // handle the exception
}
```

If no file named result.dat exists, a new file will be created. If the file already exists, then the current contents of the file will be erased and replaced with the data that your program writes to the file. This will be done without any warning. To avoid overwriting a file that already exists, you can check whether a file of the same name already exists before trying to create the stream, as discussed later in this section. An IOException might occur in the PrintWriter constructor if, for example, you are trying to create a file on a disk that is "write-protected," meaning that it cannot be modified.

After you are finished using a file, it's a good idea to close the file, to tell the operating system that you are finished using it. You can close a file by calling the close() method of the associated stream. Once a file has been closed, it is no longer possible to read data from it or write data to it, unless you open it again as a new stream. (Note that for most stream classes, the close() method can throw an IOException, which must be handled; PrintWriter overrides this method so that it cannot throw such exceptions.) If you forget to close a file, the file will ordinarily be closed automatically when the program terminates or when the file object is garbage collected, but in the case of an output file, some of the data that has been written to the file might be lost. This can occur because data that is written to a file can be buffered; that is, the data is not sent immediately to the file but is retained in main memory (in a "buffer") until a larger chunk of data is ready to be written. This is done for efficiency. The close() method of an output stream will cause all the data in the buffer to be sent to the file. Every output stream also has a flush() method that can be called to force any data in the buffer to be written to the file without closing the file.

As a complete example, here is a program that will read numbers from a file named data.dat, and will then write out the same numbers in reverse order to another file named result.dat. It is assumed that data.dat contains only one number on each line. Exception-handling is used to check for problems along the way. Although the application is not a particularly useful one, this program demonstrates the basics of working with files. (By the way, at the end of this program, you'll find our first example of a finally clause in a try statement. When the computer executes a try statement, the commands in its finally clause are guaranteed to be executed, no matter what.)

```
import java.io.*;
import java.util.ArrayList;
/**
 * Reads numbers from a file named data.dat and writes them to a file
 * named result.dat in reverse order.  The input file should contain
 * exactly one real number per line.
 */
public class ReverseFile {

   public static void main(String[] args) {
      TextReader data;     // Character input stream for reading data.
      PrintWriter result;  // Character output stream for writing data.
      ArrayList<Double> numbers;  // An ArrayList for holding the data.
      numbers = new ArrayList<Double>();

      try {  // Create the input stream.
         data = new TextReader(new FileReader("data.dat"));
      }
      catch (FileNotFoundException e) {
         System.out.println("Can't find file data.dat!");
         return;  // End the program by returning from main().
      }

      try {  // Create the output stream.
         result = new PrintWriter(new FileWriter("result.dat"));
      }
      catch (IOException e) {
         System.out.println("Can't open file result.dat!");
         System.out.println("Error: " + e);
         data.close();  // Close the input file.
         return;        // End the program.
      }
      try {

          // Read numbers from the input file , adding them to the ArrayList.
          while ( data.eof() == false ) {  // Read until end-of-file.
             double inputNumber = data.getlnDouble();
             numbers.add( inputNumber );
          }
          // Output the numbers in reverse order.

          for (int i = numbers.size()-1; i >= 0; i--)
             result.println(numbers.get(i));
          System.out.println("Done!");
       }
       catch (IOException e) {
             // Some problem reading the data from the input file.
          System.out.println("Input Error: " + e.getMessage());
       }
       finally {
             // Finish by closing the files , whatever else may have happened.
          data.close();
          result.close();
       }
   }  // end of main()
} // end of class
```

10.2.2 Files and Directories

The subject of file names is actually more complicated than I've let on so far. To fully specify a file, you have to give both the name of the file and the name of the directory where that file is located. A simple file name like "data.dat" or "result.dat" is taken to refer to a file in a directory that is called the current directory (also known as the "default directory" or "working directory"). The current directory is not a permanent thing. It can be changed by the user or by a program. Files not in the current directory must be referred to by a path name, which includes both the name of the file and information about the directory where it can be found.

To complicate matters even further, there are two types of path names, absolute path names and relative path names. An absolute path name uniquely identifies one file among all the files available to the computer. It contains full information about which directory the file is in and what the file's name is. A relative path name tells the computer how to locate the file starting from the current directory.

It's reasonably safe to say, though, that if you stick to using simple file names only, and if the files are stored in the same directory with the program that will use them, then you will be OK.

It is possible for a Java program to find out the absolute path names for two important directories, the current directory and the user's home directory. The names of these directories are system properties, and they can be read using the method calls:

- `System.getProperty(‘‘user.dir′′)`–returns the absolute path name of the current directory as a `String`.

- `System.getProperty(‘‘user.home′′)`–returns the absolute path name of the user's home directory as a String.

To avoid some of the problems caused by differences in path names between platforms, Java has the class `java.io.File`. An object belonging to this class represents a file. More precisely, an object of type `File` represents a file **name** rather than a file as such. The file to which the name refers might or might not exist. Directories are treated in the same way as files, so a `File` object can represent a directory just as easily as it can represent a file.

A `File` object has a constructor, **new** `File(String)`, that creates a `File` object from a path name. The name can be a simple name, a relative path, or an absolute path. For example, new File("data.dat") creates a `File` object that refers to a file named data.dat, in the current directory. Another constructor has two parameters: **new** `File(File, String)`. The first is a `File` object that refers to the directory that contains the file. The second can be the name of the file or a relative path from the directory to the file.

`File` objects contain several useful instance methods. Assuming that file is a variable of type `File`, here are some of the methods that are available:

- `file.exists()`–This boolean-valued method returns true if the file named by the `File` object already exists. You can use this method if you want to avoid overwriting the contents of an existing file when you create a new `FileWriter`.

- `file.isDirectory()`–This boolean-valued method returns true if the `File` object refers to a directory. It returns false if it refers to a regular file or if no file with the given name exists.

- `file.delete()`–Deletes the file, if it exists. Returns a boolean value to indicate whether the file was successfully deleted.

- `file.list()`–If the `File` object refers to a directory, this method returns an array of type `String[ ]` containing the names of the files in that directory. Otherwise, it returns null.

Here, for example, is a program that will list the names of all the files in a directory specified by the user. Just for fun, I have used a `Scanner` to read the user's input:

```
import java.io.File;
import java.util.Scanner;

/**
 * This program lists the files in a directory specified by
 * the user.  The user is asked to type in a directory name.
 * If the name entered by the user is not a directory, a
 * message is printed and the program ends.
 */


public class DirectoryList {

   public static void main(String[] args) {

      String directoryName;  // Directory name entered by the user.
      File directory;        // File object referring to the directory.
      String[] files;        // Array of file names in the directory.
      Scanner scanner;       // For reading a line of input from the user.

      scanner = new Scanner(System.in);  // scanner reads from standard input.

      System.out.print("Enter a directory name: ");
      directoryName = scanner.nextLine().trim();
      directory = new File(directoryName);

      if (directory.isDirectory() == false) {
          if (directory.exists() == false)
             System.out.println("There is no such directory!");
          else
             System.out.println("That file is not a directory.");
      }
      else {
          files = directory.list();
          System.out.println("Files in directory \"" + directory + "\":");
          for (int i = 0; i < files.length; i++)
             System.out.println("   " + files[i]);
      }

   } // end main()

} // end class DirectoryList
```

All the classes that are used for reading data from files and writing data to files have constructors that take a `File` object as a parameter. For example, if file is a variable of type File, and you want to read character data from that file, you can create a FileReader to do so by saying **new** FileReader(file). If you want to use a TextReader to read from the file, you could say:

```
TextReader data;

try {
   data = new TextReader( new FileReader(file) );
}
catch (FileNotFoundException e) {
   ... // handle the exception
}
```

10.3 Programming With Files

IN THIS SECTION, we look at several programming examples that work with files, using the techniques that were introduced previously.

10.3.1 Copying a File

As a first example, we look at a simple command-line program that can make a copy of a file. Copying a file is a pretty common operation, and every operating system already has a command for doing it. However, it is still instructive to look at a Java program that does the same thing. Many file operations are similar to copying a file, except that the data from the input file is processed in some way before it is written to the output file. All such operations can be done by programs with the same general form.

Since the program should be able to copy any file, we can't assume that the data in the file is in human-readable form. So, we have to use InputStream and OutputStream to operate on the file rather than Reader and Writer. The program simply copies all the data from the InputStream to the OutputStream, one byte at a time. If source is the variable that refers to the InputStream, then the method source.read() can be used to read one byte. This method returns the value -1 when all the bytes in the input file have been read. Similarly, if copy refers to the OutputStream, then copy.write(b) writes one byte to the output file. So, the heart of the program is a simple while loop. As usual, the I/O operations can throw exceptions, so this must be done in a TRY..CATCH statement:

```
while(true) {
   int data = source.read();
   if (data < 0)
      break;
   copy.write(data);
}
```

The file-copy command in an operating system such as UNIX uses command line arguments to specify the names of the files. For example, the user might say "copy original.dat backup.dat" to copy an existing file, original.dat, to a file named backup.dat. Command-line arguments can also be used in Java programs. The command line arguments are stored in the array of strings, args, which is a parameter to the main() method. The program can retrieve the command-line arguments from this array. For example, if the program is named CopyFile and if the user runs the program with the command "`java CopyFile work.dat oldwork.dat`", then in the program, `args[0]` will be the string "`work.dat`" and `args[1]` will be the string "`oldwork.dat`". The value of `args.length` tells the program how many command-line arguments were specified by the user.

My CopyFile program gets the names of the files from the command-line arguments. It prints an error message and exits if the file names are not specified. To add a little interest, there are two ways to use the program. The command line can simply specify the two file names. In that case, if the output file already exists, the program will print an error message and end. This is to make sure that the user won't accidently overwrite an important file. However, if the command line has three arguments, then the first argument must be "-f" while the second and third arguments are file names. The -f is a command-line option, which is meant to modify the behavior of the program. The program interprets the -f to mean that it's OK to overwrite an existing program. (The "f" stands for "force," since it forces the file to be copied in spite of what would otherwise have been considered an error.) You can see in the source code how the command line arguments are interpreted by the program:

```
import java.io.*;
/** Makes a copy of a file. The original file and the name of the
 * copy must be given as command-line arguments. In addition, the
 * first command-line argument can be "-f"; if present, the program
 * will overwrite an existing file; if not, the program will report
 * an error and end if the output file already exists. The number
 * of bytes that are copied is reported. */
public class CopyFile {
   public static void main(String[] args) {

      String sourceName; //Name of the source file,specified on the command line.
      String copyName;     // Name of the copy specified on the command line.
      InputStream source;  // Stream for reading from the source file.
      OutputStream copy;   // Stream for writing the copy.
      boolean force;  // This is set to true if the "-f" option
                      //    is specified on the command line.
      int byteCount;  // Number of bytes copied from the source file.

      /* Get file names from the command line and check for the
         presence of the -f option. If the command line is not one
         of the two possible legal forms, print an error message and
         end this program. */
      if (args.length == 3 && args[0].equalsIgnoreCase("-f")) {
         sourceName = args[1];
         copyName = args[2];
         force = true;
      }
      else if (args.length == 2) {
         sourceName = args[0];
         copyName = args[1];
         force = false;
      }
      else {
         System.out.println("Usage:java CopyFile <source-file> <copy-name>");
         System.out.println("or java CopyFile -f <source-file> <copy-name>");
         return;
      }
      /* Create the input stream.  If an error occurs, end the program. */
      try {
         source = new FileInputStream(sourceName);
      }
      catch (FileNotFoundException e) {
         System.out.println("Can't find file \"" + sourceName + "\".");
         return;
      }
      /* If the output file already exists and the -f option was not
         specified, print an error message and end the program. */
      File file = new File(copyName);
      if (file.exists() && force == false) {
          System.out.println(
               "Output file exists.  Use the -f option to replace it.");
          return;
      }
      /* Create the output stream.  If an error occurs, end the program. */

      try {
         copy = new FileOutputStream(copyName);
      }
      catch (IOException e) {
         System.out.println("Can't open output file \"" + copyName + "\".");
         return;
      }

      /* Copy one byte at a time from the input stream to the output
         stream, ending when the read() method returns -1 (which is
         the signal that the end of the stream has been reached).  If any
         error occurs, print an error message.  Also print a message if
         the file has been copied successfully.  */
      byteCount = 0;
      try {
         while (true) {
            int data = source.read();
            if (data < 0)
               break;
            copy.write(data);
            byteCount++;
         }
         source.close();
         copy.close();
         System.out.println("Successfully copied " + byteCount + " bytes.");
      }
      catch (Exception e) {
         System.out.println("Error occurred while copying.  "
                                   + byteCount + " bytes copied.");
         System.out.println("Error: " + e);
      }
   }  // end main()
} // end class CopyFile
```