Object Oriented Programming with Java

School of Computer Science
University of KwaZulu-Natal

January 30, 2006

Object Oriented Programming with Java

Notes for the Computer Science Module
Object Oriented Programming
COMP200

Adapted from
Introduction to Programming Using Java
Version 4.1, June 2004
by David J. Eck
http://math.hws.edu/javanotes/

School of Computer Science
University of KwaZulu-Natal
Durban
February 2006

Contents

Acknowledgements

Theses notes are, in the most part, taken from David J. Ecks online book INTRODUCTION TO PROGRAMMING USING JAVA, VERSION 4.1, JUNE 2004. We've used chapters 5,6,7, 9, 10, and 12. These notes are released under the same terms and conditions.

Some parts of these notes are based on the notes of Prof Wayne Goddard of Clemson University.

Chapter 1

Design I: Class Level Design

1.1 Program Design

UNDERSTANDING HOW PROGRAMS WORK IS ONE THING. Designing a program to perform some particular task is another thing altogether. **Stepwise refinement** can be used to methodically develop an algorithm.

Stepwise refinement is inherently a top-down process, but the process does have a "bottom," that is, a point at which you stop refining the pseudocode algorithm and translate what you have directly into a programming language. In the absence of subroutines, the process would not bottom out until you get down to the level of assignment statements and very primitive input/output operations. But if you have subroutines lying around to perform certain useful tasks, you can stop refining as soon as you've managed to express your algorithm in terms of those tasks.

This allows you to add a bottom-up element to the top-down approach of stepwise refinement. Given a problem, you might start by writing some subroutines that perform tasks relevant to the problem domain. The subroutines become a toolbox of ready-made tools that you can integrate into your algorithm as you develop it. (Alternatively, you might be able to buy or find a software toolbox written by someone else, containing subroutines that you can use in your project as black boxes.)

Subroutines can also be helpful even in a strict top-down approach. As you refine your algorithm, you are free at any point to take any sub-task in the algorithm and make it into a subroutine. Developing that subroutine then becomes a separate problem, which you can work on separately. Your main algorithm will merely call the subroutine. This, of course, is just a way of breaking your problem down into separate, smaller problems. It is still a top-down approach because the top-down analysis of the problem tells you what subroutines to write. In the bottom-up approach, you start by writing or obtaining subroutines that are relevant to the problem domain, and you build your solution to the problem on top of that foundation of subroutines.

1.1.1 Preconditions and Postconditions

When working with subroutines as building blocks, it is important to be clear about how a subroutine interacts with the rest of the program. This interaction is specified by the contract of the subroutine. A convenient way to express the contract of a subroutine is in terms of preconditions and postconditions.

The precondition of a subroutine is something that must be true when the subroutine is called, if the subroutine is to work correctly. For example, for the built-in function `Math.sqrt(x)`, a precondition is that the parameter, `x`, is greater than or equal to zero, since it is not possible to take the square root of a negative number. In terms of a contract, a precondition represents an obligation of the *caller* of the subroutine. If you call a subroutine without meeting its precondition, then there is no reason to expect it to work properly. The program might crash or give incorrect results, but you can only blame yourself, not the subroutine.

A postcondition of a subroutine represents the other side of the contract. It is something that will be true after the subroutine has run (assuming that its preconditions were met–and that there are no bugs in the subroutine). The postcondition of the function `Math.sqrt()` is that the square of the value that is returned by this function is equal to the parameter that is provided when the subroutine is called. Of course, this will only be true if the preconditiion–that the parameter is greater than or equal to zero–is met. A postcondition of the built-in subroutine `System.out.print()` is that the value of the parameter has been displayed on the screen.

Preconditions most often give restrictions on the acceptable values of parameters, as in the example of `Math.sqrt(x)`. However, they can also refer to global variables that are used in the subroutine. The postcondition of a subroutine specifies the task that it performs. For a function, the postcondition should specify the value that the function returns.

Subroutines are often described by comments that explicitly specify their preconditions and postconditions. When you are given a pre-written subroutine, a statement of its preconditions and postcondtions tells you how to use it and what it does. When you are assigned to write a subroutine, the preconditions and postconditions give you an exact specification of what the subroutine is expected to do. I will use this approach in the example that constitutes the rest of this section. I will also use it occasionally later in the book, although I will generally be less formal in my commenting style.

1.1.2 A Design Example

Let's work through an example of program design using subroutines. In this example, we will both use prewritten subroutines as building blocks and design new subroutines that we need to complete the project.

Suppose that we have an already-written class called `Mosaic`. This class allows a program to work with a window that displays little colored rectangles arranged in rows and columns. The window can be opened, closed, and otherwise manipulated with static member subroutines defined in the `Mosaic` class. Here are some of the available routines:

```
void Mosaic.open(int rows, int cols, int w, int h);
   Precondition: The parameters rows, cols, w, and h are greater
      than zero.
   Postcondition: A "mosaic" window is opened on the screen that can
      display rows and columns of colored rectangles.  Each rectangle
      is w pixels wide and h pixels high.  The number of rows is given
      by the first parameter and the number of columns by the second.
      Initially, all the rectangles are black.  Note: The rows are
      numbered from 0 to rows - 1, and the columns are numbered from
      0 to cols - 1.
```

```
void Mosaic.setColor(int row, int col, int r, int g, int b);
   Precondition: row and col are in the valid ranges of row numbers and
      column numbers.  r, g, and b are in the range 0 to 255.  Also,
      the mosaic window should be open.
   Postcondition: The color of the rectangle in row number row and
      column number col has been set to the color specified by r, g & b.
      r gives the amount of red in the color with 0 representing no red
      and 255 representing the maximum possible amount of red.  The
      larger the value of r, the more red in the color.  g and b work
      similarly for the green and blue color components.

int Mosaic.getRed(int row, int col);
int Mosaic.getBlue(int row, int col);

int Mosaic.getGreen(int row, int col);
   Precondition: row and col are in the valid ranges of row numbers and
      column numbers.  Also, the mosaic window should be open.
   Postcondition: Returns an int value that represents one of the three
      color components of the rectangle in row number row and column
      number col.  The return value is in the range 0 to 255.
      (Mosaic.getRed() returns the red component of the rectangle,
      Mosaic.getGreen() the green component, and Mosaic.getBlue() the
      blue component.)

void Mosaic.delay(int milliseconds);
   Precondition: milliseconds is a positive integer.
   Postcondition: The program has paused for at least the number of
      milliseconds given by the parameter, where one second is equal to
      1000 milliseconds.  Note: This can be used to insert a time delay
      in the program (to regulate the speed at which the colors are
      changed, for example).

boolean Mosaic.isOpen();
Precondition: None.
Postcondition: The return value is true if the mosaic window is open on
   the screen, and is false otherwise.  Note: The window will be closed
   if the user clicks its close box.  It can also be closed
   programmatically by calling the subroutine Mosaic.close().
```

My idea is to use the `Mosaic` class as the basis for a neat animation. I want to fill the window with randomly colored squares, and then randomly change the colors in a loop that continues as long as the window is open. "Randomly change the colors" could mean a lot of different things, but after thinking for a while, I decide it would be interesting to have a "disturbance" that wanders randomly around the window, changing the color of each square that it encounters. Here's an applet that shows what the program will do:

Applet Reference: see (Applet "RandomMosaicWalkApplet")

Figure 1.1: Snapshot of the RandomMosaicWalkApplet

With basic routines for manipulating the window as a foundation, I can turn to the specific problem at hand. A basic outline for my program is

```
Open a Mosaic window Fill window with random colors; Move around,
changing squares at random.
```

Filling the window with random colors seems like a nice coherent task that I can work on separately, so let's decide to write a separate subroutine to do it. The third step can be expanded a bit more, into the steps: Start in the middle of the window, then keep moving to a new square and changing the color of that square. This should continue as long as the mosaic window is still open. Thus we can refine the algorithm to:

```
Open a Mosaic window
Fill window with random colors;
Set the current position to the middle square in the window;
As long as the mosaic window is open:
     Randomly change color of current square;
     Move current position up, down, left, or right, at random;
```

I need to represent the current position in some way. That can be done with two `int` variables named `currentRow` and `currentColumn`. I'll use 10 rows and 20 columns of squares in my mosaic, so setting the current position to be in the center means setting `currentRow` to 5 and `currentColumn` to 10. I already have a subroutine, `Mosaic.open()`, to open the window, and I have a function, `Mosaic.isOpen()`, to test whether the window is open. To keep the main routine simple, I decide that I will write two more subroutines of my own to carry out the two tasks in the while loop. The algorithm can then be written in Java as:

```
Mosaic.open(10,20,10,10);
fillWithRandomColors();
currentRow = 5; // Middle row, halfway down the window.
currentColumn = 10; // Middle column.
while ( Mosaic.isOpen() ) {
     changeToRandomColor(currentRow,
     currentColumn);
     randomMove();
}
```

With the proper wrapper, this is essentially the `main()` routine of my program. It turns out I have to make one small modification: To prevent the animation from running too fast, the line "`Mosaic.delay(20);` " is added to the `while` loop.

The `main()` routine is taken care of, but to complete the program, I still have to write the subroutines `fillWithRandomColors()`, `changeToRandomColor(int,int)`, and `randomMove()`. Writing each of these subroutines is a separate, small task.

The `fillWithRandomColors()` routine is defined by the postcondition that "each of the rectangles in the mosaic has been changed to a random color." Pseudocode for an algorithm to accomplish this task can be given as:

```
For each row:
   For each column:
      set the square in that row and column to a random color
```

"For each row" and "for each column" can be implemented as for loops. We've already planned to write a subroutine `changeToRandomColor` that can be used to set the color. (The possibility of reusing subroutines in several places is one of the big payoffs of using them!) So, `fillWithRandomColors()` can be written in Java as:

```
static void fillWithRandomColors() {
   for (int row = 0; row < 10; row++)
       for (int column = 0; column < 20; column++)
            changeToRandomColor(row,column); }
```

Turning to the `changeToRandomColor` subroutine, we already have a method, `Mosaic.setColor(row,col,r,g,b)`, that can be used to change the color of a square. If we want a random color, we just have to choose random values for `r`, `g`, and `b`. According to the precondition of the `Mosaic.setColor()` subroutine, these random values must be integers in the range from 0 to 255. A formula for randomly selecting such an integer is "`(int)(256*Math.random())` ". So the random color subroutine becomes:

```
static void changeToRandomColor(int rowNum, int colNum) {
     int red = (int)(256*Math.random());
     int green = (int)(256*Math.random());
     int blue = (int)(256*Math.random());
     mosaic.setColor(rowNum,colNum,red,green,blue);
}
```

Finally, consider the `randomMove` subroutine, which is supposed to randomly move the disturbance up, down, left, or right. To make a random choice among four directions, we can choose a random integer in the range 0 to 3. If the integer is 0, move in one direction; if it is 1, move in another direction; and so on. The position of the disturbance is given by the variables `currentRow` and `currentColumn`. To "move up" means to subtract 1 from `currentRow`. This leaves open the question of what to do if `currentRow` becomes -1, which would put the disturbance above the window. Rather than let this happen, I decide to move the disturbance to the opposite edge of the applet by setting `currentRow` to 9. (Remember that the 10 rows are numbered from 0 to 9.) Moving the disturbance

down, left, or right is handled similarly. See the code below for `randomMove`. It uses a `switch` statement to decide which direction to move, the code becomes.

Putting this all together, we get the following complete program. The variables `currentRow` and `currentColumn` are defined as static members of the class, rather than local variables, because each of them is used in several different subroutines. This program actually depends on two other classes, `Mosaic` and another class called `MosaicCanvas` that is used by `Mosaic`. If you want to compile and run this program, both of these classes must be available to the program.

```
public class RandomMosaicWalk {
   /* This program shows a window full of randomly colored
      squares.  A "disturbance" moves randomly around
      in the window, randomly changing the color of
      each square that it visits.  The program runs
      until the user closes the window.
   */

   static int currentRow; // row currently containing the disturbance
   static int currentColumn; // column currently containing disturbance

   public static void main(String[] args) {
          // Main program creates the window, fills it with
          // random colors, then moves the disturbance in
          // a random walk around the window for as long as
          // the window is open.
       Mosaic.open(10,20,10,10);
       fillWithRandomColors();
       currentRow = 5;   // start at center of window
       currentColumn = 10;
       while (Mosaic.isOpen()) {
           changeToRandomColor(currentRow, currentColumn);
           randomMove();
           Mosaic.delay(20);
       }
   }  // end of main()

   static void fillWithRandomColors() {
           // Precondition:  The mosaic window is open.
           // Postcondition: Each rectangles have been set to a
           //                random color
        for (int row=0; row < 10; row++) {
           for (int column=0; column < 20; column++) {
               changeToRandomColor(row, column);
           }
        }
   }  // end of fillWithRandomColors()
```

```
   static void changeToRandomColor(int rowNum, int colNum) {
          // Precondition:  rowNum and colNum are in the valid range
          //                of row and column numbers.
          // Postcondition: The rectangle in the specified row and
          //                column has been changed to a random color.
       int red = (int)(256*Math.random());    // choose random levels in range
       int green = (int)(256*Math.random());  // 0 to 255 for red, green,
       int blue = (int)(256*Math.random());   // and blue color components
       Mosaic.setColor(rowNum,colNum,red,green,blue);
    }  // end of changeToRandomColor()

    static void randomMove() {
          // Precondition:  The global variables currentRow and currentColumn
          //                specify a valid position in the grid.
          // Postcondition: currentRow or currentColumn is changed to
          //                one of the neighboring positions in the grid,
          //                up, down, left, or right from the previous
          //                position.  (If this moves the position outside
          //                the grid, then it is moved to the opposite edge
          //                of the window.
       int directionNum; // Randomly set to 0, 1, 2, or 3 to choose direction.
       directionNum = (int)(4*Math.random());
       switch (directionNum) {
          case 0:  // move up
             currentRow--;
             if (currentRow < 0)
                currentRow = 9;
             break;
          case 1:  // move right
             currentColumn++;
             if (currentColumn >= 20)
                currentColumn = 0;
             break;
          case 2:  // move down
             currentRow ++;
             if (currentRow >= 10)
                currentRow = 0;
             break;
          case 3:
             currentColumn--;
             if (currentColumn < 0)
                currentColumn = 19;
             break;
       }
    }  // end of randomMove()
} // end of class RandomMosaicWalk
```

1.2 Toolboxes, API's, and Packages

AS COMPUTERS AND THEIR USER INTERFACES have become easier to use, they have also become more complex for programmers to deal with. You can write programs for a simple console-style user interface using just a few subroutines that write output to the console and read the user's typed replies. A modern graphical user interface, with windows, buttons, scroll bars, menus, text-input boxes, and so on, might make things easier for the user. But it forces the programmer to cope with a hugely expanded array of possibilities. The programmer sees this increased complexity in the form of great numbers of subroutines that are provided for managing the user interface, as well as for other purposes.

Someone who wants to program for Macintosh computers – and to produce programs that look and behave the way users expect them to – must deal with the Macintosh Toolbox, a collection of well over a thousand different subroutines. There are routines for opening and closing windows, for drawing geometric figures and text to windows, for adding buttons to windows, and for responding to mouse clicks on the window. There are other routines for creating menus and for reacting to user selections from menus. Aside from the user interface, there are routines for opening files and reading data from them, for communicating over a network, for sending output to a printer, for handling communication between programs, and in general for doing all the standard things that a computer has to do. Windows 98 and Windows 2000 provide their own sets of subroutines for programmers to use, and they are quite a bit different from the subroutines used on the Mac.

The analogy of a "toolbox" is a good one to keep in mind. Every programming project involves a mixture of innovation and reuse of existing tools. A programmer is given a set of tools to work with, starting with the set of basic tools that are built into the language: things like variables, assignment statements, if statements, and loops. To these, the programmer can add existing toolboxes full of routines that have already been written for performing certain tasks. These tools, if they are well-designed, can be used as true black boxes: They can be called to perform their assigned tasks without worrying about the particular steps they go through to accomplish those tasks. The innovative part of programming is to take all these tools and apply them to some particular project or problem (word-processing, keeping track of bank accounts, processing image data from a space probe, Web browsing, computer games,...). This is called applications programming.

A software toolbox is a kind of black box, and it presents a certain interface to the programmer. This interface is a specification of what routines are in the toolbox, what parameters they use, and what tasks they perform. This information constitutes the API, or Applications Programming Interface, associated with the toolbox. The Macintosh API is a specification of all the routines available in the Macintosh Toolbox. A company that makes some hardware device – say a card for connecting a computer to a network – might publish an API for that device consisting of a list of routines that programmers can call in order to communicate with and control the device. Scientists who write a set of routines for doing some kind of complex computation – such as solving "differential equations", say – would provide an API to allow others to use those routines without understanding the details of the computations they perform.

The Java programming language is supplemented by a large, standard API. You've seen part of this API already, in the form of mathematical subroutines such as `Math.sqrt()`, the `String` data type and its associated routines, and the `System.out.print()` routines. The standard Java API includes routines for working with graphical user interfaces, for

network communication, for reading and writing files, and more. It's tempting to think of these routines as being built into the Java language, but they are technically subroutines that have been written and made available for use in Java programs.

Java is platform-independent. That is, the same program can run on platforms as diverse as Macintosh, Windows, UNIX, and others. The same Java API must work on all these platforms. But notice that it is the **interface** that is platform-independent; the **implementation** varies from one platform to another. A Java system on a particular computer includes implementations of all the standard API routines. A Java program includes only **calls** to those routines. When the Java interpreter executes a program and encounters a call to one of the standard routines, it will pull up and execute the implementation of that routine which is appropriate for the particular platform on which it is running. This is a very powerful idea. It means that you only need to learn one API to program for a wide variety of platforms.

Like all subroutines in Java, the routines in the standard API are grouped into classes. To provide larger-scale organization, classes in Java can be grouped into packages. You can have even higher levels of grouping, since packages can also contain other packages. In fact, the entire standard Java API is implemented in several packages. One of these, which is named "`java`", contains the non-GUI packages as well as the original AWT graphics user interface classes. Another package, "`javax`", was added in Java version 1.2 and contains the classes used by the Swing graphical user interface.

A package can contain both classes and other packages. A package that is contained in another package is sometimes called a "sub-package." Both the `java` package and the `javax` package contain sub-packages. One of the sub-packages of `java`, for example, is called "`awt`". Since `awt` is contained within `java`, its full name is actually `java.awt`. This is the package that contains classes related to the AWT graphical user interface, such as a `Button` class which represents push-buttons on the screen and the `Graphics` class which provides routines for drawing on the screen. Since these classes are contained in the package `java.awt`, their full names are actually `java.awt.Button` and `java.awt.Graphics`. (I hope that by now you've gotten the hang of how this naming thing works in Java.) Similarly, `javax` contains a sub-package named `javax.swing`, which includes such classes as `javax.swing.JButton` and `javax.swing.JApplet`.

The `java` package includes several other sub-packages, such as `java.io`, which provides facilities for input/output, `java.net`, which deals with network communication, and `java.applet`, which implements the basic functionality of applets. The most basic package is called `java.lang`. This package contains fundamental classes such as `String` and `Math`.

It might be helpful to look at a graphical representation of the levels of nesting in the `java` package, its sub-packages, the classes in those sub-packages, and the subroutines in those classes. This is not a complete picture, since it shows only a few of the many items in each element:

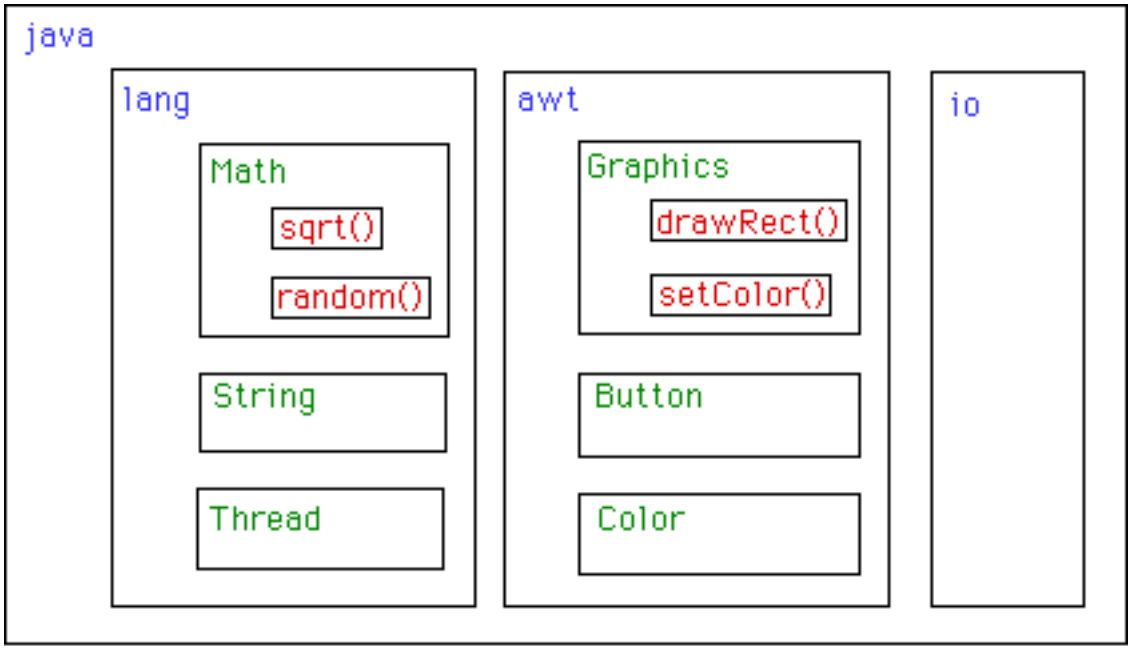
Let's say that you want to use the class `java.awt.Color` in a program that you are writing. One way to do this is to use the full name of the class. For example, you could say

```
java.awt.Color rectColor;
```

to declare a variable named `rectColor` whose type is `java.awt.Color`. Of course, this can get tiresome, so Java makes it possible to avoid using the full names of classes. If you put

```
import java.awt.Color;
```

at the beginning of a Java source code file, then, in the rest of the file, you can abbrevi-



ate the full name `java.awt.Color` to just the name of the class, `Color`. This would allow you to say just

```
Color rectColor;
```

to declare the variable `rectColor`. (The only effect of the `import` statement is to allow you to use simple class names instead of full "package.class" names; you aren't really importing anything substantial. If you leave out the `import` statement, you can still access the class – you just have to use its full name.) There is a shortcut for importing all the classes from a given package. You can import all the classes from `java.awt` by saying

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

and you can import all the classes from `javax.swing` with the line

```
import javax.swing.*;
```

In fact, any Java program that uses a graphical user interface is likely to begin with one or both of these lines. A program might also include lines such as "`import java.net.*;` " or "`import java.io.*;` " to get easy access to networking and input/output classes. (When you start importing lots of packages in this way, you have to be careful about one thing: It's possible for two classes that are in different packages to have the same name. For example, both the `java.awt` package and the `java.util` package contain classes named `List`. If you import both `java.awt.*` and `java.util.*`, the simple name `List` will be ambiguous. If you try to declare a variable of type `List`, you will get a compiler error message about an ambiguous class name. The solution is simple: use the full name of the class, either `java.awt.List` or `java.util.List`. Another solution is to use `import` to import the individual classes you need, instead of importing entire packages.)

Because the package `java.lang` is so fundamental, all the classes in `java.lang` are **automatically** imported into every program. It's as if every program began with the statement "`import java.lang.*;` ". This is why we have been able to use the class name `String` instead of `java.lang.String`, and `Math.sqrt()` instead of `java.lang.Math.sqrt()`.

It would still, however, be perfectly legal to use the longer forms of the names.

Programmers can create new packages. Suppose that you want some classes that you are writing to be in a package named `utilities`. Then the source code file that defines those classes must begin with the line "`package utilities;` ". Any program that uses the classes should include the directive "`import utilities.*;` " to obtain access to all the classes in the `utilities` package. Unfortunately, things are a little more complicated than this. Remember that if a program uses a class, then the class must be "available" when the program is compiled and when it is executed. Exactly what this means depends on which Java environment you are using. Most commonly, classes in a package named `utilities` should be in a directory with the name `utilities`, and that directory should be located in the same place as the program that uses the classes.

In projects that define large numbers of classes, it makes sense to organize those classes into one or more packages. It also makes sense for programmers to create new packages as toolboxes that provide functionality and API's for dealing with areas not covered in the standard Java API. (And in fact such "toolmaking" programmers often have more prestige than the applications programmers who use their tools.)

You need to know about packages mainly so that you will be able to import the standard packages. These packages are always available to the programs that you write. You might wonder where the standard classes are actually located. Again, that depends to some extent on the version of Java that you are using. But they are likely to be collected together into a large file named `rt.jar` or `classes.zip`, which is located in some place where the Java compiler and the Java interpreter will know to look for it.

Every class is actually part of a package. If a class is not specifically placed in a package, then it is put in something called the default package, which has no name. All the examples that you see in these notes are in the default package.

1.3 Javadoc

Good programming means extensive comments and documentation. At the very least, explain the function of each instance variable, and for each method explain its purpose, parameters, returns, where applicable. You should also strive for a consistent layout and for expressive variable names.

A program that is well-documented is much more valuable than the same program without the documentation. Java comes with a tool called javadoc that can make it easier to produce the documentation is a readable and organized format. **JavaDoc** is a program that will automatically extract/generate an HTML help-page from code that is properly commented. In particular, it is designed produce a help file that, for a class, lists the methods, constructors and public fields, and for each method explains what it does together with pre-conditions, post-conditions, the meaning of the parameters, exceptions that may be thrown and other things.

Javadoc is especially useful for documenting classes and packages of classes that are meant to be used by other programmers. A programmer who wants to use pre-written classes shouldn't need to search through the source code to find out how to use them. If the documentation in the source code is in the correct format, javadoc can separate out the documentation and make it into a set of web pages. The web pages are automatically formatted and linked into an easily browseable Web site. Sun Microsystem's on-line documentation for the standard Java API was produced using javadoc.

Javadoc is actually very easy to use. In a source code file, javadoc documentation looks like a regular multi-line comment, except that it begins with "/**" instead of with

"/*". Each such comment is associated with some class, member variable, or method. The documentation for each item must be placed in a comment that precedes the item. (This is how javadoc knows which item the comment is for.) The comments can include certain special notations. For example, the notation "@return" is used to begin the description of the return value of a function. And "@param ¡parameter-name¿" marks the beginning of the description of a parameter of a method. For example, here is a short utility method with a javadoc comment:

```
/**
  * Return the real number represented by the string s,
  * or return Double.NaN if s does not represent a legal
  * real number.
  * @param s String to interpret as real number.
  * @return the real number represented by s.
*/
public static double stringToReal(String s) {
     try {
           return Double.parseDouble(s);
     }
     catch (NumberFormatException e){
           return Double.NaN;
     }
 }
```

Sun's Java Software Development Kit includes javadoc as a program that can be used on the command line. This program takes one or more Java source code files, extracts the javadoc comments from them, and prepares Web pages containing the documentation. Integrated development environments for Java (such as the Eclipse IDE) typically include a menu command for generating javadoc documentation.

Chapter 2

Object Based Programming

WHEREAS A SUBROUTINE represents a single task, an object can encapsulate both data (in the form of instance variables) and a number of different tasks or "behaviors" related to that data (in the form of instance methods). Therefore objects provide another, more sophisticated type of structure that can be used to help manage the complexity of large programs.

2.1 Objects, Instance Methods, and Instance Variables

OBJECT-ORIENTED PROGRAMMING (OOP) represents an attempt to make programs more closely model the way people think about and deal with the world. In the older styles of programming, a programmer who is faced with some problem must identify a computing task that needs to be performed in order to solve the problem. Programming then consists of finding a sequence of instructions that will accomplish that task. But at the heart of object-oriented programming, instead of tasks we find objects – entities that have behaviors, that hold information, and that can interact with one another. Programming consists of designing a set of objects that somehow model the problem at hand. Software objects in the program can represent real or abstract entities in the problem domain. This is supposed to make the design of the program more natural and hence easier to get right and easier to understand.

To some extent, OOP is just a change in point of view. We can think of an object in standard programming terms as nothing more than a set of variables together with some subroutines for manipulating those variables. In fact, it is possible to use object-oriented techniques in any programming language. However, there is a big difference between a language that makes OOP possible and one that actively supports it. An object-oriented programming language such as Java includes a number of features that make it very different from a standard language. In order to make effective use of those features, you have to "orient" your thinking correctly.

Objects are closely related to classes. We have already been working with classes for several chapters, and we have seen that a class can contain variables and subroutines. If an object is also a collection of variables and subroutines, how do they differ from classes? And why does it require a different type of thinking to understand and use them effectively?

I have said that classes "describe" objects, or more exactly that the non-static portions of classes describe objects. But it's probably not very clear what this means. The more usual terminology is to say that objects belong to classes, but this might not be much clearer. (There is a real shortage of English words to properly distinguish all the concepts

involved. An object certainly doesn't "belong" to a class in the same way that a member variable "belongs" to a class.) From the point of view of programming, it is more exact to say that classes are used to create objects. A class is a kind of factory for constructing objects. The non-static parts of the class specify, or describe, what variables and subroutines the objects will contain. This is part of the explanation of how objects differ from classes: Objects are created and destroyed as the program runs, and there can be many objects with the same structure, if they are created using the same class.

Consider a simple class whose job is to group together a few static member variables. For example, the following class could be used to store information about the person who is using the program:

```
class UserData {
    static String name;
    static int age;
}
```

In a program that uses this class, there is only one copy of each of the variables `UserData.name` and `UserData.age`. There can only be one "user," since we only have memory space to store data about one user. The class, `UserData`, and the variables it contains exist as long as the program runs. Now, consider a similar class that includes non-static variables:

```
class PlayerData {
   String name;
   int age;
}
```

In this case, there is no such variable as `PlayerData.name` or `PlayerData.age`, since `name` and `age` are not static members of `PlayerData`. So, there is nothing much in the class at all – except the potential to create objects. But, it's a lot of potential, since it can be used to create any number of objects! Each object will have its **own** variables called `name` and `age`. There can be many "players" because we can make new objects to represent new players on demand. A program might use this class to store information about multiple players in a game. Each player has a name and an age. When a player joins the game, a new `PlayerData` object can be created to represent that player. If a player leaves the game, the `PlayerData` object that represents that player can be destroyed. A system of objects in the program is being used to dynamically model what is happening in the game. You can't do this with "static" variables!

An object that belongs to a class is said to be an instance of that class. The variables that the object contains are called instance variables. The subroutines that the object contains are called instance methods. (Recall that in the context of object-oriented programming, "method" is a synonym for "subroutine". From now on, for subroutines in objects, I will prefer the term "method.") For example, if the `PlayerData` class, as defined above, is used to create an object, then that object is an instance of the `PlayerData` class, and `name` and `age` are instance variables in the object. It is important to remember that the class of an object determines the **types** of the instance variables; however, the actual data is contained inside the individual objects, not the class. Thus, each object has its own set of data.

An applet that scrolls a message across a Web page might include a subroutine named `scroll()`. Since the applet is an object, this subroutine is an instance method of the applet. The source code for the method is in the class that is used to create the applet.

Still, it's better to think of the instance method as belonging to the object, not to the class. The non-static subroutines in the class merely specify the instance methods that every object created from the class will contain. The `scroll()` methods in two different applets do the same thing in the sense that they both scroll messages across the screen. But there is a real difference between the two `scroll()` methods. The messages that they scroll can be different. (You might say that the subroutine definition in the class specifies what type of behavior the objects will have, but the specific behavior can vary from object to object, depending on the values of their instance variables.)

As you can see, the static and the non-static portions of a class are very different things and serve very different purposes. Many classes contain only static members, or only non-static. However, it is possible to mix static and non-static members in a single class, and we'll see a few examples later in this chapter where it is reasonable to do so. By the way, static member variables and static member subroutines in a class are sometimes called class variables and class methods, since they belong to the class itself, rather than to instances of that class. This terminology is most useful when the class contains both static and non-static members.

So far, I've been talking mostly in generalities, and I haven't given you much idea what you have to put in a program if you want to work with objects. Let's look at a specific example to see how it works. Consider this extremely simplified version of a `Student` class, which could be used to store information about students taking a course:

```
class Student {

   String name;  // Student's name.
   double test1, test2, test3;   // Grades on three tests.

   double getAverage() {  // compute average test grade
      return (test1 + test2 + test3) / 3;
   }
}  // end of class Student
```

None of the members of this class are declared to be `static`, so the class exists only for creating objects. This class definition says that any object that is an instance of the `Student` class will include instance variables named `name`, `test1`, `test2`, and `test3`, and it will include an instance method named `getAverage()`. The names and tests in different objects will generally have different values. When called for a particular student, the method `getAverage()` will compute an average using **that student's** test grades. Different students can have different averages. (Again, this is what it means to say that an instance method belongs to an individual object, not to the class.)

In Java, a class is a **type**, similar to the built-in types such as `int` and `boolean`. So, a class name can be used to specify the type of a variable in a declaration statement, the type of a formal parameter, or the return type of a function. For example, a program could define a variable named `std` of type `Student` with the statement

```
Student std;
```

However, declaring a variable does **not** create an object! This is an important point, which is related to this Very Important Fact:

In Java, no variable can ever hold an object.
A variable can only hold a reference to an object.

You should think of objects as floating around independently in the computer's memory. In fact, there is a special portion of memory called the heap where objects live. Instead

of holding an object itself, a variable holds the information necessary to find the object in memory. This information is called a reference or pointer to the object. In effect, a reference to an object is the address of the memory location where the object is stored. When you use a variable of class type, the computer uses the reference in the variable to find the actual object.

Objects are actually created by an operator called `new`, which creates an object and returns a reference to that object. For example, assuming that `std` is a variable of type `Student`, declared as above, the assignment statement

```
std = new Student();
```

would create a new object which is an instance of the class `Student`, and it would store a reference to that object in the variable `std`. The value of the variable is a reference to the object, not the object itself. It is not quite true, then, to say that the object is the "value of the variable `std` " (though sometimes it is hard to avoid using this terminology). It is certainly **not at all true** to say that the object is "stored in the variable `std`." The proper terminology is that "the variable `std` refers to the object," and I will try to stick to that terminology as much as possible.

So, suppose that the variable `std` refers to an object belonging to the class `Student`. That object has instance variables `name` , `test1`, `test2`, and `test3`. These instance variables can be referred to as `std.name`, `std.test1`, `std.test2`, and `std.test3`. This follows the usual naming convention that when `B` is part of `A`, then the full name of `B` is `A.B`. For example, a program might include the lines

```
System.out.println("Hello, "  +  std.name
                     + ".  Your test grades are:");
System.out.println(std.test1);
System.out.println(std.test2);
System.out.println(std.test3);
```

This would output the name and test grades from the object to which `std` refers. Similarly, `std` can be used to call the `getAverage()` instance method in the object by saying `std.getAverage()`. To print out the student's average, you could say:

```
System.out.println( "Your average is " + std.getAverage() );
```

More generally, you could use `std.name` any place where a variable of type `String` is legal. You can use it in expressions. You can assign a value to it. You can even use it to call subroutines from the `String` class. For example, `std.name.length()` is the number of characters in the student's name.

It is possible for a variable like `std`, whose type is given by a class, to refer to no object at all. We say in this case that `std` holds a null reference. The null reference is written in Java as "`null` ". You can store a null reference in the variable `std` by saying

```
std = null;
```

and you could test whether the value of `std` is null by testing

```
 if (std == null) .  .  .
```

If the value of a variable is `null`, then it is, of course, illegal to refer to instance variables or instance methods through that variable – since there is no object, and hence no instance variables to refer to. For example, if the value of the variable `std` is `null`, then it would be illegal to refer to `std.test1`. If your program attempts to use a null reference illegally like this, the result is an error called a null pointer exception.

Let's look at a sequence of statements that work with objects:

```
Student std, std1,       // Declare four variables of
          std2, std3;    //   type Student.
std = new Student();     // Create a new object belonging
                         //   to the class Student, and
                         //   store a reference to that
                         //   object in the variable std.
std1 = new Student();    // Create a second Student object
                         //   and store a reference to
                         //   it in the variable std1.
std2 = std1;             // Copy the reference value in std1
                         //   into the variable std2.
std3 = null;             // Store a null reference in the
                         //   variable std3.

std.name = "John Smith";  // Set values of some instance variables.
std1.name = "Mary Jones";

     // (Other instance variables have default
     //    initial values of zero.)
```

After the computer executes these statements, the situation in the computer's memory looks like this:

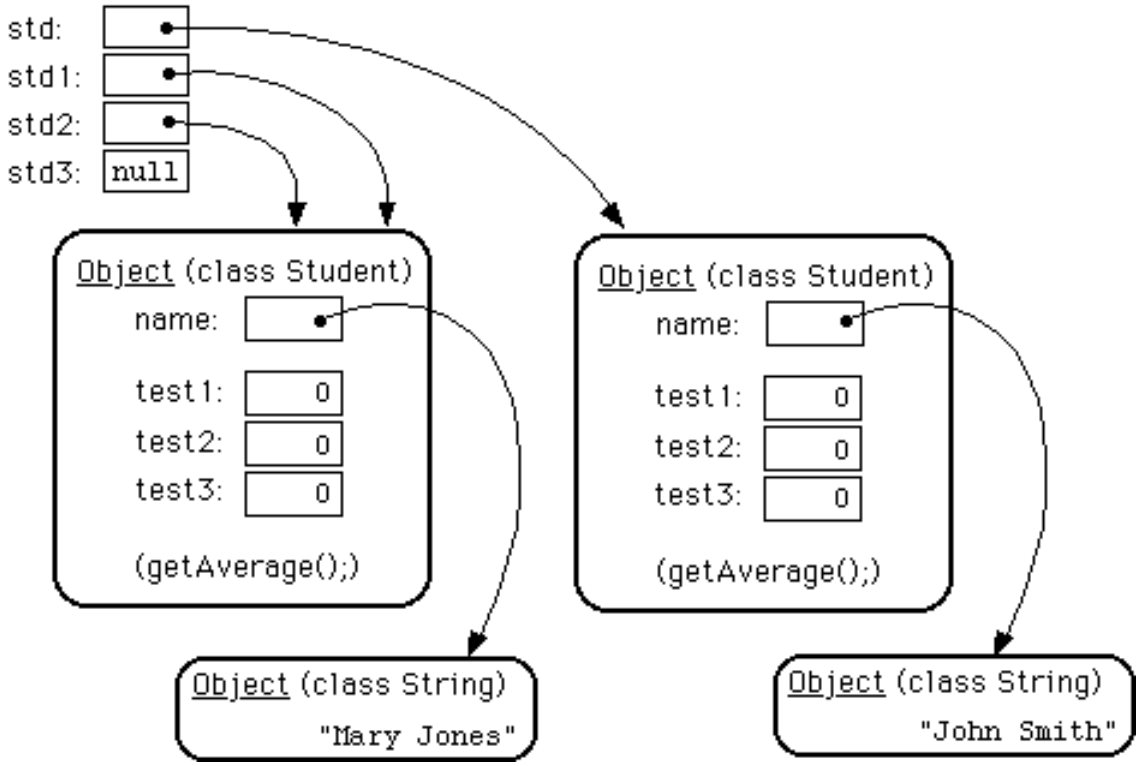


Figure 2.1: Objects in Memory

This picture shows variables as little boxes, labeled with the names of the variables. Objects are shown as boxes with round corners. When a variable contains a reference

to an object, the value of that variable is shown as an arrow pointing to the object. The variable `std3`, with a value of `null`, doesn't point anywhere. The arrows from `std1` and `std2` both point to the same object. This illustrates a Very Important Point:

When one object variable is assigned to another, only a reference is copied. The object referred to is not copied.

When the assignment "`std2 = std1 ;`" was executed, no new object was created. Instead, `std2` was set to refer to the very same object that `std1` refers to. This has some consequences that might be surprising. For example, `std1.name` and `std2.name` refer to exactly the same variable, namely the instance variable in the object that both `std1` and `std2` refer to. After the string `''Mary Jones''` is assigned to the variable `std1.name`, it is also be true that the value of `std2.name` is `''Mary Jones''`. There is a potential for a lot of confusion here, but you can help protect yourself from it if you keep telling yourself, "The object is not in the variable. The variable just holds a pointer to the object."

You can test objects for equality and inequality using the operators == and !=, but here again, the semantics are different from what you are used to. When you make a test "`if (std1 == std2)` ", you are testing whether the values stored in `std1` and `std2` are the same. But the values are references to objects, not objects. So, you are testing whether `std1` and `std2` refer to the same object, that is, whether they point to the same location in memory. This is fine, if its what you want to do. But sometimes, what you want to check is whether the instance variables in the objects have the same values. To do that, you would need to ask whether "`std1.test1 == std2.test1 && std1.test2 == std2.test2 && std1.test3 == std2.test3 && std1.name.equals(std2.name)` "

I've remarked previously that `Strings` are objects, and I've shown the strings `''Mary Jones''` and `''John Smith''` as objects in the above illustration. A variable of type `String` can only hold a reference to a string, not the string itself. It could also hold the value `null`, meaning that it does not refer to any string at all. This explains why using the == operator to test strings for equality is not a good idea. Suppose that `greeting` is a variable of type `String`, and that the string it refers to is `''Hello''`. Then would the test `greeting == ''Hello''` be true? Well, maybe, maybe not. The variable `greeting` and the `String` literal `''Hello''` each refer to a string that contains the characters H-e-l-l-o. But the strings could still be different objects, that just happen to contain the same characters. The function `greeting.equals(''Hello'')` tests whether `greeting` and `''Hello''` contain the same characters, which is almost certainly the question you want to ask. The expression `greeting == ''Hello''` tests whether `greeting` and `''Hello''` contain the same characters **stored in the same memory location**.

The fact that variables hold references to objects, not objects themselves, has a couple of other consequences that you should be aware of. They follow logically, if you just keep in mind the basic fact that the object is not stored in the variable. The object is somewhere else; the variable points to it.

Suppose that a variable that refers to an object is declared to be `final`. This means that the value stored in the variable can never be changed, once the variable has been initialized. The value stored in the variable is a reference to the object. So the variable will continue to refer to the same object as long as the variable exists. However, this does not prevent the data **in the object** from changing. The variable is `final`, not the object. It's perfectly legal to say

```
final Student stu = new Student();
stu.name = "John Doe";  // Change data in the object;
                        // The value stored in stu is not changed.
```

Next, suppose that `obj` is a variable that refers to an object. Let's consider what happens when `obj` is passed as an actual parameter to a subroutine. The value of `obj` is assigned to a formal parameter in the subroutine, and the subroutine is executed. The subroutine has no power to change the value stored in the variable, `obj`. It only has a copy of that value. However, that value is a reference to an object. Since the subroutine has a reference to the object, it can change the data stored in the object. After the subroutine ends, `obj` still points to the same object, but the data stored **in the object** might have changed. Suppose `x` is a variable of type `int` and `stu` is a variable of type `Student`. Compare:

```
void dontChange(int z) {
    z = 42;
}

The lines:

   x = 17;
   dontChange(x);
   System.out.println(x);

output the value 17.

The value of x is not
changed by the subroutine,
which is equivalent to

   z = x;
   z = 42;
```

```
void change(Student s) {
     s.name = "Fred";
}

The lines:

   stu.name = "Jane";
   change(stu);
   System.out.println(stu.name);

output the value "Fred".

The value of stu is not
changed, but stu.name is.
This is equivalent to

   s = stu;
   s.name = "Fred";
```

2.2 Constructors and Object Initialization

OBJECT TYPES IN JAVA ARE VERY DIFFERENT from the primitive types. Simply declaring a variable whose type is given as a class does not automatically create an object of that class. Objects must be explicitly constructed. For the computer, the process of constructing an object means, first, finding some unused memory in the heap that can be used to hold the object and, second, filling in the object's instance variables. As a programmer, you don't care where in memory the object is stored, but you will usually want to exercise some control over what initial values are stored in a new object's instance variables. In many cases, you will also want to do more complicated initialization or bookkeeping every time an object is created.

An instance variable can be assigned an initial value in its declaration, just like any other variable. For example, consider a class named `PairOfDice`. An object of this class will represent a pair of dice. It will contain two instance variables to represent the numbers showing on the dice and an instance method for rolling the dice:

```
public class PairOfDice {
    public int die1 = 3;   // Number showing on the first die.
    public int die2 = 4;   // Number showing on the second die.

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
         die1 = (int)(Math.random()*6) + 1;
         die2 = (int)(Math.random()*6) + 1;
    }
} // end class PairOfDice
```

The instance variables `die1` and `die2` are initialized to the values 3 and 4 respectively. These initializations are executed whenever a `PairOfDice` object is constructed. It's important to understand when and how this happens. There can be many `PairOfDice` objects. Each time one is created, it gets its own instance variables, and the assignments "`die1 = 3` " and "`die2 = 4` " are executed to fill in the values of those variables. To make this clearer, consider a variation of the `PairOfDice` class:

```
public class PairOfDice {
    public int die1 = (int)(Math.random()*6) + 1;
    public int die2 = (int)(Math.random()*6) + 1;

    public void roll() {
         die1 = (int)(Math.random()*6) + 1;
         die2 = (int)(Math.random()*6) + 1;
    }
} // end class PairOfDice
```

Here, the dice are initialized to random values, as if a new pair of dice were being thrown onto the gaming table. Since the initialization is executed for each new object, a set of random initial values will be computed for each new pair of dice. Different pairs of dice can have different initial values. For initialization of **static** member variables, of course, the situation is quite different. There is only one copy of a `static` variable, and initialization of that variable is executed just once, when the class is first loaded.

If you don't provide any initial value for an instance variable, a default initial value is provided automatically. Instance variables of numerical type (`int`, `double`, etc.) are automatically initialized to zero if you provide no other values; `boolean` variables are initialized to `false` ; and `char` variables, to the Unicode character with code number zero. An instance variable can also be a variable of object type. For such variables, the default initial value is `null`. (In particular, since `Strings` are objects, the default initial value for `String` variables is `null`.)

Objects are created with the operator, `new`. For example, a program that wants to use a `PairOfDice` object could say:

```
PairOfDice dice;   // Declare a variable of type PairOfDice.
dice = new PairOfDice();  // Construct a new object and store a
                          //   reference to it in the variable.
```

In this example, "`new PairOfDice()` " is an expression that allocates memory for the object, initializes the object's instance variables, and then returns a reference to the object.

This reference is the value of the expression, and that value is stored by the assignment statement in the variable, `dice`. Part of this expression, "`PairOfDice()` ", looks like a subroutine call, and that is no accident. It is, in fact, a call to a special type of subroutine called a constructor. This might puzzle you, since there is no such subroutine in the class definition. However, every class has a constructor. If the programmer doesn't provide one, then the system will provide a default constructor. This default constructor does nothing beyond the basics: allocate memory and initialize instance variables. If you want more than that to happen when an object is created, you can include one or more constructors in the class definition.

The definition of a constructor looks much like the definition of any other subroutine, with three exceptions. A constructor does not have any return type (not even `void`). The name of the constructor must be the same as the name of the class in which it is defined. The only modifiers that can be used on a constructor definition are the access modifiers `public`, `private`, and `protected`. (In particular, a constructor can't be declared `static`.)

However, a constructor does have a subroutine body of the usual form, a block of statements. There are no restrictions on what statements can be used. And it can have a list of formal parameters. In fact, the ability to include parameters is one of the main reasons for using constructors. The parameters can provide data to be used in the construction of the object. For example, a constructor for the `PairOfDice` class could provide the values that are initially showing on the dice. Here is what the class would look like in that case:

```
public class PairOfDice {
    public int die1;   // Number showing on the first die.
    public int die2;   // Number showing on the second die.

    public PairOfDice(int val1, int val2) {
            // Constructor.  Creates a pair of dice that
            // are initially showing the values val1 and val2.
         die1 = val1;  // Assign specified values
         die2 = val2;  //           to the instance variables.
    }

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
         die1 = (int)(Math.random()*6) + 1;
         die2 = (int)(Math.random()*6) + 1;
    }
} // end class PairOfDice
```

The constructor is declared as "`public PairOfDice(int val1, int val2)...` ", with no return type and with the same name as the name of the class. This is how the Java compiler recognizes a constructor. The constructor has two parameters, and values for these parameters must be provided when the constructor is called. For example, the expression "`new PairOfDice(3,4)` " would create a `PairOfDice` object in which the values of the instance variables `die1` and `die2` are initially 3 and 4. Of course, in a program, the value returned by the constructor should be used in some way, as in

```
PairOfDice dice;            // Declare a variable of type PairOfDice.
dice = new PairOfDice(1,1); // Let dice refer to a new PairOfDice
                            //   object that initially shows 1, 1.
```

Now that we've added a constructor to the `PairOfDice` class, we can no longer create an object by saying "`new PairOfDice()` "! The system provides a default constructor for a class **only** if the class definition does not already include a constructor. However, this is not a big problem, since we can add a second constructor to the class, one that has no parameters. In fact, you can have as many different constructors as you want, as long as their signatures are different, that is, as long as they have different numbers or types of formal parameters. In the `PairOfDice` class, we might have a constructor with no parameters which produces a pair of dice showing random numbers:

```
public class PairOfDice {
    public int die1;   // Number showing on the first die.
    public int die2;   // Number showing on the second die.

    public PairOfDice() {
            // Constructor.  Rolls the dice, so that they initially
            // show some random values.
        roll();  // Call the roll() method to roll the dice.
    }

    public PairOfDice(int val1, int val2) {
            // Constructor.  Creates a pair of dice that
            // are initially showing the values val1 and val2.
        die1 = val1;  // Assign specified values
        die2 = val2;  //            to the instance variables.
    }

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }
} // end class PairOfDice
```

Now we have the option of constructing a `PairOfDice` object either with "`new PairOfDice()` " or with "`new PairOfDice(x,y)` ", where `x` and `y` are `int` -valued expressions.

This class, once it is written, can be used in any program that needs to work with one or more pairs of dice. None of those programs will ever have to use the obscure incantation "`(int)(Math.random()*6)+1` ", because it's done inside the `PairOfDice` class. And the programmer, having once gotten the dice-rolling thing straight will never have to worry about it again. Here, for example, is a main program that uses the `PairOfDice` class to count how many times two pairs of dice are rolled before the two pairs come up showing the same value:

```
public class RollTwoPairs {

    public static void main(String[] args) {
        PairOfDice firstDice;  // Refers to the first pair of dice.
        firstDice = new PairOfDice();

        PairOfDice secondDice; // Refers to the second pair of dice.
        secondDice = new PairOfDice();

        int countRolls;  // Counts how many times the two pairs of
                         //    dice have been rolled.

        int total1;      // Total showing on first pair of dice.
        int total2;      // Total showing on second pair of dice.

        countRolls = 0;

        do {  // Roll the two pairs of dice until totals are the same.

            firstDice.roll();    // Roll the first pair of dice.
            total1 = firstDice.die1 + firstDice.die2;   // Get total.
            System.out.println("First pair comes up  " + total1);

            secondDice.roll();    // Roll the second pair of dice.
            total2 = secondDice.die1 + secondDice.die2;   // Get total.
            System.out.println("Second pair comes up " + total2);

            countRolls++;   // Count this roll.

            System.out.println();  // Blank line.

        } while (total1 != total2);

        System.out.println("It took " + countRolls
                           + " rolls until the totals were the same.");
    } // end main()
} // end class RollTwoPairs
```

This applet simulates this program:

Applet Reference: see (Applet "RollTwoPairsConsole")

Constructors are subroutines, but they are subroutines of a special type. They are certainly not instance methods, since they don't belong to objects. Since they are responsible for creating objects, they exist before any objects have been created. They are more like `static` member subroutines, but they are not and cannot be declared to be `static`. In fact, according to the Java language specification, they are technically not members of the class at all!

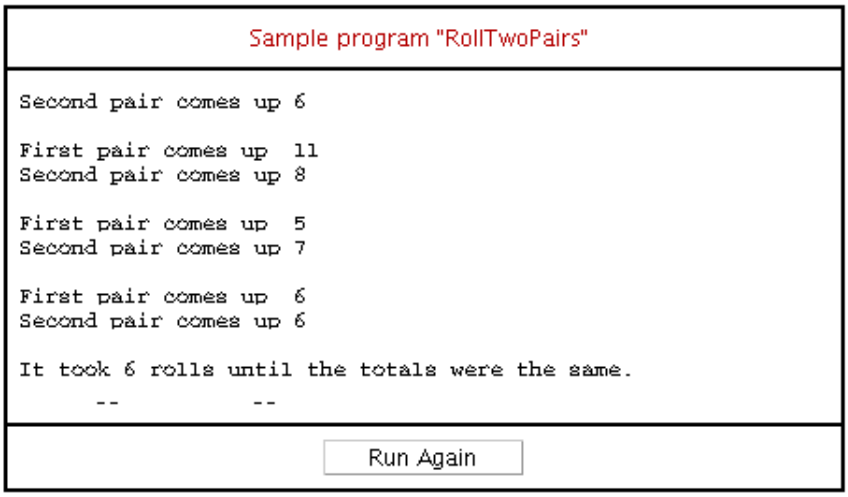


Figure 2.2: Applet "RollTwoPairsConsole"

Unlike other subroutines, a constructor can only be called using the `new` operator, in an expression that has the form

`new` **class-name** (**parameter-list**)

where the **parameter-list** is possibly empty. I call this an expression because it computes and returns a value, namely a reference to the object that is constructed. Most often, you will store the returned reference in a variable, but it is also legal to use a constructor call in other ways, for example as a parameter in a subroutine call or as part of a more complex expression. Of course, if you don't save the reference in a variable, you won't have any way of referring to the object that was just created.

A constructor call is more complicated than an ordinary subroutine or function call. It is helpful to understand the exact steps that the computer goes through to execute a constructor call:

1. First, the computer gets a block of unused memory in the heap, large enough to hold an object of the specified type.

2. It initializes the instance variables of the object. If the declaration of an instance variable specifies an initial value, then that value is computed and stored in the instance variable. Otherwise, the default initial value is used.

3. The actual parameters in the constructor, if any, are evaluated, and the values are assigned to the formal parameters of the constructor.

4. The statements in the body of the constructor, if any, are executed.

5. A reference to the object is returned as the value of the constructor call.

The end result of this is that you have a reference to a newly constructed object. You can use this reference to get at the instance variables in that object or to call its instance methods.

For another example, consider the `Student` class. I'll add a constructor, and I'll also take the opportunity to make the instance variable, `name`, private.

```
public class Student {
   private String name;                  // Student's name.
   public double test1, test2, test3;   // Grades on three tests.

   Student(String theName) {
        // Constructor for Student objects;
        // provides a name for the Student.
      name = theName;
   }

   public String getName() {
        // Accessor method for reading value of private
        // instance variable, name.
      return name;
   }

   public double getAverage() {   // Compute average test grade.
      return (test1 + test2 + test3) / 3;
   }
}  // end of class Student
```

An object of type `Student` contains information about some particular student. The constructor in this class has a parameter of type `String` , which specifies the name of that student. Objects of type `Student` can be created with statements such as:

```
std = new Student("John Smith");
std1 = new Student("Mary Jones");
```

In the original version of this class, the value of `name` had to be assigned by a program after it created the object of type `Student`. There was no guarantee that the programmer would always remember to set the `name` properly. In the new version of the class, there is no way to create a `Student` object except by calling the constructor, and that constructor automatically sets the `name`. The programmer's life is made easier, and whole hordes of frustrating bugs are squashed before they even have a chance to be born.

Another type of guarantee is provided by the `private` modifier. Since the instance variable, `name`, is `private`, there is no way for any part of the program outside the `Student` class to get at the `name` directly. The program sets the value of `name`, indirectly, when it calls the constructor. I've provided a function, `getName()`, that can be used from outside the class to find out the `name` of the student. But I haven't provided any way to change the name. Once a student object is created, it keeps the same name as long as it exists.

2.2.1 Garbage Collection

So far, this section has been about creating objects. What about destroying them? In Java, the destruction of objects takes place automatically.

An object exists in the heap, and it can be accessed only through variables that hold references to the object. What should be done with an object if there are no variables that refer to it? Such things can happen. Consider the following two statements (though in reality, you'd never do anything like this):

```
Student std = new Student("John Smith");
std = null;
```

In the first line, a reference to a newly created `Student` object is stored in the variable `std`. But in the next line, the value of `std` is changed, and the reference to the `Student` object is gone. In fact, there are now no references whatsoever to that object stored in any variable. So there is no way for the program ever to use the object again. It might as well not exist. In fact, the memory occupied by the object should be reclaimed to be used for another purpose.

Java uses a procedure called garbage collection to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are "garbage". In the above example, it was very easy to see that the `Student` object had become garbage. Usually, it's much harder. If an object has been used for a while, there might be several references to the object stored in several variables. The object doesn't become garbage until all those references have been dropped.

In many other programming languages, it's the programmer's responsibility to delete the garbage. Unfortunately, keeping track of memory usage is very error-prone, and many serious program bugs are caused by such errors. A programmer might accidently delete an object even though there are still references to that object. This is called a dangling pointer error, and it leads to problems when the program tries to access an object that is no longer there. Another type of error is a memory leak, where a programmer neglects to delete objects that are no longer in use. This can lead to filling memory with objects that are completely inaccessible, and the program might run out of memory even though, in fact, large amounts of memory are being wasted.

Because Java uses garbage collection, such errors are simply impossible. Garbage collection is an old idea and has been used in some programming languages since the 1960s. You might wonder why all languages don't use garbage collection. In the past, it was considered too slow and wasteful. However, research into garbage collection techniques combined with the incredible speed of modern computers have combined to make garbage collection feasible. Programmers should rejoice.

2.3 Programming with Objects

THERE ARE SEVERAL WAYS in which object-oriented concepts can be applied to the process of designing and writing programs. The broadest of these is object-oriented analysis and design which applies an object-oriented methodology to the earliest stages of program development, during which the overall design of a program is created. Here, the idea is to identify things in the problem domain that can be modeled as objects. On another level, object-oriented programming encourages programmers to produce generalized software components that can be used in a wide variety of programming projects.

2.3.1 Built-in Classes

Although the focus of object-oriented programming is generally on the design and implementation of new classes, it's important not to forget that the designers of Java have already provided a large number of reusable classes. Some of these classes are meant to be extended to produce new classes, while others can be used directly to create useful objects. A true mastery of Java requires familiarity with the full range of built-in classes – something that takes a lot of time and experience to develop. In the nextchapter, we will begin the study of Java's GUI classes, and you will encounter other built-in classes

throughout the remainder of this book. But let's take a moment to look at a few built-in classes that you might find useful.

A string can be built up from smaller pieces using the + operator, but this is not very efficient. If `str` is a `String` and `ch` is a character, then executing the command "`str=str+ch;` " involves creating a whole new string that is a copy of `str`, with the value of `ch` appended onto the end. Copying the string takes some time. Building up a long string letter by letter would require a surprising amount of processing. The class `java.lang.StringBuffer` makes it possible to be efficient about building up a long string from a number of smaller pieces. Like a `String`, a `StringBuffer` contains a sequence of characters. However, it is possible to add new characters onto the end of a `StringBuffer` without making a copy of the data that it already contains. If `buffer` is a variable of type `StringBuffer` and `x` is a value of any type, then the command `buffer.append(x)` will add `x`, converted into a string representation, onto the end of the data that was already in the buffer. This command actually modifies the buffer, rather than making a copy, and that can be done efficiently. A long string can be built up in a `StringBuffer` using a sequence of `append()` commands. When the string is complete, the function `buffer.toString()` will return a copy of the string in the buffer as an ordinary value of type `String`.

A number of useful classes are collected in the package `java.util`. For example, this package contains classes for working with collections of objects (one of the contexts in which wrapper classes for primitive types are useful). We will study these collection classes in Chapter12. The class `java.util.Date` is used to represent times. When a `Date` object is constructed without parameters, the result represents the current date and time, so an easy way to display this information is:

```
System.out.println( new Date() );
```

Of course, to use the `Date` class in this way, you must make it available by importing it with one of the statements "`importjava.util.Date;` " or "`importjava.util.*;` " at the beginning of your program.

Finally, I will mention the class `java.util.Random`. An object belonging to this class is a *source* of random numbers. (The standard function `Math.random()` uses one of these objects behind the scenes to generate its random numbers.) An object of type `Random` can generate random integers, as well as random real numbers. If `randGen` is created with the command:

```
Random randGen = new Random();
```

and if `N` is a positive integer, then `randGen.nextInt(N)` generates a random integer in the range from `0` to `N-1`. For example, this makes it a little easier to roll a pair of dice. Instead of saying "`die1=(int)(6*Math.random())+1;`", one can say
"`die1=randGen.nextInt(6)+1;`".
(Since you also have to import the class `java.util.Random` and create the `Random` object, you might not agree that it is actually easier.)

The main point here, again, is that many problems have already been solved, and the solutions are available in Java's standard classes. If you are faced with a task that looks like it should be fairly common, it might be worth looking through a Java reference to see whether someone has already written a subroutine that you can use.

2.3.2 Generalized Software Components

Every programmer builds up a stock of techniques and expertise expressed as snippets of code that can be reused in new programs using the tried-and-true method of cut-and-paste: The old code is physically copied into the new program and then edited to cus-

tomize it as necessary. The problem is that the editing is error-prone and time-consuming, and the whole enterprise is dependent on the programmer's ability to pull out that particular piece of code from last year's project that looks like it might be made to fit. (On the level of a corporation that wants to save money by not reinventing the wheel for each new project, just keeping track of all the old wheels becomes a major task.)

Well-designed classes are software components that can be reused without editing. A well-designed class is not carefully crafted to do a particular job in a particular program. Instead, it is crafted to model some particular type of object or a single coherent concept. Since objects and concepts can recur in many problems, a well-designed class is likely to be reusable without modification in a variety of projects.

Furthermore, in an object-oriented programming language, it is possible to make subclasses of an existing class. This makes classes even more reusable. If a class needs to be customized, a subclass can be created, and additions or modifications can be made in the subclass without making any changes to the original class. This can be done even if the programmer doesn't have access to the source code of the class and doesn't know any details of its internal, hidden implementation. We will discuss subclasses later.

2.4 Programming Examples

The `PairOfDice` class is already an example of a generalized software component, although one that could certainly be improved. The class represents a single, coherent concept, "a pair of dice." The instance variables hold the data relevant to the state of the dice, that is, the number showing on each of the dice. The instance method represents the behaviour of a pair of dice, that is, the ability to be rolled. This class would be reusable in many different programming projects.

On the other hand, the `Student` class is not very reusable. It seems to be crafted to represent students in a particular course where the grade will be based on three tests. If there are more tests or quizzes or papers, it's useless. If there are two people in the class who have the same name, we are in trouble (one reason why numerical student ID's are often used). Admittedly, it's much more difficult to develop a general-purpose student class than a general-purpose pair-of-dice class. But this particular `Student` class is good mostly as an example in a programming textbook.

Let's do another example in a domain that is simple enough that we have a chance of coming up with something reasonably reusable. Consider card games that are played with a standard deck of playing cards (a so-called "poker" deck, since it is used in the game of poker). In a typical card game, each player gets a hand of cards. The deck is shuffled and cards are dealt one at a time from the deck and added to the players' hands. In some games, cards can be removed from a hand, and new cards can be added. The game is won or lost depending on the value (ace, 2, ..., king) and suit (spades, diamonds, clubs, hearts) of the cards that a player receives. If we look for nouns in this description, there are several candidates for objects: game, player, hand, card, deck, value, and suit. Of these, the value and the suit of a card are simple values, and they will just be represented as instance variables in a `Card` object. In a complete program, the other five nouns might be represented by classes. But let's work on the ones that are most obviously reusable: card, hand, and deck.

If we look for verbs in the description of a card game, we see that we can shuffle a deck and deal a card from a deck. This gives use us two candidates for instance methods in a `Deck` class. Cards can be added to and removed from hands. This gives two candidates for instance methods in a `Hand` class. Cards are relatively passive things, but we need to be

able to determine their suits and values. We will discover more instance methods as we go along.

First, we'll design the deck class in detail. When a deck of cards is first created, it contains 52 cards in some standard order. The `Deck` class will need a constructor to create a new deck. The constructor needs no parameters because any new deck is the same as any other. There will be an instance method called `shuffle()` that will rearrange the 52 cards into a random order. The `dealCard()` instance method will get the next card from the deck. This will be a function with a return type of `Card`, since the caller needs to know what card is being dealt. It has no parameters. What will happen if there are no more cards in the deck when its `dealCard()` method is called? It should probably be considered an error to try to deal a card from an empty deck. But this raises another question: How will the rest of the program know whether the deck is empty? Of course, the program could keep track of how many cards it has used. But the deck itself should know how many cards it has left, so the program should just be able to ask the deck object. We can make this possible by specifying another instance method, `cardsLeft()`, that returns the number of cards remaining in the deck. This leads to a full specification of all the subroutines in the `Deck` class:

Constructor and instance methods in class Deck:

```
public Deck()
      // Constructor.  Create an unshuffled deck of cards.

public void shuffle()
      // Put all the used cards back into the deck,
      // and shuffle it into a random order.

public int cardsLeft()
      // As cards are dealt from the deck, the number of
      // cards left decreases.  This function returns the
      // number of cards that are still left in the deck.

public Card dealCard()
      // Deals one card from the deck and returns it.
```

This is everything you need to know in order to use the `Deck` class. Of course, it doesn't tell us how to write the class. This has been an exercise in design, not in programming. In fact, writing the class involves a programming technique, arrays, which will not be covered until Chapter 8. Nevertheless, you can look at the source code, Deck.java, if you want. And given the source code, you can use the class in your programs without understanding the implementation.

We can do a similar analysis for the `Hand` class. When a hand object is first created, it has no cards in it. An `addCard()` instance method will add a card to the hand. This method needs a parameter of type `Card` to specify which card is being added. For the `removeCard()` method, a parameter is needed to specify which card to remove. But should we specify the card itself ("Remove the ace of spades"), or should we specify the card by its position in the hand ("Remove the third card in the hand")? Actually, we don't have to decide, since we can allow for both options. We'll have two `removeCard()` instance methods, one with a parameter of type `Card` specifying the card to be removed and one

with a parameter of type `int` specifying the position of the card in the hand. (Remember that you can have two methods in a class with the same name, provided they have different types of parameters.) Since a hand can contain a variable number of cards, it's convenient to be able to ask a hand object how many cards it contains. So, we need an instance method `getCardCount()` that returns the number of cards in the hand. When I play cards, I like to arrange the cards in my hand so that cards of the same value are next to each other. Since this is a generally useful thing to be able to do, we can provide instance methods for sorting the cards in the hand. Here is a full specification for a reusable `Hand` class:

Constructor and instance methods in class Hand:

```
public Hand() {
      // Create a Hand object that is initially empty.
public void clear() {
      // Discard all cards from the hand, making the hand empty.
public void addCard(Card c) {
      // Add the card c to the hand.  c should be non-null.
      // (If c is null, nothing is added to the hand.)
public void removeCard(Card c) {
      // If the specified card is in the hand, it is removed.
public void removeCard(int position) {
      // If the specified position is a valid position in the
      // hand, then the card in that position is removed.
public int getCardCount() {
      // Return the number of cards in the hand.
public Card getCard(int position) {
       // Get the card from the hand in given position, where
       // positions are numbered starting from 0.  If the
       // specified position is not the position number of
       // a card in the hand, then null is returned.
public void sortBySuit() {
      // Sorts the cards in the hand so that cards of the same
      // suit are grouped together, and within a suit the cards
      // are sorted by value.  Note that aces are considered
      // to have the lowest value, 1.
public void sortByValue() {
      // Sorts the cards in the hand so that cards of the same
      // value are grouped together.  Cards with the same value
      // are sorted by suit. Note that aces are considered
      // to have the lowest value, 1.
```

Again, you don't yet know enough to implement this class. But given the source code, Hand.java, you can use the class in your own programming projects.

We **have** covered enough material to write a `Card` class. The class will have a constructor that specifies the value and suit of the card that is being created. There are four suits, which can be represented by the integers 0, 1, 2, and 3. It would be tough to remember which number represents which suit, so I've defined named constants in the `Card` class to represent the four possibilities. For example, `Card.SPADES` is a constant that represents the suit, spades. (These constants are declared to be `public final static ints`. This

is one case in which it makes sense to have `static` members in a class that otherwise has only instance variables and instance methods.) The possible values of a card are the numbers 1, 2, ..., 13, with 1 standing for an ace, 11 for a jack, 12 for a queen, and 13 for a king. Again, I've defined some named constants to represent the values of aces and face cards. So, cards can be constructed by statements such as:

```
card1 = new Card( Card.ACE, Card.SPADES );  // Construct ace of spades.
card2 = new Card( 10, Card.DIAMONDS );   // Construct 10 of diamonds.
card3 = new Card( v, s );  // This is OK, as long as v and s
                           // are integer expressions.
```

A `Card` object needs instance variables to represent its value and suit. I've made these `private` so that they cannot be changed from outside the class, and I've provided instance methods `getSuit()` and `getValue()` so that it will be possible to discover the suit and value from outside the class. The instance variables are initialized in the constructor, and are never changed after that. In fact, I've declared the instance variables `suit` and `value` to be `final`, since they are never changed after they are initialized. (An instance variable can be declared `final` provided it is either given an initial value in its declaration or is initialized in every constructor in the class.)

Finally, I've added a few convenience methods to the class to make it easier to print out cards in a human-readable form. For example, I want to be able to print out the suit of a card as the word "Diamonds", rather than as the meaningless code number 2, which is used in the class to represent diamonds. Since this is something that I'll probably have to do in many programs, it makes sense to include support for it in the class. So, I've provided instance methods `getSuitAsString()` and `getValueAsString()` to return string representations of the suit and value of a card. Finally, there is an instance method `toString()` that returns a string with both the value and suit, such as "Queen of Hearts". There is a good reason for calling this method `toString()`. When any object is output with `System.out.print()`, the object's `toString()` method is called to produce the string representation of the object. For example, if `card` refers to an object of type `Card`, then `System.out.println(card)` is equivalent to `System.out.println(card.toString())`. Similarly, if an object is appended to a string using the + operator, the object's `toSring()` method is used. Thus,

```
System.out.println( "Your card is the " + card );
```

is equivalent to

```
System.out.println( "Your card is the " + card.toString() );
```

If the card is the queen of hearts, either of these will print out "`Your card is the Queen of Hearts`".

Here is the complete `Card` class. It is general enough to be highly reusable, so the work that went into designing, writing, and testing it pays off handsomely in the long run.

```
/*
   An object of class card represents one of the 52 cards in a
   standard deck of playing cards.  Each card has a suit and
   a value.
*/

public class Card {

    public final static int SPADES = 0,     // Codes for the 4 suits.
                            HEARTS = 1,
                            DIAMONDS = 2,
                            CLUBS = 3;

    public final static int ACE = 1,      // Codes for non-numeric cards.
                            JACK = 11,    //   Cards 2 through 10 have
                            QUEEN = 12,   //   their  numerical values
                            KING = 13;    //   for their codes.

    private final int suit;   // The suit of this card, one of the
                              //    four constants:  SPADES, HEARTS,
                              //    DIAMONDS, CLUBS.

    private final int value;  // The value of this card, from 1 to 13.

    public Card(int theValue, int theSuit) {
            // Construct a card with the specified value and suit.
            // Value must be between 1 and 13.  Suit must be between
            // 0 and 3.  If the parameters are outside these ranges,
            // the constructed card object will be invalid.
        value = theValue;
        suit = theSuit;
    }

    public int getSuit() {
            // Return the int that codes for this card's suit.
        return suit;
    }

    public int getValue() {
            // Return the int that codes for this card's value.
        return value;
    }

    public String getSuitAsString() {
            // Return a String representing the card's suit.
            // (If the card's suit is invalid, "??" is returned.)
        switch ( suit ) {
           case SPADES:   return "Spades";
           case HEARTS:   return "Hearts";
```

```
            case DIAMONDS: return "Diamonds";
            case CLUBS:    return "Clubs";
            default:       return "??";
        }
    }

    public String getValueAsString() {
            // Return a String representing the card's value.
            // If the card's value is invalid, "??" is returned.
        switch ( value ) {
            case 1:   return "Ace";
            case 2:   return "2";
            case 3:   return "3";
            case 4:   return "4";
            case 5:   return "5";
            case 6:   return "6";
            case 7:   return "7";
            case 8:   return "8";
            case 9:   return "9";
            case 10:  return "10";
            case 11:  return "Jack";
            case 12:  return "Queen";
            case 13:  return "King";
            default:  return "??";
        }
    }

    public String toString() {
            // Return a String representation of this card, such as
            // "10 of Hearts" or "Queen of Spades".
        return getValueAsString() + " of " + getSuitAsString();
    }
} // end class Card
```

2.5 More about Access Modifiers

A class can be declared to be `public`. A public class can be accessed from anywhere. Certain classes have to be public. A class that defines a stand-alone application must be public, so that the system will be able to get at its `main()` routine. A class that defines an applet must be public so that it can be used by a Web browser. If a class is not declared to be `public`, then it can only be used by other classes in the same "package" as the class. Classes that are not explicitly declared to be in any package are put into something called the default package. All the examples in this textbook are in the default package, so they are all accessible to one another whether or not they are declared public. So, except for applications and applets, which must be `public`, it makes no practical difference whether our classes are declared to be public or not.

However, once you start writing packages, it does make a difference. A package should contain a set of related classes. Some of those classes are meant to be public, for access from outside the package. Others can be part of the internal workings of the package,

and they should not be made public. A package is a kind of black box. The public classes in the package are the interface. (More exactly, the public variables and subroutines in the public classes are the interface). The non-public classes are part of the non-public implementation. Of course, all the classes in the package have unrestricted access to one another.

Following this model, I will tend to declare a class `public` if it seems like it might have some general applicability. If it is written just to play some sort of auxiliary role in a larger project, I am more likely not to make it `public`.

A member variable or subroutine in a class can also be declared to be `public`, which means that it is accessible from anywhere. It can be declared to be `private`, which means that it accessible only from inside the class where it is defined. Making a variable `private` gives you complete control over that variable. The only code that will ever manipulate it is the code you write in your class. This is an important kind of protection.

If no access modifier is specified for a variable or subroutine, then it is accessible from any class in the same package as the class. As with classes, in this textbook there is no practical difference between declaring a member `public` and using no access modifier at all. However, there might be stylistic reasons for preferring one over the other. And a real difference does arise once you start writing your own packages.

There is a third access modifier that can be applied to a member variable or subroutine. If it is declared to be `protected`, then it can be used in the class where it is defined and in any subclass of that class. This is obviously less restrictive than `private` and more restrictive than `public`. Classes that are written specifically to be used as a basis for making subclasses often have `protected` members. The `protected` members are there to provide a foundation for the subclasses to build on. But they are still invisible to the public at large.

2.6 Mixing Static and Non-static

Classes, as I've said, have two very distinct purposes. A class can be used to group together a set of static member variables and static member subroutines. Or it can be used as a factory for making objects. The non-static variables and subroutine definintions in the class definition specify the instance variables and methods of the objects. In most cases, a class performs one or the other of these roles, not both.

Sometimes, however, static and non-static members are mixed in a single class. In this case, the class plays a dual role. Sometimes, these roles are completely separate. It is also possible for the static and non-static parts of a class to interact. This happens when instance methods use static member variables or call static member subroutines. An instance method belongs to an object, not to the class itself, and there can be many objects with their own versions of the instance method. But there is only one copy of a static member variable. So, effectively, we have many objects sharing that one variable.

Suppose, for example, that we want to write a `PairOfDice` class that uses the `Random` class mentioned previously for rolling the dice. To do this, a `PairOfDice` object needs access to an object of type `Random`. But there is no need for each `PairOfDice` object to have a separate `Random` object. (In fact, it would not even be a good idea: Because of the way random number generators work, a program should, in general, use only one source of random numbers.) A nice solution is to have a single `Random` variable as a `static` member of the `PairOfDice` class, so that it can be shared by all `PairOfDice` objects. For example:

```
class PairOfDice {
   private static Random randGen = new Random();
   public int die1;   // Number showing on the first die.
   public int die2;   // Number showing on the second die.

   public PairOfDice() {
           // Constructor.  Creates a pair of dice that
           // initially shows random values.
        roll();
   }
   public void roll() {
           // Roll the dice by setting each of the dice to be
           // a random number between 1 and 6.
        die1 = randGen.nextInt(6) + 1;
        die2 = randGen.nextInt(6) + 1;
   }
} // end class PairOfDice
```

As another example, let's rewrite the `Student` class. I've added an `ID` for each student and a `static` member called `nextUniqueID`. Although there is an `ID` variable in each student object, there is only one `nextUniqueID` variable.

```
public class Student {
     private String name;  // Student's name.
     private int ID;  // Unique ID number for this student.
     public double test1, test2, test3;   // Grades on three tests.
     private static int nextUniqueID = 0;
               // keep track of next available unique ID number

     Student(String theName) {
          // Constructor for Student objects;
          // provides a name for the Student,
          // and assigns the student a unique
          // ID number.
        name = theName;
        nextUniqueID++;
        ID = nextUniqueID;
     }

     public String getName() {
          // Accessor method for reading value of private
          // instance variable, name.
        return name;
     }

     public int getID() {
          // Accessor method for reading value of ID.
        return ID;
     }
```

```
    public double getAverage() {
         // Compute average test grade.
       return (test1 + test2 + test3) / 3;
    }
}  // end of class Student
```

The initialization "`nextUniqueID = 0` " is done only once, when the class is first loaded. Whenever a `Student` object is constructed and "`nextUniqueID++;`" is ex ecuted, it's always the same static member variable that is being incremented. When the very first `Student` object is created, `nextUniqueID` becomes 1. When the second object is created, `nextUniqueID` becomes 2. After the third object, it becomes 3. And so on. The constructor stores the new value of `nextUniqueID` in the `ID` variable of the object that is being created. Of course, `ID` is an instance variable, so every object has its own individual `ID` variable. The class is constructed so that each student will automatically get a different value for its `ID` variable. Furthermore, the `ID` variable is `private`, so there is no way for this variable to be tampered with after the object has been created. You are guaranteed, just by the way the class is designed, that every student object will have its own permanent, unique identification number. Which is kind of cool if you think about it.

2.7 Input and Ouput in Java

2.7.1 The Scanner Class

`System` is the name of the built-in class that contains methods and objects used to get input from the keyboard, print text on the screen, and do file input/output (I/0).

`System.out` is the name of the object we use to to display text on the screen. When you invoke `print` and `println`, you invoke them on the object named `System.out`.

Java 5.0 introduced a simplified reader class, the `Scanner` class which accesses the input or a file, and does limited conversion for the user.

For example, to read input from the keyboard one usually connects to the input stream (the object `System.in`), reads a line (as a string) at a time and breaks the string into tokens (parts separated by some delimiter, usually whitespace) and then convert the tokens to various number formats. With the Scanner class, we construct a Scanner object with the stream `System.in`) as input. The resulting tokens may then be converted into values of different types using the various `next` methods.

By default the scanner uses whitespace as the delimiter. Whitespace may be blank spaces, tabs or newlines. The scanner can also use delimiters other than whitespace.

The Scanner class provides various `next` methods (One for each primitive type). Some of these are listed below.:

- `next()` returns the next token as a string.
- `nextInt()` returns the next integer.
- `nextDouble()` returns the next double.
- `nextLong()` returns the next long.
- `nextBoolean()` returns the next boolean.

Each of these methods has a companion `hasNext()` method: `hasNextInteger()`, `hasNextDouble()`, `hasNextLong()` etc. that returns `true` if the next token is an integer, double or long etc.

The `next()` methods and their primitive-type companion methods (such as `nextInt()` and `hasNextInt()`) first skip any input that matches the delimiter pattern (e.g. spaces), and then attempt to return the next token.

Unfortunately, there is no `next` method for reading in a character. You can read the character as a string and convert it as shown below.

```
Scanner sc = new Scanner(System.in);
char c = sc.next().charAt(0);
```

This code allows a user to read input from `System.in`.

```
import java.util.Scanner;
  ...
Scanner keyboard = new Scanner(System.in);   //Create a Scanner object

System.out.print("Enter your name> ");
String name = keyboard.next();  //Read the next input token as a string
System.out.println();
System.out.print("Enter grade> ");
int grade = keyboard.nextInt(); //Read the next input token
```

The steps are straight-forward: create a scanner object (you may choose any name for it) and read from it using the appropriate `next` method.

You can also use the Scanner class to read input from a file—this is not much different from doing input from the keyboard: instead of passing `System.in` as a parameter to the Scanner constructor, we pass a file object. Once the Scanner object has been created with either `System.in` or a file, you use the Scanner object in the same way. As an example, this code allows long types to be assigned from entries in a file `myNumbers`:

```
Scanner myFile = new Scanner(new File("myNumbers"));
while (myFile.hasNextLong()) {
    long aLong = myFile.nextLong();
}
```

2.7.2 Formatted Output

Java now includes the ability to format output. The formatting system is similar to that in C language.

To produce formatted output you require a format string and an argument list. The format string is a `String` which may contain fixed text and one or more embedded format specifiers. The format specifiers specify how the argument/s should be formatted.

The format specifiers for general, character, and numeric types have the following syntax:

%[argument_index$][flags][width][.precision]conversion

Consider `%1$3.1f` as an example.

- The optional argument_index is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by ”1$”, the second by ”2$”, etc. In the example above, 1$ refers to the first argument.

- The required conversion is a character indicating how the argument should be formatted. The set of valid conversions for a given argument depends on the argument's data type.

  The following conversions apply to integer types (byte, short, int, long, and BigInteger:

  - 'd' The result is formatted as a decimal integer
  - 'o' The result is formatted as an octal integer
  - 'x', 'X' The result is formatted as a hexadecimal integer

  The following conversions apply to floating point types (float, double, and BigDecimal) :

  - 'e', 'E' The result is formatted as a decimal number in computerized scientific notation
  - 'f' The result is formatted as a decimal number

  In the example above, `f` means that we want to format the first argument as a decimal number.

- The optional flags is a set of characters that modify the output format. The set of valid flags depends on the conversion. For example the '-' flag means that the result will be left justified and the '+' means that the result will always include a sign.

- The optional width is a non-negative decimal integer indicating the minimum number of characters to be written to the output. The width refers to how many characters will be used to write the argument. If the width is larger than the length of the number, then blanks will be used to fill up the space.

- The optional precision is a non-negative decimal integer usually used to restrict the number of characters. The specific behavior depends on the conversion. For the floating-point conversions 'e', 'E', and 'f' the precision is the number of digits after the decimal separator.

Consider the following example where we read in a maximum mark and a student's mark and we then calculate and print the percentage:

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

            Scanner keyboard = new Scanner(System.in);
            System.out.print("  Enter the max mark    > ");
            double max = keyboard.nextDouble();
            System.out.print("  Enter your mark    > ");
            double mark = keyboard.nextDouble();

            double percentage = (mark/max)*100;

            System.out.printf("You scored %1$.1f out'' +
                              ''of %2$5.1f which is %3$3.2f %%  ",
                              mark, max, percentage);
    }
```

Notice that the `printf` statement consists of the format string and three arguments. The format string specifes how the arguments are to be formatted. For example %2$3.1f specifies that the 2^{nd} argument (`max`) must be formatted as a decimal number (`f`) using a width of 3 characters with one decimal place. This outputs

```
You scored 13.0 out of  30.0 which is 43.33 %
```

The %% is needed to print the %. Notice also the extra spaces in front of the 30.0.

A final example: Calculating and outputting compound interest on an initial input amount.

```
/*
* Calculates compound interest - by determining final amount
* after investing an amount for a number years at R% interest
*/

import java.util.*;

public class CompInterest2  {

    public static void main(String[] args)   {
        double  amount,    // the initial amount invested
        years,     // the number of years invested
        rate,      // the annual interest rate as a %
        finalAmt;  // the final amount

        //the Scanner object to do input from the keyboard
        Scanner keyboard = new Scanner(System.in);

        // display a heading
        System.out.println("This program calculates compound interest");

        // input amount, years and interest rate
        System.out.print("  Enter initial amount    > R ");
        amount = keyboard.nextDouble();
        System.out.print("  Enter years invested    > ");
        years = keyboard.nextDouble();
        System.out.print("  Enter interest rate (%) > ");
        rate = keyboard.nextDouble();
        System.out.println();

        // calculate the final amount
        finalAmt = amount * Math.pow((100+rate)/100,years);

        // display the result
        System.out.printf("R %1$5.2f invested for %2$.0f years" +
                          "at %3$.2f %% interest yields R %4$.2f",
                           amount,years,rate,finalAmt );
        System.out.println();
     }
}
```

2.8 Summary

Classes define types. Objects are concrete instances of class and there may be multiple instances of a class. Objects store data in **fields** or **instance variables**, initialize this data with **constructors**, and manipulate this data with **methods**. The current value of the fields is the **state** of the object. A variable of class type stores either a **reference** to an object or the value `null`.

Constructors and Methods

A constructor sets up an object at creation. For example:

```
Prof wayne = new Prof();
```

A constructor can have arguments that are used to initialize the instance variables:

```
Prof wayne=new Prof("goddard");
```

A constructor must NOT have a type:

```
public class Prof {
   String name;
   Prof(String n) {
      name = n;
   }
 }
```

When a class is created, instance variables are initialized to default values: 0 for `int`, `null` for objects etc. Nevertheless, you should explicitly set primitive variables.

A method has a **header** with formal **parameters**, and a **body**. Its **signature** is the types of its parameters and its own type. The return type `void` means that the method returns nothing. A method is called with the . notation. For example:

```
object.method();
```

Most objects have **accessor** methods: these allow the user to **get** data from the object. **Mutator** methods allow the user to **set** data in the object.

A method should normally check its arguments. It notifies the caller of a problem by using an **exception** or a special return value. The programmer should try to avoid exceptions: consider error recovery and avoidance.

Access Modifiers

The fields of an object are accessible by all the methods of that object. A variable has a **scope** (where it is accessible from) and a **lifetime** (when it exists). The variables defined in a method are called **local variables** and are accessible only within the method and exist only while the method is being executed; an exception is **static** variables whose lifetime is the program's execution: they are always available.

A method or field can be made `private`, `public` or `protected`. `private` means it is accessible by no other classe, and `public` that it is accessible by all other classes. (We will discuss `protected` later.) Note that within the code for the class, one can access all fields of all objects of that type, even if they are private. For example, if one created one's own version of an integer, one could write:

```
public class MyInteger {
   private int val;
   public boolean isGreaterThan(MyInteger other) {
      return (this.val>other.val);
   }
```

The `this` refers to the current object; in this case it could be omitted:

```
      return (val>other.val);
```

Constants are usually uppercase: for example:

```
public final static int MAX_SIZE=100;
```

There are also `static` methods. Consider, for example, a `Coord` class that stores a point. A static method might be used to compute the distance between two points:

```
public static double distance( Coord p1, Coord p2)
```

It is then called as follows

```
Coord X = new Coord(0.0,0.0);
Coord Y = new Coord(3.0,4.0);
System.out.println( Coord.distance(X,Y) );
```

Assignment and Equality Testing

If `A` and `B` are primitive data types, then `A=B` sets `A` to the value of `B`: changing `B` will have no impact on `A`. If `A` and `B` are objects, then `A=B` sets `A` to refer to the same object that `B` references: until either `A` or `B` is re-assigned, they will point to the same object.

The test `A==B`, for objects, only compares whether they reference the SAME object in memory. It is good practice to override the `equals()` method if one will need to test two objects of one's class for equality. This will be discussed in the section on **casting**.

Objects as Return Values and Parameters

An object (even an array) can be returned from a method (but only one). An object can also be a parameter to a method.

Note that the method receives a **reference** to an object (this is called **pass by reference**). This means that changes to the object so referenced ARE reflected outside the class. On the other hand, changing a primitive data type parameter does NOT affect the value outside the class:

```
void clear(int x,int[] A){
  x=0;
  for(int i=0;i<A.length;i++)
     A[i]=0;
}
```

is called from

```
int y = 42 ;
int[] B = {1,2,3,4,5};
clear(y,B);
```

will cause all the values of `B` to be zero, but will not change `y`.

Objects for Numbers

For every primitive data type there is a corresponding object. So there are `Integer`, `Double` and `Character` classes, etc. These have several methods including:

- constructors which take the corresponding base type;
- accessor methods such as `double doubleVal()` which return the base type; and
- `static` utilities such as `Integer.parseInt(String str)` that translate a `String` into the base type.

Exceptions

Java uses `Exceptions` to signal problems with execution, such as an array index out of range or running out of memory. A `RuntimeException` is called an **unchecked exception** since it does not require compiler check. (Other `Exceptions` are checked and require `throws` statements and/or `try...catch` construction.) Example usage:

```
if( index<0 )
   throw new RuntimeException("Bad index");
```

2.9 Quiz Questions

Question 1: Object-oriented programming uses *classes* and *objects*. What are classes and what are objects? What is the relationship between classes and objects?

Question 2: Explain carefully what *null* means in Java, and why this special value is necessary.

Question 3: What is a *constructor?* What is the purpose of a constructor in a class?

Question 4: Suppose that `Kumquat` is the name of a class and that `fruit` is a variable of type `Kumquat`. What is the meaning of the statement "`fruit = new Kumquat();` "? That is, what does the computer do when it executes this statement? (Try to give a complete answer. The computer does several things.)

Question 5: What is meant by the terms *instance variable* and *instance method* ?

Question 6: Explain what is meant by the terms *subclass* and *superclass*.

Question 7: Explain the term *polymorphism*.

Question 8: Java uses "garbage collection" for memory management. Explain what is meant here by garbage collection. What is the alternative to garbage collection?

Question 9: For this problem, you should write a very simple but complete class. The class represents a counter that counts 0, 1, 2, 3, 4,.... The name of the class should be `Counter`. It has one `private` instance variable representing the value of the counter. It has two instance methods: `increment()` adds one to the counter value, and `getValue()` returns the current counter value. Write a complete definition for the class, `Counter`.

Question 10: This problem uses the `Counter` class from Question 9. The following program segment is meant to simulate tossing a coin 100 times. It should use two `Counter` objects, `headCount` and `tailCount`, to count the number of heads and the number of tails. Fill in the blanks so that it will do so.

```
Counter headCount, tailCount;
tailCount = new Counter();
headCount = new Counter();
```

```
for ( int flip = 0;  flip < 100;  flip++ ) {
   if (Math.random() < 0.5)    // There's a 50/50 chance that this is true.

       ______________________ ;   // Count a "head".

   else

       ______________________ ;   // Count a "tail".
}

System.out.println("There were " + ___________________ + " heads.");

System.out.println("There were " + ___________________ + " tails.");
```

2.10 Programming Exercises

1. In all versions of the `PairOfDice` class, the instance variables `die1` and `die2` are declared to be `public`. They really should be private, so that they are protected from being changed from outside the class. Write another version of the `PairOfDice` class in which the instance variables `die1` and `die2` are `private`. Your class will need methods that can be used to find out the values of `die1` and `die2`. (The idea is to protect their values from being changed from outside the class, but still to allow the values to be read.) Include other improvements in the class, if you can think of any. Test your class with a short program that counts how many times a pair of dice is rolled, before the total of the two dice is equal to two.

2. A common programming task is computing statistics of a set of numbers. (A statistic is a number that summarizes some property of a set of data.) Common statistics include the mean (also known as the average) and the standard deviation (which tells how spread out the data are from the mean). I have written a little class called `StatCalc` that can be used to compute these statistics, as well as the sum of the items in the dataset and the number of items in the dataset. You can read the source code for this class in the file StatCalc.java. If `calc` is a variable of type `StatCalc`, then the following methods are defined:

   - `calc.enter(item);` where `item` is a number, adds the item to the dataset.
   - `calc.getCount()` is a function that returns the number of items that have been added to the dataset.
   - `calc.getSum()` is a function that returns the sum of all the items that have been added to the dataset.
   - `calc.getMean()` is a function that returns the average of all the items.
   - `calc.getStandardDeviation()` is a function that returns the standard deviation of the items.

   Typically, all the data are added one after the other calling the `enter()` method over and over, as the data become available. After all the data have been entered, any of the other methods can be called to get statistical information about the data.

The methods `getMean()` and `getStandardDeviation()` should only be called if the number of items is greater than zero.

Modify the current source code, `StatCalc.java`, to add instance methods `getMax()` and `getMin()`. The `getMax()` method should return the largest of all the items that have been added to the dataset, and `getMin()` should return the smallest. You will need to add two new instance variables to keep track of the largest and smallest items that have been seen so far.

Test your new class by using it in a program to compute statistics for a set of non-zero numbers entered by the user. Start by creating an object of type `StatCalc` :

```
StatCalc  calc;   // Object to be used to process the data.
calc = new StatCalc();
```

Read numbers from the user and add them to the dataset. Use 0 as a sentinel value (that is, stop reading numbers when the user enters 0). After all the user's non-zero numbers have been entered, print out each of the six statistics that available from `calc`.

3. This problem uses the `PairOfDice` class from Exercise 5.1 and the `StatCalc` class from Exercise 5.2.

   The program in Exercise 4.4 performs the experiment of counting how many times a pair of dice is rolled before a given total comes up. It repeats this experiment 10000 times and then reports the average number of rolls. It does this whole process for each possible total (2, 3, ..., 12).

   Redo that exercise. But instead of just reporting the average number of rolls, you should also report the standard deviation and the maximum number of rolls. Use a `PairOfDice` object to represent the dice. Use a `StatCalc` object to compute the statistics. (You'll need a new `StatCalc` object for each possible total, 2, 3, ..., 12. You can use a new pair of dice if you want, but it's not necessary.)

4. The `BlackjackHand` class is an extension of the `Hand` class. `BlackjackHand` includes an instance method, `getBlackjackValue()`, that returns the value of the hand for the game of Blackjack. For this exercise, you will also need the `Deck` and `Card` classes.

   A Blackjack hand typically contains from two to six cards. Write a program to test the `BlackjackHand` class. You should create a `BlackjackHand` object and a `Deck` object. Pick a random number between 2 and 6. Deal that many cards from the deck and add them to the hand. Print out all the cards in the hand, and then print out the value computed for the hand by `getBlackjackValue()`. Repeat this as long as the user wants to continue.

   Your program will depend on Card.java, Deck.java, Hand.java, and BlackjackHand.java.

5. Write a program that let's the user play Blackjack. The game will be a simplified version of Blackjack as it is played in a casino. The computer will act as the dealer. As in the previous exercise, your program will need the classes defined in Card.java, Deck.java, Hand.java, and BlackjackHand.java. (This is the longest and most complex program that has come up so far in the exercises.)

   You should first write a subroutine in which the user plays one game. The subroutine should return a `boolean` value to indicate whether the user wins the game or not.

Return `true` if the user wins, `false` if the dealer wins. The program needs an object of class `Deck` and two objects of type `BlackjackHand`, one for the dealer and one for the user. The general object in Blackjack is to get a hand of cards whose value is as close to 21 as possible, without going over. The game goes like this.

First, two cards are dealt into each player's hand. If the dealer's hand has a value of 21 at this point, then the dealer wins. Otherwise, if the user has 21, then the user wins. (This is called a "Blackjack".) Note that the dealer wins on a tie, so if both players have Blackjack, then the dealer wins.

Now, if the game has not ended, the user gets a chance to add some cards to her hand. In this phase, the user sees her own cards and sees **one** of the dealer's two cards. (In a casino, the dealer deals himself one card face up and one card face down. All the user's cards are dealt face up.) The user makes a decision whether to "Hit", which means to add another card to her hand, or to "Stand", which means to stop taking cards.

If the user Hits, there is a possibility that the user will go over 21. In that case, the game is over and the user loses. If not, then the process continues. The user gets to decide again whether to Hit or Stand.

If the user Stands, the game will end, but first the dealer gets a chance to draw cards. The dealer only follows rules, without any choice. The rule is that as long as the value of the dealer's hand is less than or equal to 16, the dealer Hits (that is, takes another card). The user should see all the dealer's cards at this point. Now, the winner can be determined: If the dealer has gone over 21, the user wins. Otherwise, if the dealer's total is greater than or equal to the user's total, then the dealer wins. Otherwise, the user wins.

Two notes on programming: At any point in the subroutine, as soon as you know who the winner is, you can say "`return true ;`" or "`return false ;`" to end the subroutine and return to the main program. To avoid having an over-abundance of variables in your subroutine, remember that a method call such as `userHand.getBlackjackValue()` can be used anywhere that a number could be used, including in an output statement or in the condition of an `if` statement.

Write a main program that lets the user play several games of Blackjack. To make things interesting, give the user 100 dollars, and let the user make bets on the game. If the user loses, subtract the bet from the user's money. If the user wins, add an amount equal to the bet to the user's money. End the program when the user wants to quit or when she runs out of money.

Chapter 3

Object Oriented Programming I

Inheritance, Polymorphism, and Abstract Classes

A CLASS REPRESENTS A SET OF OBJECTS which share the same structure and behaviors. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behavior by providing the instance methods that express the behavior of the objects. This is a powerful idea. However, something like this can be done in most programming languages. The central new idea in object-oriented programming – the idea that really distinguishes it from traditional programming – is to allow classes to express the similarities among objects that share **some**, but not all, of their structure and behavior. Such similarities can be expressed using inheritance and polymorphism.

The topics covered in this section are relatively advanced aspects of object-oriented programming. Any programmer should know what is meant by subclass, inheritance, and polymorphism.

3.1 Inheritance, Polymorphism, and Abstract Classes

The term inheritance refers to the fact that one class can inherit part or all of its structure and behavior from another class.

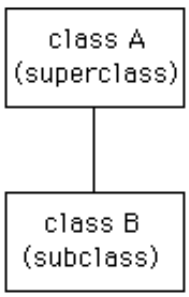


Figure 3.1: Inheritance

The class that does the inheriting is said to be a subclass of the class from which it

inherits. If class B is a subclass of class A, we also say that class A is a superclass of class B. (Sometimes the terms derived class and base class are used instead of subclass and superclass.) A subclass can add to the structure and behavior that it inherits. It can also replace or modify inherited behavior (though not inherited structure). The relationship between subclass and superclass is sometimes shown by a diagram in which the subclass is shown below, and connected to, its superclass.

In Java, when you create a new class, you can declare that it is a subclass of an existing class. If you are defining a class named "B" and you want it to be a subclass of a class named "A", you would write

```
class B extends A {
    .
    .  // additions to, and modifications of,
    .  // stuff inherited from class A
    .
}
```

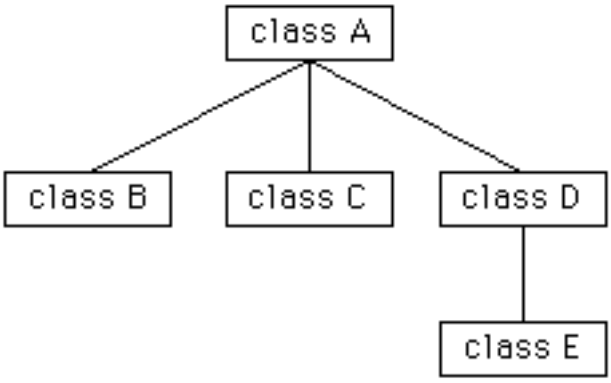


Figure 3.2: Hierarchy

Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as "sibling classes," share some structures and behaviors – namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram to the left, classes B, C, and D are sibling classes. Inheritance can also extend over several "generations" of classes. This is shown in the diagram, where class E is a subclass of class D which is itself a subclass of class A. In this case, class E is considered to be a subclass of class A, even though it is not a direct subclass.

Let's look at an example. Suppose that a program has to deal with motor vehicles, including cars, trucks, and motorcycles. (This might be a program used by a Department of Motor Vehicles to keep track of registrations.) The program could use a class named `Vehicle` to represent all types of vehicles. The `Vehicle` class could include instance variables such as `registrationNumber` and `owner` and instance methods such as `transferOwnership()`. These are variables and methods common to all vehicles. Three subclasses of `Vehicle` – `Car`, `Truck`, and `Motorcycle` – could then be used to hold variables and methods specific to particular types of vehicles. The `Car` class might add an instance variable `numberOfDoors`, the `Truck` class might have `numberOfAxels`, and the `Motorcycle` class could have a boolean variable `hasSidecar`. (Well, it could in theory at

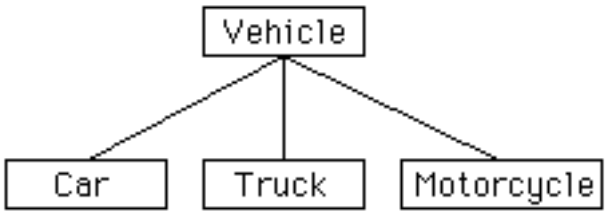


Figure 3.3: An Example Hierarchy

least, even if it might give a chuckle to the people at the Department of Motor Vehicles.) The declarations of these classes in Java program would look, in outline, like this:

```
class Vehicle {
   int registrationNumber;
   Person owner;  // (assuming that a Person class has been defined)
   void transferOwnership(Person newOwner) {
       . . .
   }
   . . .
}
class Car extends Vehicle {
   int numberOfDoors;
   . . .
}
class Truck extends Vehicle {
   int numberOfAxels;
   . . .
}
class Motorcycle extends Vehicle {
   boolean hasSidecar;
   . . .
}
```

Suppose that `myCar` is a variable of type `Car` that has been declared and initialized with the statement

```
Car myCar = new Car();
```

(Note that, as with any variable, it is OK to declare a variable and initialize it in a single statement. This is equivalent to the declaration "`Car myCar;`" followed by the assignment statement "`myCar = new Car();`".) Given this declaration, a program could refer to `myCar.numberOfDoors`, since `numberOfDoors` is an instance variable in the class `Car`. But since class `Car` extends class `Vehicle`, a car also has all the structure and behavior of a vehicle. This means that `myCar.registrationNumber`, `myCar.owner`, and `myCar.transferOwnership()` also exist.

Now, in the real world, cars, trucks, and motorcycles are in fact vehicles. The same is true in a program. That is, an object of type `Car` or `Truck` or `Motorcycle` is automatically an object of type `Vehicle`.

This brings us to the following Important Fact:

A variable that can hold a reference to an object of class A can also hold a reference to an object belonging to any subclass of A.

The practical effect of this in our example is that an object of type `Car` can be assigned to a variable of type `Vehicle`. That is, it would be legal to say `Vehicle myVehicle = myCar;` or even `Vehicle myVehicle = new Car();`

After either of these statements, the variable `myVehicle` holds a reference to a `Vehicle` object that happens to be an instance of the subclass, `Car`. The object "remembers" that it is in fact a `Car`, and not **just** a `Vehicle`. Information about the actual class of an object is stored as part of that object. It is even possible to test whether a given object belongs to a given class, using the `instanceof` operator. The test: `if (myVehicle instanceof Car) ...` determines whether the object referred to by `myVehicle` is in fact a car.

On the other hand, the assignment statement `myCar = myVehicle;` would be illegal because `myVehicle` could potentially refer to other types of vehicles that are not cars. The computer will not allow you to assign an `int` value to a variable of type `short`, because not every `int` is a `short` . Similarly, it will not allow you to assign a value of type `Vehicle` to a variable of type `Car` because not every vehicle is a car. As in the case of `int`s and `short`s, the solution here is to use type-casting. If, for some reason, you happen to know that `myVehicle` does in fact refer to a `Car`, you can use the type cast `(Car)myVehicle` to tell the computer to treat `myVehicle` as if it were actually of type `Car`. So, you could say `myCar = (Car)myVehicle;` and you could even refer to `((Car)myVehicle).numberOfDoors`. As an example of how this could be used in a program, suppose that you want to print out relevant data about a vehicle. You could say:

```
System.out.println("Vehicle Data:");
System.out.println("Registration number:  "
                              + myVehicle.registrationNumber);
if (myVehicle instanceof Car) {
   System.out.println("Type of vehicle:  Car");
   Car c;
   c = (Car)myVehicle;
   System.out.println("Number of doors:  " + c.numberOfDoors);
}
else if (myVehicle instanceof Truck) {
   System.out.println("Type of vehicle:  Truck");
   Truck t;
   t = (Truck)myVehicle;
   System.out.println("Number of axels:  " + t.numberOfAxels);
}
else if (myVehicle instanceof Motorcycle) {
   System.out.println("Type of vehicle:  Motorcycle");
   Motorcycle m;
   m = (Motorcycle)myVehicle;
   System.out.println("Has a sidecar:    " + m.hasSidecar);
}
```

Note that for object types, when the computer executes a program, it checks whether

type-casts are valid. So, for example, if `myVehicle` refers to an object of type `Truck`, then the type cast `(Car)myVehicle` will produce an error.

As another example, consider a program that deals with shapes drawn on the screen. Let's say that the shapes include rectangles, ovals, and roundrects of various colors.

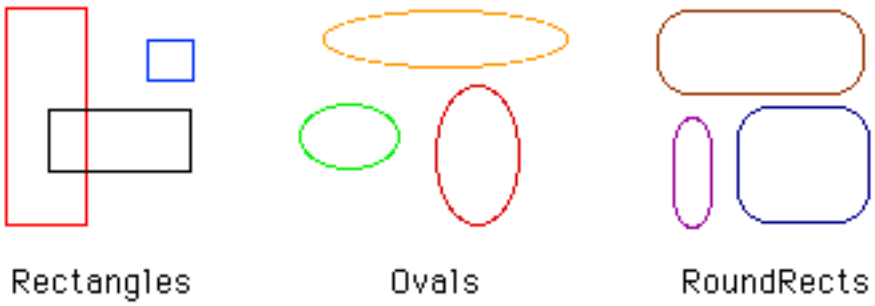


Figure 3.4: Shapes

Three classes, `Rectangle`, `Oval`, and `RoundRect`, could be used to represent the three types of shapes. These three classes would have a common superclass, `Shape`, to represent features that all three shapes have in common. The `Shape` class could include instance variables to represent the color, position, and size of a shape. It could include instance methods for changing the color, position, and size of a shape. Changing the color, for example, might involve changing the value of an instance variable, and then redrawing the shape in its new color:

```
class Shape {
    Color color;   // Color of the shape.  (Recall that class Color
                   // is defined in package java.awt.  Assume
                   // that this class has been imported.)

    void setColor(Color newColor) {
    // Method to change the color of the shape.
       color = newColor; // change value of instance variable
       redraw(); // redraw shape, which will appear in new color
    }

    void redraw() {
              // method for drawing the shape
       ? ? ?  // what commands should go here?
    }

    . . .     // more instance variables and methods

} // end of class Shape
```

Now, you might see a problem here with the method `redraw()`. The problem is that each different type of shape is drawn differently. The method `setColor()` can be called for any type of shape. How does the computer know which shape to draw when it executes the `redraw()` ? Informally, we can answer the question like this: The computer executes `redraw()` by asking the shape to redraw **itself**. Every shape object knows what it has to do to redraw itself.

In practice, this means that each of the specific shape classes has its own `redraw()` method:

```
class Rectangle extends Shape {
      void redraw() {
          . . .  // commands for drawing a rectangle
      }
      . . . //possibly, more methods and variables
}

class Oval extends Shape {
      void redraw() {
          . . .  // commands for drawing an oval
      }
      . . . // possibly, more methods and variables
}

class RoundRect extends Shape {
      void redraw() {
          . . . // commands for drawing a rounded rectangle
      }
      . . . // possibly, more methods and variables
}
```

If `oneShape` is a variable of type `Shape`, it could refer to an object of any of the types, `Rectangle`, `Oval`, or `RoundRect`. As a program executes, and the value of `oneShape` changes, it could even refer to objects of different types at different times! Whenever the statement

`oneShape.redraw();`

is executed, the redraw method that is actually called is the one appropriate for the type of object to which `oneShape` actually refers. There may be no way of telling, from looking at the text of the program, what shape this statement will draw, since it depends on the value that `oneShape` happens to have when the program is executed. Even more is true. Suppose the statement is in a loop and gets executed many times. If the value of `oneShape` changes as the loop is executed, it is possible that the very same statement "`oneShape.redraw();` " will call different methods and draw different shapes as it is executed over and over. We say that the `redraw()` method is polymorphic. A method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied. Polymorphism is one of the major distinguishing features of object-oriented programming.

Perhaps this becomes more understandable if we change our terminology a bit: In object-oriented programming, calling a method is often referred to as sending a message to an object. The object responds to the message by executing the appropriate method. The statement "`oneShape.redraw();` " is a message to the object referred to by `oneShape`. Since that object knows what type of object it is, it knows how it should respond to the message. From this point of view, the computer always executes "`oneShape.redraw();` " in the same way: by sending a message. The response to the message depends, naturally, on who receives it. From this point of view, objects are active entities that send and receive messages, and polymorphism is a natural, even necessary, part of this view. Polymorphism just means that different objects can respond to the same message in different ways.

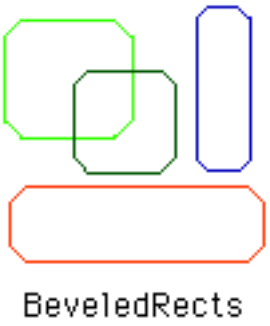


Figure 3.5: BeveledRects

One of the most beautiful things about polymorphism is that it lets code that you write do things that you didn't even conceive of, at the time you wrote it. If for some reason, I decide that I want to add beveled rectangles to the types of shapes my program can deal with, I can write a new subclass, `BeveledRect`, of class `Shape` and give it its own `redraw()` method. Automatically, code that I wrote previously – such as the statement `oneShape.redraw()` – can now suddenly start drawing beveled rectangles, even though the beveled rectangle class didn't exist when I wrote the statement!

In the statement "`oneShape.redraw();` ", the `redraw` message is sent to the object `oneShape`. Look back at the method from the `Shape` class for changing the color of a shape:

```
void setColor(Color newColor) {
   color = newColor; // change value of instance variable
   redraw(); // redraw shape, which will appear in new color
}
```

A `redraw` message is sent here, but which object is it sent to? Well, the `setColor` method is itself a message that was sent to some object. The answer is that the `redraw` message is sent to that same object, the one that received the `setColor` message. If that object is a rectangle, then it is the `redraw()` method from the `Rectangle` class that is executed. If the object is an oval, then it is the `redraw()` method from the `Oval` class. This is what you should expect, but it means that the `redraw();` statement in the `setColor()` method does **not** necessarily call the `redraw()` method in the `Shape` class! The `redraw()` method that is executed could be in any subclass of `Shape`.

Again, this is not a real surprise if you think about it in the right way. Remember that an instance method is always contained in an object. The class only contains the source code for the method. When a `Rectangle` object is created, it contains a `redraw()` method. The source code for that method is in the `Rectangle` class. The object also contains a `setColor()` method. Since the `Rectangle` class does not define a `setColor()` method, the source code for the rectangle's `setColor()` method comes from the superclass, `Shape`. But even though the source codes for the two methods are in different classes, the methods themselves are part of the same object. When the rectangle's `setColor()` method is executed and calls `redraw()`, the `redraw()` method that is executed is the one in the same object.

Whenever a `Rectangle`, `Oval`, or `RoundRect` object has to draw itself, it is the `redraw()` method in the appropriate class that is executed. This leaves open the question, What does the `redraw()` method in the `Shape` class do? How should it be defined?

The answer may be surprising: We should leave it blank! The fact is that the class `Shape` represents the abstract idea of a shape, and there is no way to draw such a thing. Only particular, concrete shapes like rectangles and ovals can be drawn. So, why should there even be a `redraw()` method in the `Shape` class? Well, it has to be there, or it would be illegal to call it in the `setColor()` method of the `Shape` class, and it would be illegal to write "`oneShape.redraw()` ;", where `oneShape` is a variable of type `Shape`. The compiler would complain that `oneShape` is a variable of type `Shape` and there's no `redraw()` method in the `Shape` class.

Nevertheless the version of `redraw()` in the `Shape` class will never be called. In fact, if you think about it, there can never be any reason to construct an actual object of type `Shape` ! You can have **variables** of type `Shape`, but the objects they refer to will always belong to one of the subclasses of `Shape`. We say that `Shape` is an abstract class. An abstract class is one that is not used to construct objects, but only as a basis for making subclasses. An abstract class exists **only** to express the common properties of all its subclasses.

Similarly, we say that the `redraw()` method in class `Shape` is an abstract method, since it is never meant to be called. In fact, there is nothing for it to do – any actual redrawing is done by `redraw()` methods in the subclasses of `Shape`. The `redraw()` method in `Shape` has to be there. But it is there only to tell the computer that all `Shapes` understand the `redraw` message. As an abstract method, it exists merely to specify the common interface of all the actual, concrete versions of `redraw()` in the subclasses of `Shape`. There is no reason for the abstract `redraw()` in class `Shape` to contain any code at all.

`Shape` and its `redraw()` method are semantically abstract. You can also tell the computer, syntactically, that they are abstract by adding the modifier "`abstract` " to their definitions. For an abstract method, the block of code that gives the implementation of an ordinary method is replaced by a semicolon. An implementation must be provided for the abstract method in any concrete subclass of the abstract class. Here's what the `Shape` class would look like as an abstract class:

```
abstract class Shape {

    Color color;   // color of shape.

    void setColor(Color newColor) {
          // method to change the color of the shape
       color = newColor; // change value of instance variable
       redraw(); // redraw shape, which will appear in new color
    }

    abstract void redraw();
          // abstract method -- must be defined in
          // concrete subclasses

    . . .          // more instance variables and methods

} // end of class Shape
```

Once you have done this, it becomes illegal to try to create actual objects of type `Shape`, and the computer will report an error if you try to do so.

In Java, every class that you declare has a superclass. If you don't specify a superclass, then the superclass is automatically taken to be `Object`, a predefined class that is part of

the package `java.lang`. (The class `Object` itself has no superclass, but it is the only class that has this property.) Thus,

`class myClass { . . .`

is exactly equivalent to

`class myClass extends Object { . . .`

Every other class is, directly or indirectly, a subclass of `Object`. This means that any object, belonging to any class whatsoever, can be assigned to a variable of type `Object`. The class `Object` represents very general properties that are shared by all objects, belonging to any class. `Object` is the most abstract class of all!

The `Object` class actually finds a use in some cases where objects of a very general sort are being manipulated. For example, java has a standard class, `java.util.ArrayList`, that represents a list of `Objects`. (The `ArrayList` class is in the package `java.util`. If you want to use this class in a program you write, you would ordinarily use an `import` statement to make it possible to use the short name, `ArrayList`, instead of the full name, `java.util.ArrayList`. `ArrayList` is discussed more fully later) The `ArrayList` class is very convenient, because an `ArrayList` can hold any number of objects, and it will grow, when necessary, as objects are added to it. Since the items in the list are of type `Object`, the list can actually hold objects of any type.

A program that wants to keep track of various `Shapes` that have been drawn on the screen can store those shapes in an `ArrayList`. Suppose that the `ArrayList` is named `listOfShapes`. A shape, `oneShape`, can be added to the end of the list by calling the instance method "`listOfShapes.add(oneShape);` ". The shape could be removed from the list with "`listOfShapes.remove(oneShape);` ". The number of shapes in the list is given by the function "`listOfShapes.size()` ". And it is possible to retrieve the i-th object from the list with the function call "`listOfShapes.get(i)` ". (Items in the list are numbered from 0 to `listOfShapes.size() - 1`.) However, note that this method returns an `Object`, not a `Shape`. (Of course, the people who wrote the `ArrayList` class didn't even know about `Shapes`, so the method they wrote could hardly have a return type of `Shape` !) Since you know that the items in the list are, in fact, `Shapes` and not just `Objects`, you can type-cast the `Object` returned by `listOfShapes.get(i)` to be a value of type `Shape` :

`oneShape = (Shape)listOfShapes.get(i);`

Let's say, for example, that you want to redraw all the shapes in the list. You could do this with a simple `for` loop, which is lovely example of object-oriented programming and polymorphism:

```
for (int i = 0; i < listOfShapes.size(); i++) {
   Shape s;  // i-th element of the list, considered as a Shape
   s = (Shape)listOfShapes.get(i);
   s.redraw();
}
```

It might be worthwhile to look at an applet that actually uses an abstract `Shape` class and an `ArrayList` to hold a list of shapes:

Applet Reference: see (Applet "ShapeDraw")

If you click one of the buttons along the bottom of this applet, a shape will be added to the screen in the upper left corner of the applet. The color of the shape is given by the "pop-up menu" in the lower right. Once a shape is on the screen, you can drag it around with the mouse. A shape will maintain the same front-to-back order with respect to other

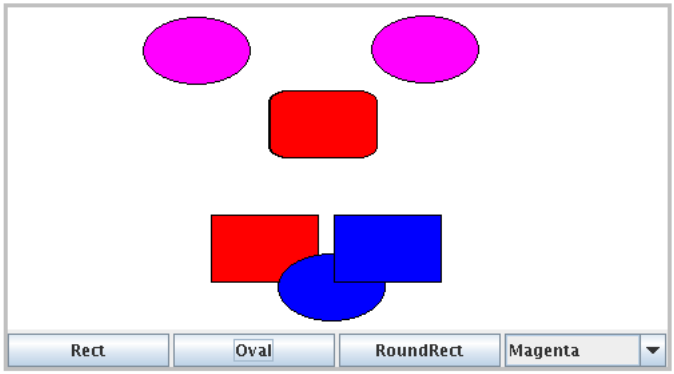


Figure 3.6: ShapeDraw Applet

shapes on the screen, even while you are dragging it. However, you can move a shape out in front of all the other shapes if you hold down the shift key as you click on it.

In this applet the only time when the actual class of a shape is used is when that shape is added to the screen. Once the shape has been created, it is manipulated entirely as an abstract shape. The routine that implements dragging, for example, works only with variables of type `Shape`. As the `Shape` is being dragged, the dragging routine just calls the `Shape's` draw method each time the shape has to be drawn, so it doesn't have to know how to draw the shape or even what type of shape it is. The object is responsible for drawing itself. If I wanted to add a new type of shape to the program, I would define a new subclass of `Shape`, add another button to the applet, and program the button to add the correct type of shape to the screen. No other changes in the programming would be necessary.

You might want to look at the source code for this applet, ShapeDraw.java, even though you won't be able to understand all of it at this time. Even the definitions of the shape classes are somewhat different from those I described in this section. (For example, the `draw()` method used in the applet has a parameter of type `Graphics`. This parameter is required because of the way Java handles all drawing.) I'll return to this example in later chapters when you know more about applets. However, it would still be worthwhile to look at the definition of the `Shape` class and its subclasses in the source code for the applet. You might also check how an `ArrayList` is used to hold a list of shapes.

3.1.1 Extending Existing Classes

We have been discussing subclasses, but so far we have dealt mostly with the theory. In the remainder of this section, I want to emphasize the practical matter of Java syntax by giving an example.

In day-to-day programming, especially for programmers who are just beginning to work with objects, subclassing is used mainly in one situation. There is an existing class that can be adapted with a few changes or additions. This is much more common than designing groups of classes and subclasses from scratch. The existing class can be extended to make a subclass. The syntax for this is

```
class subclass-name extends existing-class-name {
      .   // Changes and additions.
      .
}
```

(Of course, the class can optionally be declared to be `public`.)

As an example, suppose you want to write a program that plays the card game, Blackjack. You can use the `Card`, `Hand`, and `Deck` classes developed previously. However, a hand in the game of Blackjack is a little different from a hand of cards in general, since it must be possible to compute the "value" of a Blackjack hand according to the rules of the game. The rules are as follows: The value of a hand is obtained by adding up the values of the cards in the hand. The value of a numeric card such as a three or a ten is its numerical value. The value of a Jack, Queen, or King is 10. The value of an Ace can be either 1 or 11. An Ace should be counted as 11 unless doing so would put the total value of the hand over 21. Note that this means that the second, third, or fourth Ace in the hand will always be counted as 1.

One way to handle this is to extend the existing `Hand` class by adding a method that computes the Blackjack value of the hand. Here's the definition of such a class:

```
public class BlackjackHand extends Hand {
   public int getBlackjackValue() {
             // Returns the value of this hand for the game of Blackjack.

          int val;      // The value computed for the hand.
          boolean ace;  // Set to true if the hand contains an ace.
          int cards;    // Number of cards in the hand.

          val = 0;
          ace = false;
          cards = getCardCount();

          for ( int i = 0;  i < cards;  i++ ) {
                  // Add the value of the i-th card in the hand.
              Card card;    // The i-th card;
              int cardVal;  // The blackjack value of the i-th card.
              card = getCard(i);
              cardVal = card.getValue();  // The normal value, 1 to 13.
              if (cardVal > 10) {
                  cardVal = 10;   // For a Jack, Queen, or King.
              }
              if (cardVal == 1) {
                  ace = true;     // There is at least one ace.
              }
              val = val + cardVal;
           }
           // Now, val is the value of the hand, counting any ace as 1.
           // If there is an ace, and if changing its value from 1 to
           // 11 would leave the score less than or equal to 21,
           // then do so by adding the extra 10 points to val.
           if ( ace == true  &&  val + 10 <= 21 )
               val = val + 10;

           return val;
       }  // end getBlackjackValue()
  } // end class BlackjackHand
```

Since BlackjackHand is a subclass of Hand, an object of type BlackjackHand contains all the instance variables and instance methods defined in Hand, plus the new instance method getBlackjackValue(). For example, if bHand is a variable of type BlackjackHand, then the following are all legal method calls: bHand.getCardCount(), bHand.removeCard(0), and bHand.getBlackjackValue().

Inherited variables and methods from the Hand class can also be used in the definition of BlackjackHand (except for any that are declared to be private). The statement "cards = getCardCount(); " in the above definition of getBlackjackValue() calls the instance method getCardCount(), which was defined in the Hand class.

Extending existing classes is an easy way to build on previous work. We'll see that many standard classes have been written specifically to be used as the basis for making subclasses.

3.2 this and super

ALTHOUGH THE BASIC IDEAS of object-oriented programming are reasonably simple and clear, they are subtle, and they take time to get used to. And unfortunately, beyond the basic ideas there are a lot of details. This section and the next cover more of those annoying details. You should not necessarily master everything in these two sections the first time through, but you should read it to be aware of what is possible. For the most part, when I need to use this material later in the text, I will explain it again briefly, or I will refer you back to it. In this section, we'll look at two variables, this and super that are automatically defined in any instance method.

3.2.1 The Special Variables this and super

A static member of a class has a simple name, which can only be used inside the class definition. For use outside the class, it has a full name of the form **class-name.simple-name**. For example, "System.out " is a static member variable with simple name "out " in the class "System ". It's always legal to use the full name of a static member, even within the class where it's defined. Sometimes it's even necessary, as when the simple name of a static member variable is hidden by a local variable of the same name.

Instance variables and instance methods also have simple names. The simple name of such an instance member can be used in instance methods in the class where the instance member is defined. Instance members also have full names, but remember that instance variables and methods are actually contained in objects, not classes. The full name of an instance member has to contain a reference to the object that contains the instance member. To get at an instance variable or method from outside the class definition, you need a variable that refers to the object. Then the full name is of the form **variable-name.simple-name**. But suppose you are writing the definition of an instance method in some class. How can you get a reference to the object that contains that instance method? You might need such a reference, for example, if you want to use the full name of an instance variable, because the simple name of the instance variable is hidden by a local variable or parameter.

Java provides a special, predefined variable named "this " that you can use for such purposes. The variable, this, is used in the source code of an instance method to refer to the object that contains the method. This intent of the name, this, is to refer to "this object," the one right here that this very method is in. If x is an instance variable in the same object, then this.x can be used as a full name for that variable. If otherMethod()

is an instance method in the same object, then `this.otherMethod()` could be used to call that method. Whenever the computer executes an instance method, it automatically sets the variable, `this`, to refer to the object that contains the method.

One common use of `this` is in constructors. For example:

```
public class Student {

    private String name;  // Name of the student.

    public Student(String name) {
         // Constructor.  Create a student with specified name.
       this.name = name;
    }
      .
      .   // More variables and methods.
      .
 }
```

In the constructor, the instance variable called `name` is hidden by a formal parameter. However, the instance variable can still be referred to by its full name, `this.name`. In the assignment statement, the value of the formal parameter, `name`, is assigned to the instance variable, `this.name`. This is considered to be acceptable style: There is no need to dream up cute new names for formal parameters that are just used to initialize instance variables. You can use the same name for the parameter as for the instance variable.

There are other uses for `this`. Sometimes, when you are writing an instance method, you need to pass the object that contains the method to a subroutine, as an actual parameter. In that case, you can use `this` as the actual parameter. For example, if you wanted to print out a string representation of the object, you could say "`System.out.println(this);` ". Or you could assign the value of `this` to another variable in an assignment statement. In fact, you can do anything with `this` that you could do with any other variable, except change its value.

Java also defines another special variable, named "`super` ", for use in the definitions of instance methods. The variable `super` is for use in a subclass. Like `this`, `super` refers to the object that contains the method. But it's forgetful. It forgets that the object belongs to the class you are writing, and it remembers only that it belongs to the superclass of that class. The point is that the class can contain additions and modifications to the superclass. `super` doesn't know about any of those additions and modifications; it can only be used to refer to methods and variables in the superclass.

Let's say that the class that you are writing contains an instance method `doSomething()`. Consider the subroutine call statement `super.doSomething()`. Now, `super` doesn't know anything about the `doSomething()` method in the subclass. It only knows about things in the superclass, so it tries to execute a method named `doSomething()` from the superclass. If there is none – if the `doSomething()` method was an addition rather than a modification – you'll get a syntax error.

The reason `super` exists is so you can get access to things in the superclass that are *hidden* by things in the subclass. For example, `super.x` always refers to an instance variable named `x` in the superclass. This can be useful for the following reason: If a class contains an instance variable with the same name as an instance variable in its superclass, then an object of that class will actually contain two variables with the same name: one defined as part of the class itself and one defined as part of the superclass. The variable in the

subclass does not **replace** the variable of the same name in the superclass; it merely **hides** it. The variable from the superclass can still be accessed, using `super`.

When you write a method in a subclass that has the same signature as a method in its superclass, the method from the superclass is hidden in the same way. We say that the method in the subclass overrides the method from the superclass. Again, however, `super` can be used to access the method from the superclass.

The major use of `super` is to override a method with a new method that **extends** the behavior of the inherited method, instead of **replacing** that behavior entirely. The new method can use `super` to call the method from the superclass, and then it can add additional code to provide additional behavior. As an example, suppose you have a `PairOfDice` class that includes a `roll()` method. Suppose that you want a subclass, `GraphicalDice`, to represent a pair of dice drawn on the computer screen. The `roll()` method in the `GraphicalDice` class should do everything that the `roll()` method in the `PairOfDice` class does. We can express this with a call to `super.roll()`. But in addition to that, the `roll()` method for a `GraphicalDice` object has to redraw the dice to show the new values. The `GraphicalDice` class might look something like this:

```
public class GraphicalDice extends PairOfDice {

    public void roll() {
            // Roll the dice, and redraw them.
         super.roll();  // Call the roll method from PairOfDice.
         redraw();      // Call a method to draw the dice.
    }
       .
       .  // More stuff, including definition of redraw().
       .
}
```

Note that this allows you to extend the behavior of the `roll()` method even if you don't know how the method is implemented in the superclass!

Here is a more complete example. The `RandomBrighten applet` shows a disturbance that moves around in a mosaic of little squares. As it moves, the squares it visits become a brighter red. The result looks interesting, but I think it would be prettier if the pattern were symmetric. The symmetric applet can be programmed as an easy extension of the original applet.

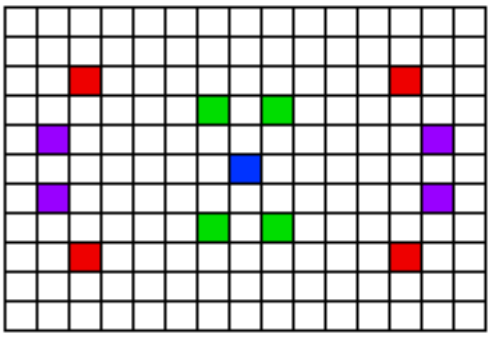


Figure 3.7: Symmetric Brighten Applet

In the symmetric version, each time a square is brightened, the squares that can be obtained from that one by horizontal and vertical reflection through the center of the mosaic are also brightened. The four red squares in the picture, for example, form a set of such symmetrically placed squares, as do the purple squares and the green squares. (The blue square is at the center of the mosaic, so reflecting it doesn't produce any other squares; it's its own reflection.)

The original applet is defined by the class `RandomBrighten`. This class uses features of Java that you won't learn about for a while yet, but the actual task of brightening a square is done by a single method called `brighten()`. If `row` and `col` are the row and column numbers of a square, then "`brighten(row,col);` " increases the brightness of that square. All we need is a subclass of `RandomBrighten` with a modified `brighten()` routine. Instead of just brightening one square, the modified routine will also brighten the horizontal and vertical reflections of that square. But how will it brighten each of the four individual squares? By calling the `brighten()` method from the original class. It can do this by calling `super.brighten()`.

There is still the problem of computing the row and column numbers of the horizontal and vertical reflections. To do this, you need to know the number of rows and the number of columns. The `RandomBrighten` class has instance variables named `ROWS` and `COLUMNS` to represent these quantities. Using these variables, it's possible to come up with formulas for the reflections, as shown in the definition of the `brighten()` method below.

Here's the complete definition of the new class:

```
public class SymmetricBrighten extends RandomBrighten {

   void brighten(int row, int col) {
         // Brighten the specified square and its horizontal
         // and vertical reflections.  This overrides the brighten
         // method from the RandomBrighten class, which just
         // brightens one square.
      super.brighten(row, col);
      super.brighten(ROWS - 1 - row, col);
      super.brighten(row, COLUMNS - 1 - col);
      super.brighten(ROWS - 1 - row, COLUMNS - 1 - col);
   }

} // end class SymmetricBrighten
```

This is the entire source code for the applet!

3.2.2 Constructors in Subclasses

Constructors are not inherited. That is, if you extend an existing class to make a subclass, the constructors in the superclass do `not` become part of the subclass. If you want constructors in the subclass, you have to define new ones from scratch. If you don't define any constructors in the subclass, then the computer will make up a default constructor, with no parameters, for you.

This could be a problem, if there is a constructor in the superclass that does a lot of necessary work. It looks like you might have to repeat all that work in the subclass! This could be a **real** problem if you don't have the source code to the superclass, and don't know how it works, or if the constructor in the superclass initializes `private` member variables that you don't even have access to in the subclass!

Obviously, there has to be some fix for this, and there is. It involves the special variable, `super`. As the very first statement in a constructor, you can use `super` to call a constructor from the superclass. The notation for this is a bit ugly and misleading, and it can only be used in this one particular circumstance: It looks like you are calling `super` as a subroutine (even though `super` is not a subroutine and you can't call constructors the same way you call other subroutines anyway). As an example, assume that the `PairOfDice` class has a constructor that takes two integers as parameters. Consider a subclass:

```
public class GraphicalDice extends PairOfDice {

          public GraphicalDice() {  // Constructor for this class.

              super(3,4);  // Call the constructor from the
                           //   PairOfDice class, with parameters 3, 4.

              initializeGraphics();  // Do some initialization specific
                                     //   to the GraphicalDice class.
          }
             .
             .  // More constructors, methods, variables...
             .
    }
```

This might seem rather technical, but unfortunately it is sometimes necessary. By the way, you can use the special variable `this` in exactly the same way to call another constructor in the same class. This can be useful since it can save you from repeating the same code in several constructors.

3.3 Summary

3.3.1 Rules for Constructors

* Constructors can use any access modifier, including private.
* The constructor name must match the name of the class.
* Constructors must not have a return type.
* If you don't explicitly provide a constructor, a default, noargument constructor is automatically generated by the compiler.
* The compiler won't generate any constructor if you've typed in any other constructor(s).
* Every constructor must have as its first statement either a call to an overloaded constructor (this()) or a call to the superclass constructor (super()).
* If you've type in a constructor, without a call to super(), the compiler will insert a noarg call to super() automatically.
* A call to super class constructor can be either a noarg call or can include arguments passed to the super constructor.

* You can provide your own no-arg constructor.

* You cannot call an instance method, or access an instance variable, until after the super constructor runs.

* You can access static variables and methods, only as part of the call to super() or this(). (Example: super(Animal.DoThings()))

* The only way a constructor can be invoked is from within another constructor—you can't write code that actually calls a constructor.

3.3.2 Overloaded and Overridden Methods

Methods can be overloaded or overridden, but constructors can be only overloaded. Overloaded methods and constructors let you use the same method name (or constructor) but with different argument lists. Overriding lets you redefine a method in a subclass, when you need new subclassspecific behavior. Overloaded methods let you reuse the same method name in a class, but with different arguments (and optionally, a different return type). Overloading a method often means you're being a little nicer to those who call your methods, because your code takes on the burden of coping with different argument types rather than forcing the caller to do conversions prior to invoking your method. The rules for overloaded methods are simple:

- Overloaded methods must change the argument list.
- Overloaded methods can change the return type.
- Overloaded methods can change the access modifier.
- A method can be overloaded in the same class or in a subclass.

Overloading Example

```
public void changeSize(int size, String name, float pattern) { }
```

The following methods are legal overloads of the `changeSize()` method:

```
public void changeSize(int size, String name) { }
public int changeSize(int size, float pattern) { }
```

Deciding which of the matching methods to invoke is based on the arguments. Overriding lets you redefine a method in a subclass, when you need new subclass specific behavior.

Rules for overridden methods

1. The argument list must exactly match that of the overridden method.
2. The return type must exactly match that of the overridden method.
3. The access level must not be more restrictive than that of the overridden method.
4. The access level can be less restrictive than that of the overridden method.

Illegal Override Code and their problems

Consider the following example:

```
public class Animal {
      public void eat() { }
}
```

Illegal Override	Problem with Override
private void eat() { }	Access modifier is more restrictive
public void eat(String food) { }	A legal overload, not an override, because the argument list changed
public String eat() { }	Not an override because of the return type, but not an overload either because there's no change in the argument list

3.3.3 Inheritance Summary

In Java you can make one class an **extension** of another. This is called **inheritance**. The classes form an **is-a** hierarchy. The advantage of inheritance is that it avoids code duplication, promotes code reuse, and improves maintenance and extendability. In fact, every class automatically extends `Object`.

Substitution/subtyping: A variable may reference objects of its declared type or any subtype of its declared type; subtype objects may be used whenever supertype objects are expected. There are times an objects may need to be **cast** back to original type. Note that the actual type of an object is forever fixed.

To create a class `Rectangle` which extends a class `Shape`, one starts with

```
class Rectangle extends Shape {
```

The class `Rectangle` then automatically has all the methods of class `Shape`, and you can add some of your own. The default constructor for `Rectangle` is automatically called at the start of the constructor for `Shape`.

The methods and the instance variables of the superclass can be accessed (if not declared as `private`). This can be done with the prefix `super`. The methods and instance variables of a class may be declared as `protected` to allow direct access only to extensions.

Overriding is providing a method with the same signature as the superclass but different body. This method takes precedence on the subclass object.

3.3.4 Casting

Recall that the **type of a variable** specifies what it can reference: it can reference any object that is of that type or a subtype of that type. It is an informational message to the compiler. On the other hand, an actual **object has a fixed type**. The type of the object that a variable references is sometimes called the **run-time type** of that variable.

A **cast** tells the compiler that it may assume that the object has the type that the cast says. If at run-time the object doesn't have the claimed type, you will get a `ClassCastException`.

Suppose class `Rectangle` extends class `Shape`:

```
Shape X;
Rectangle Y;
Y = new Rectangle();
X = Y;  // okay
X = new Rectangle(); // okay
Y = (Rectangle)X; // cast needed
```

Casting is also used for primitive data types, but this is more like a conversion. For example:

```
System.out.print((char)('A'+2))
```

prints out "C"; without the `char` cast it prints out 67 (the code for C).

For example, producing an `equals` method for the `Coord` class mentioned earlier:

```
public boolean equals(Object other){
  if(other instanceof Coord){
     Coord otherPoint = (Coord) other;
     return (this.x==otherPoint.x && this.y==otherPoint.y);
  }
  else return false;
}
```

3.3.5 Polymorphism

Polymorphism (multiple forms) is achieved in Java through over-riding methods in a sub-class. For example, suppose we have a class `Sound` that has a method `play()`. We might have sub-classes `Flute` and `Drum` that extend `Sound` and over-ride this method to provide effects specific to that instrument. Thus objects which are `Sounds` can come in many forms. Java ensures that the appropriate form of the method is used. Hence we can obtain multiple behaviors, but at the same time we can have some general methods which treat all `Sounds` the same.

For example, we could create an array of `Sounds` to represent some music. Then an iterator through the array, calling `play()` at each step, will produce the right sound at the right place. We can also create sub-classes that do not override the method—then the `play()` of `Sound` will be used.

Another example of polymorphism is our writing of a `toString()` method for (most of) our classes.

Chapter 4

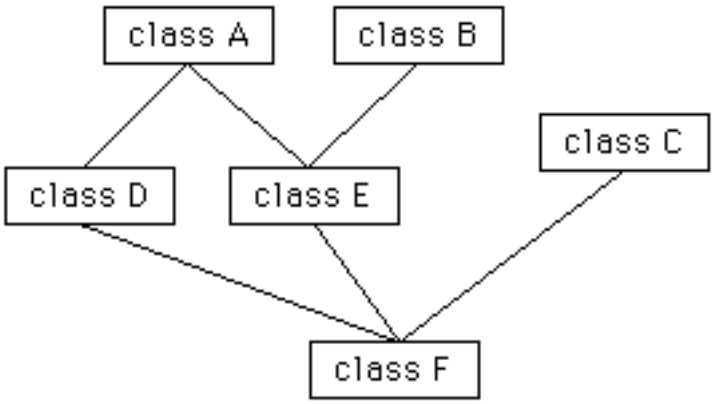
Object Oriented Programming II

Interfaces and Nested Classes

THIS SECTION simply pulls together a few more miscellaneous features of object oriented progrmming in Java. Read it now, or just look through it and refer back to it later when you need this material.

4.1 Interfaces

Some object-oriented programming languages, such as C++, allow a class to extend two or more superclasses. This is called multiple inheritance. In the illustration below, for example, class E is shown as having both class A and class B as direct superclasses, while class F has three direct superclasses.



Multiple Inheritance (NOT allowed in Java)

Such multiple inheritance is **not** allowed in Java. The designers of Java wanted to keep the language reasonably simple, and felt that the benefits of multiple inheritance were not worth the cost in increased complexity. However, Java does have a feature that can be used to accomplish many of the same goals as multiple inheritance: interfaces.

We've encountered the term "interface" before, in connection with black boxes in general and subroutines in particular. The interface of a subroutine consists of the name of

the subroutine, its return type, and the number and types of its parameters. This is the information you need to know if you want to call the subroutine. A subroutine also has an implementation: the block of code which defines it and which is executed when the subroutine is called.

In Java, `interface` is a reserved word with an additional, technical meaning. An "`interface` " in this sense consists of a set of subroutine interfaces, without any associated implementations. A class can implement an `interface` by providing an implementation for each of the subroutines specified by the interface. Here is an example of a very simple Java `interface` :

```
public interface Drawable {
   public void draw(Graphics g);
}
```

This looks much like a class definition, except that the implementation of the method `draw()` is omitted. A class that implements the `interface`, `Drawable`, must provide an implementation for this method. Of course, the class can also include other methods and variables. For example,

```
class Line implements Drawable {
    public void draw(Graphics g) {
        . . . // do something -- presumably, draw a line
    }
    . . . // other methods and variables
 }
```

Any class that implements the `Drawable` interface defines a `draw()` instance method. Any object created from such a class includes a `draw()` method. We say that an **object** implements an `interface` if it belongs to a class that implements the interface. For example, any object of type `Line` implements the `Drawable` interface. Note that it is not enough for the object to include a `draw()` method. The class that it belongs to has to *say* that it "`implements Drawable`.

While a class can **extend** only one other class, it can **implement** any number of interfaces. In fact, a class can both extend another class and implement one or more interfaces. So, we can have things like

```
class FilledCircle extends Circle
                        implements Drawable, Fillable {
   . . .
}
```

The point of all this is that, although interfaces are not classes, they are something very similar. An interface is very much like an abstract class, that is, a class that can never be used for constructing objects, but can be used as a basis for making subclasses. The subroutines in an interface are abstract methods, which must be implemented in any concrete class that implements the interface. And as with abstract classes, even though you can't construct an object from an interface, you can declare a variable whose type is given by the interface. For example, if `Drawable` is an interface, and if `Line` and `FilledCircle` are classes that implement `Drawable`, then you could say:

```
Drawable figure;  // Declare a variable of type Drawable.  It can
                  //    refer to any object that implements the
                  //    Drawable interface.

figure = new Line();  // figure now refers to an object of class Line
figure.draw(g);   // calls draw() method from class Line

figure = new FilledCircle();   // Now, figure refers to an object
                               //   of class FilledCircle.
figure.draw(g);   // calls draw() method from class FilledCircle
```

A variable of type `Drawable` can refer to any object of any class that implements the `Drawable` interface. A statement like `figure.draw(g)`, above, is legal because `figure` is of type `Drawable`, and any `Drawable` object has a `draw()` method.

Note that a type is something that can be used to declare variables. A type can also be used to specify the type of a parameter in a subroutine, or the return type of a function. In Java, a type can be either a class, an interface, or one of the eight built-in primitive types. These are the only possibilities. Of these, however, only classes can be used to construct new objects.

You are not likely to need to write your own interfaces until you get to the point of writing fairly complex programs. However, there are a few interfaces that are used in important ways in Java's standard packages. You'll learn about some of these standard interfaces in the next few chapters.

4.1.1 Interface Summary

An **interface** specifies the methods which a class should contain. A class can then **implement** an interface (actually it can implement more than one). In doing so, it must implement every method in the interface (it can have more). An interface cannot specify the behavior of the constructor. Note that an **interface** is a valid type for a variable. Example: the interface

```
interface Number {
    void increment();
    void add(Number other);
    //...ETC...
}
```

the implementation:

```
class MyInteger implements Number {
      int x;
      void increment(){x++;}
      //... ETC...
}
```

the calling program (but then one can only execute **Number**'s methods on **ticket**).

```
Number ticket = new MyInteger();
ticket.increment();
```

There are also **abstract** classes where some of the methods are implemented and so are not.

4.1.2 Interface Example: Comparable and Comparator

The `Comparable` interface is an in-built interface to enable Java to provide facilities for sorting and searching problems. The idea is that to sort a collection, be it Strings, ints, or database records, one needs a way to compare two items. Implementing the `Comparable` interface guarantees that there is a method

- `int compareTo(Object other)`

which returns one of three values: -1 if `this` comes before `other`, $+1$ if `this` comes after `other`, and 0 if the two objects are to be considered the same. (This method would usually be compatible with the `equals` method: `compareTo==0` if and only if `equals==true`.)

In-built classes such as `String` and `Integer` implement this interface. The package `java.util.Array` has sorting methods which take arrays of Objects that implement `Comparable`.

A more general idea is a `Comparator`. Suppose, for example, one wanted to sort Strings in a different way to the standard way. The idea is to define a comparator which specifies how two Strings are to be compared (with methods `compare` and `equals`). The comparator is then a parameter to the constructor of the Collection.

4.2 Nested Classes

A class seems like it should be a pretty important thing. A class is a high-level building block of a program, representing a potentially complex idea and its associated data and behaviors. I've always felt a bit silly writing tiny little classes that exist only to group a few scraps of data together. However, such trivial classes are often useful and even essential. Fortunately, in Java, I can ease the embarrassment, because one class can be nested inside another class. My trivial little class doesn't have to stand on its own. It becomes part of a larger more respectable class. This is particularly useful when you want to create a little class specifically to support the work of a larger class. And, more seriously, there are other good reasons for nesting the definition of one class inside another class.

In Java, a nested class or inner class is any class whose definition is inside the definition of another class. Inner classes can be either named or anonymous. I will come back to the topic of anonymous classes later in this section. A named inner class looks just like any other class, except that it is nested inside another class. (It can even contain further levels of nested classes, but you shouldn't carry these things too far.)

Like any other item in a class, a named inner class can be either static or non-static. A static nested class is part of the static structure of the containing class. It can be used inside that class to create objects in the usual way. If it has not been declared private, then it can also be used outside the containing class, but when it is used outside the class, its name must indicate its membership in the containing class. This is similar to other static components of a class: A static nested class is part of the class itself in the same way that static member variables are parts of the class itself.

For example, suppose a class named `WireFrameModel` represents a set of lines in three-dimensional space. (Such models are used to represent three-dimensional objects in graphics programs.) Suppose that the `WireFrameModel` class contains a static nested class, `Line`, that represents a single line. Then, outside of the class `WireFrameModel`, the `Line` class would be referred to as `WireFrameModel.Line`. Of course, this just follows the normal naming convention for static members of a class. The definition of the `WireFrameModel` class with its nested `Line` class would look, in outline, like this:

```
public class WireFrameModel {

   . . . // other members of the WireFrameModel class

   static public class Line {
         // Represents a line from the point (x1,y1,z1)
         // to the point (x2,y2,z2) in 3-dimensional space.
      double x1, y1, z1;
      double x2, y2, z2;
   } // end class Line

   . . . // other members of the WireFrameModel class

} // end WireFrameModel
```

Inside the `WireframeModel` class, a `Line` object would be created with the constructor "`new Line()` ". Outside the class, "`new WireFrameModel.Line()` " would be used.

A static nested class has full access to the members of the containing class, even to the private members. This can be another motivation for declaring a nested class, since it lets you give one class access to the private members of another class without making those members generally available to other classes.

When you compile the above class definition, two class files will be created. Even though the definition of `Line` is nested inside `WireFrameModel`, the compiled `Line` class is stored in a separate file. `WireFrameModel$Line.class` will be the name of the class file for `Line`.

Non-static nested classes are not, in practice, very different from static nested classes, but a non-static nested class is actually associated to an object rather than to the class in which it is nested. This can get some getting used to.

Any non-static member of a class is not really part of the class itself (although its source code is contained in the class definition). This is true for non-static nested classes, just as it is for any other non-static part of a class. The non-static members of a class specify what will be contained in objects that are created from that class. The same is true – at least logically – for non-static nested classes. It's as if each object that belongs to the containing class has its **own copy** of the nested class. This copy has access to all the instance methods and instance variables of the object. Two copies of the nested class in different objects differ because the instance variables and methods they refer to are in different objects. In fact, the rule for deciding whether a nested class should be static or non-static is simple: If the class needs to use any instance variable or instance method, make it non-static. Otherwise, it might as well be static.

From outside the containing class, a non-static nested class has to be referred to as **variableName.NestedClassName**, where `variableName` is a variable that refers to the object that contains the class. This is actually rather rare, however. A non-static nested class is generally used only inside the class in which it is nested, and there it can be referred to by its simple name.

In order to create an object that belongs to a non-static nested class, you must first have an object that belongs to the containing class. (When working inside the class, the object "`this` " is used implicitly.) The nested class object is permanently associated with the containing class object, and it has complete access to the members of the containing class object. Looking at an example will help, and will hopefully convince you that non-static nested classes are really very natural. Consider a class that represents poker games.

This class might include a nested class to represent the players of the game. This structure of the PokerGame class could be:

```
class PokerGame {  // Represents a game of poker.

   class Player {  // Represents one of the players in this game.
             .
             .
             .
   } // end class Player

   private Deck deck;      // A deck of cards for playing the game.
   private int pot;        // The amount of money that has been bet.
          .
          .
          .

} // end class PokerGame
```

If game is a variable of type PokerGame, then, conceptually, game contains its own copy of the Player class. In an an instance method of a PokerGame object, a new Player object would be created by saying "newPlayer() ", just as for any other class. (A Player object could be created outside the PokerGame class with the expression "newgame.Player()". Again, however, this is rather rare.) The Player object will have access to the deck and pot instance variables in the PokerGame object. Each PokerGame object has its own deck and pot and Players. Players of that poker game use the deck and pot for that game; playes of another poker game use the other game's deck and pot. That's the effect of making the Player class non-static. This is the most natural way for players to behave. A Player object represents a player of one particular poker game. If Player were a **static** nested class, on the other hand, it would represent the general idea of a poker player, independent of a particular poker game.

In some cases, you might find yourself writing a nested class and then using that class in just a single line of your program. Is it worth creating such a class? Indeed, it can be, but for cases like this you have the option of using an anonymous nested class. An anonymous class is created with a variation of the new operator that has the form

```
new  superclass-or-interface () {
     //... methods-and-variables
}
```

This constructor defines a new class, without giving it a name, and it simultaneously creates an object that belongs to that class. This form of the new operator can be used in any statement where a regular "new " could be used. The intention of this expression is to create: "a new object belonging to a class that is the same as **superclass-or-interface** but with these **methods-and-variables** added." The effect is to create a uniquely customized object, just at the point in the program where you need it. Note that it is possible to base an anonymous class on an interface, rather than a class. In this case, the anonymous class must implement the interface by defining all the methods that are declared in the interface.

Anonymous classes are most often used for handling events in graphical user interfaces, and we will encounter them several times in the next two chapters. For now, we

will look at one not-very-plausible example. Consider the `Drawable` interface, which is defined earlier. Suppose that we want a `Drawable` object that draws a filled, red, 100-pixel square. Rather than defining a separate class and then using that class to create the object, we can use an anonymous class to create the object in one statement:

```
Drawable redSquare = new Drawable() {
       void draw(Graphics g) {
          g.setColor(Color.red);
          g.fillRect(10,10,100,100);
       }
   };
```

The semicolon at the end of this statement is not part of the class definition. It's the semicolon that is required at the end of every declaration statement.

When a Java class is compiled, each anonymous nested class will produce a separate class file. If the name of the main class is `MainClass`, for example, then the names of the class files for the anonymous nested classes will be `MainClass$1.class`, `MainClass$2.class`, `MainClass$3.class`, and so on.

Chapter 5

Applets, HTML, and GUI's

JAVA IS A PROGRAMMING LANGUAGE DESIGNED for networked computers and the World Wide Web. Java applets are downloaded over a network to appear on a Web page. Part of learning Java is learning to program applets and other Graphical User Interface programs. GUI programs are event-driven. That is, user actions such as clicking on a button or pressing a key on the keyboard generate events, and the program must respond to these events as they occur.

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

This chapter covers the basics of applets, graphics, components, and events. There is also a section that covers HyperText Markup Language (HTML), the language used for writing Web pages. The discussion of applets and GUI's will continue in another next chapter with more details and with more advanced techniques.

5.1 The Basic Java Applet and JApplet

JAVA APPLETS ARE SMALL PROGRAMS that are meant to run on a page in a Web browser. Very little of that statement is completely accurate, however. An applet is not a complete program. It doesn't have to be small. And while applets are generally meant to be used on Web pages, there are other ways to use them.

An applet is inherently part of a graphical user interface. It is a type of graphical component that can be displayed in a window (whether belonging to a Web browser or to some other program). When shown in a window, an applet is a rectangular area that can contain other components, such as buttons and text boxes. It can display graphical elements such as images, rectangles, and lines. And it can respond to certain "events," such as when the user clicks on the applet with a mouse.

The AWT (Abstract Windowing Toolkit) has been part of Java from the beginning. It a toolkit (API) for creating graphical user interfaces. It provides a hierarchy of classes useful for creating and displaying windows, icons and enabling manipulation of the graphical user interface with a mouse. However, it has been clear that the AWT was not powerful or flexible enough for writing complex, sophisticated applications. The Swing graphical user interface library was created to address the problems with the AWT. The Swing library is

also a GUI toolkit and parts of it extensions of AWT library. The classes that make up the Swing library can be found in the package `javax.swing`.

You can use either the AWT or Swing libraries to create applets. The AWT version of the `Applet` class, defined in the package `java.applet`, is really only useful as a basis for making subclasses. Swing includes the class `javax.swing.JApplet` to be used as a basis for writing applets. `JApplet` is actually a subclass of `Applet`, so JApplets are in fact Applets in the usual sense.

5.1.1 Using the Applet class

An object of type `Applet` has certain basic behaviors, but doesn't actually do anything useful. It's just a blank area on the screen that doesn't respond to any events. To create a useful applet, a programmer must define a subclass that extends the `Applet` class. There are several methods in the `Applet` class that are defined to do nothing at all. The programmer must override at least some of these methods and give them something to do.

When you first learned about Java programs, you encountered the idea of a `main()` routine, which is not meant to be called by the programmer. The `main()` routine of a program is there to be called by "the system" when it needs to execute the program. The programmer writes the main routine to say what happens when the system runs the program. An applet needs no `main()` routine, since it is not a stand-alone program. However, many of the methods in an applet are similar to `main()` in that they are meant to be called by the system, and the job of the programmer is to say what happens in response to the system's calls.

One of the methods that is defined in the `Applet` class to do nothing is the `paint()` method. The `paint()` method is called by the system when the applet needs to be drawn. In a subclass of `Applet`, the `paint()` method can be redefined to draw various graphical elements such as rectangles, lines, and text on the applet. The definition of this method must have the form:

```
public void paint(Graphics g) {
    // draw some stuff
}
```

The parameter `g`, of type `Graphics`, is provided by the system when it calls the `paint()` method. In Java, all drawing of any kind is done using methods provided by a `Graphics` object. There are many such methods.

As a first example of an applet, let's go the traditional route and look at an applet that displays the string "Hello World!". We'll use the `paint()` method to display this string. The `import` statements at the beginning make it possible to use the short names `Applet` and `Graphics` instead of the full names of the classes `java.applet.Applet` and `java.awt.Graphics`.

```
import java.awt.*;
import java.applet.*;
// An applet that simply displays the string Hello World!
 public class HelloWorldApplet extends Applet {
      public void paint(Graphics g) {
            g.drawString("Hello World!", 10, 30);
      }
}  // end of class HelloWorldApplet
```

The `drawString()` method, defined in the `Graphics` class, actually does the drawing. The parameters of this method specify the string to be drawn and the point in the applet where the string is to be placed. More about this later.

Now, an applet is an object, not a class. So far we have only defined a class. Where does an actual applet object come from? It is possible, of course, to create such objects:

```
Applet hw = new HelloWorldApplet();
```

This might even be useful if you are writing a program and would like to add an applet to a window you've created. Most often, however, applet objects are created by "the system." For example, when an applet appears on a page in a Web browser, "the system" means the Web browser. It is up to the browser program to create the applet object and to add it to a Web page. The Web browser, in turn, gets instructions about what is to appear on a given Web page from the source document for that page. For an applet to appear on a Web page, the source document for that page must specify the name of the applet and its size. This specification, like the rest of the document, is written in a language called HTML. Here is some HTML code that instructs a Web browser to display a `HelloWorldApplet` :

```
<center>
<applet code="HelloWorldApplet.class" width=200 height=50>
</applet>
</center>
```

and here is the applet that this code displays:

Applet Reference: see (Applet "HelloWorldApplet")

The message is displayed in a rectangle that is 200 pixels in width and 50 pixels in height. You shouldn't be able to see the rectangle as such, since by default, an applet has a background color that is the same as the color of the Web page on which it is displayed. (This might not actually be the case in your browser.)

The Applet class defines another method that is essential for programming applets, the `init()` method. This method is called just after the applet object has been created and before it appears on the screen. Its purpose is to give the applet a chance to do any necessary initialization. Again, this method is called by the system, not by your program. Your job as a programmer is just to provide a definition of the `init()` method. The definition of the method must have the form:

```
public void init() {
   // do initialization
}
```

(You might wonder, by the way, why initialization is done in the `init()` method rather than in a constructor. In fact, it is possible to define a constructor for your applet class. To create the applet object, the system calls the constructor that has no parameters. You can write such a constructor for an applet class and can do initializations in the constructor as well as in the `init()` method. The most significant difference is that when the constructor is called, the size of the applet is not available. By the time `init()`, is called, the size is known and can be used to customize the initialization according to the size. In general, though, it is customary to do applet initialization in the `init()` method.)

Suppose, for example, that we want to change the colors used by the HelloWorldApplet. An applet has a "background color" which is used to fill the entire area of the applet before any other drawing is done, and it has a "foreground color" which is used as the

default color for drawing in the applet. It is convenient to set these colors in the `init()` method. Here is a version of the HelloWorldApplet that does this:

Applet Reference: see (Applet "HelloWorldApplet2")

Figure 5.1: HelloWorldApplet2

and here is the source code for this applet, including the `init()` method:

```
import java.awt.*;
import java.applet.*;

public class HelloWorldApplet2 extends Applet {
   public void init() {
          // Initialize the applet by setting it to use blue
          // and yellow as background and foreground colors.
      setBackground(Color.blue);
      setForeground(Color.yellow);
   }

   public void paint(Graphics g) {
      g.drawString("Hello World!", 10, 30);
   }
}  // end of class HelloWorldApplet2
```

5.1.2 Using JApplet

However, JApplets have a lot of extra structure that plain Applets don't have. Because of this structure, the painting of a JApplet is a more complex affair and is handled by the system. So, when you make a subclass of `JApplet` you should **not** write a `paint()` method for it. As we will see, if you want to draw on a JApplet, you should add a component to the applet to be used for that purpose. On the other hand, you **can** and generally should write an `init()` method for a subclass of JApplet.

Let's take a look at a simple JApplet that uses Swing. This applet demonstrates some of the basic ideas of GUI programming. Although you won't understand everything in it at this time, it will give you a preliminary idea of how things work.

GUI programs use "components" such as buttons to allow interaction with the user. Our sample applet contains a button. In fact, the button is the only thing in the applet, and it fills the entire rather small applet. Here's our sample JApplet, which is named `HelloSwing` :

Applet Reference: see (Applet "HelloSwing")

If you click this button, a new window will open with a message and an "OK" button. Click the "OK" button to dismiss the window.

Figure 5.2: Hello Swing

The button in this applet is an object that belongs to the class `JButton` (more properly, `javax.swing.JButton`). When the applet is created, the button must be created and added to the applet. This is part of the process of initializing the applet and is done in the applet's `init()` method. In this method, the button is created with the statement:

```
JButton bttn = new JButton("Click Me!");
```

The parameter to the constructor specifies the text that is displayed on the button. The button does not automatically appear on the screen. It has to be added to the applet's "content pane." This is done with the statement:

```
getContentPane().add(bttn);
```

Once it has been added to the applet, a `JButton` object mostly takes care of itself. In particular, it draws itself, so you don't have to worry about drawing it. When the user clicks the button, it generates an event. The applet (or, in fact, any object) can be programmed to respond to this event. Event-handling is the major topic in GUI programming, and I will cover it in detail later. But in outline, it works like this: The type of event generated by a button is called an `ActionEvent`. For the applet to respond to an event of this type, it must define a method

```
public void actionPerformed(ActionEvent evt) { . . . }
```

Furthermore, the button must be told that the applet will be "listening" for action events from the button. This is done by calling one of the button object's instance methods, `addActionListener()`, in the applet's `init()` method.

What should the applet do in its `actionPerformed()` method? When the user clicks the button, we want a message window to appear on the screen. Fortunately, Swing makes this easy. The class `swing.javax.JOptionPane` has a static method, `showMessageDialog()`, that can be used for this purpose, so all we have to do in `actionPerformed()` is call that method.

Given all this, you can understand a lot of what goes on in the source code for the `HelloSwing` applet. This example shows several aspects of applet programming: An `init()` method sets up the applet and adds components, the components generate events, and event-handling methods say what happens in response to those events. Here is the source code:

```
// An applet that appears on the page as a button that says
// "Click Me!".  When the button is clicked, an informational
// dialog box appears to say Hello from Swing.

import javax.swing.*;    // Swing GUI classes are defined here.
import java.awt.event.*; // Event handling class are defined here.

public class HelloSwing extends JApplet implements ActionListener {

   public void init() {
         // This method is called by the system before the applet
         // appears.  It is used here to create the button and add
         // it to the "content pane" of the JApplet.  The applet
         // is also registered as an ActionListener for the button.

      JButton bttn = new JButton("Click Me!");
      bttn.addActionListener(this);
      getContentPane().add(bttn);
   } // end init()

   public void actionPerformed(ActionEvent evt) {
         // This method is called when an action event occurs.
         // In this case, the only possible source of the event
         // is the button.  So, when this method is called, we know
         // that the button has been clicked.  Respond by showing
         // an informational dialog box.  The dialog box will
         // contain an "OK" button which the user must click to
         // dismiss the dialog box.

      String title = "Greetings";  // Shown in title bar of dialog box.
      String message = "Hello from the Swing User Interface Library.";
      JOptionPane.showMessageDialog(null, message, title,
                                    JOptionPane.INFORMATION_MESSAGE);
   } // end actionPerformed()
} // end class HelloSwing
```

In this source code, I've set up the applet itself to listen for action events from the button. Some people don't consider this to be very good style. They prefer to create a separate object to listen for and respond to events. This is more "object-oriented" in the sense that each object has its own clearly defined area of responsibility. The most convenient way to make a separate event-handling object is to use a nested anonymous class. We will see more examples of this in the future, but here, for the record, is a version of `HelloSwing` that uses an anonymous class for event handling. This applet has exactly the same behavior as the original version:

```
import javax.swing.*;    // Swing GUI classes are defined here.
import java.awt.event.*; // Event handling class are defined here.

public class HelloSwing2 extends JApplet {

   public void init() {
         // This method is called by the system before the applet
         // appears.  It is used here to create the button and add
         // it to the "content pane" of the JApplet.  An anonymous
         // class is used to create an ActionListener for the button.

      JButton bttn = new JButton("Click Me!");

      bttn.addActionListener( new ActionListener() {
               // The "action listener" for the button is defined
               // by this nested anonymous class.
            public void actionPerformed(ActionEvent evt) {
                  // This method is called to respond when the user
                  // presses the button.  It displays a message in
                  // a dialog box, along with an "OK" button which
                  // the user can click to dismiss the dialog box.
               String title = "Greetings";  // Shown in box's title bar.
               String message = "Another hello from Swing.";
               JOptionPane.showMessageDialog(null, message, title,
                                       JOptionPane.INFORMATION_MESSAGE);
            } // end actionPerformed()
         });

      getContentPane().add(bttn);

   } // end init()
} // end class HelloSwing2
```

5.2 HTML Basics

APPLETS GENERALLY APPEAR ON PAGES in a Web browser program. Such pages are themselves written in a language called HTML (HyperText Markup Language). An HTML document describes the contents of a page. A Web browser interprets the HTML code to determine what to display on the page. The HTML code doesn't look much like the resulting page that appears in the browser. The HTML document does contain all the text that appears on the page, but that text is "marked up" with commands that determine the structure and appearance of the text and determine what will appear on the page in addition to the text.

HTML has developed rapidly in the last few years, and it has become a rather complicated language. In this section, I will cover just the basics of the language. While that leaves out all the fancy stuff, it does include just about everything I've used to make the Web pages in this on-line text.

It is possible to write an HTML page using an ordinary text editor, typing in all the mark-up commands by hand. However, there are many Web-authoring programs that

make it possible to create Web pages without ever looking at the underlying code. Using these tools, you can compose a Web page in much the same way that you would write a paper with a word processor. For example, Netscape Composer, which is part of Netscape Communicator, works in this way. However, my opinion is that making high-quality Web pages still requires some work with raw HTML, and serious Web authors still need to learn the HTML language.

The mark-up commands used by HTML are called tags. An HTML tag takes the form

< **tag-name optional-modifiers** >

Where the **tag-name** is a word that specifies the command, and the **optional-modifiers**, if present, are used to provide additional information for the command (much like parameters in subroutines). A modifier takes the form

modifier-name = **value**

Usually, the **value** is enclosed in quotes, and it must be if it is more than one word long or if it contains certain special characters. There are a few modifiers which have no value, in which case only the name of the modifier is present. HTML is case insensitive, which means that you can use uppercase and lowercase letters interchangeably in tags and modifiers.

A simple example of a tag is <HR>, which draws a line – also called a "horizontal rule" – across the page. The HR tag can take several possible modifiers such as WIDTH and ALIGN. For example, the short line just after the heading of this page was produced by the HTML command:

```
<HR align=center width=''33%''>
```

The WIDTH here is specified as 33% of the available space. It could also be given as a fixed number of pixels. The value for ALIGN could be CENTER, LEFT, or RIGHT. A LEFT alignment would shove the line to the left side of the page, and a RIGHT alignment, to the right side. WIDTH and ALIGN are optional modifiers. If you leave them out, then their default values will be used. The default for WIDTH is 100%, and the default for ALIGN is LEFT.

Many tags require matching closing tags, which take the form

```
</ tag-name >
```

For example, the tag <PRE> must always have a matching closing tag </PRE> later in the document. The tag applies to everything that comes between the opening tag and the closing tag. The <PRE> tag tells a Web browser to display everything between the <PRE> and the </PRE> just as it is formatted in the original HTML source code, including all the spaces and carriage returns. (But tags between <PRE> and </PRE> are still interpreted by the browser.) "PRE" stands for preformatted text. All of the sample programs in these notes are formatted using the <PRE> command.

It is important for you to understand that when you don't use PRE, the computer will completely ignore the formatting of the text in the HTML source code. The only thing it pays attention to is the tags. Five blank lines in the source code have no more effect than one blank line or even a single blank space. Outside of <PRE>, if you want to force a new line on the Web page, you can use the tag
, which stands for "break". For example, I might give my address as:

```
David Eck<BR>
Department of Mathematics and Computer Science<BR>
Hobart and William Smith Colleges<BR>
Geneva, NY 14456<BR>
```

If you want extra vertical space in your web page, you can use several
 's in a row.

Similarly, you need a tag to indicate how the text should be broken up into paragraphs. This is done with the <P> tag, which should be placed at the beginning of every paragraph. The <P> tag has a matching </P>, which should be placed at the end of each paragraph. The closing </P> is technically optional, but it is considered good form to use it. If you want all the lines of the paragraph to be shoved over to the right, you can use <PALIGN=RIGHT> instead of <P>. (This is mostly useful when used with one short line, or when used with
 to make several short lines.) You can also use <PALIGN=CENTER> for centered lines.

By the way, if tags like <P> and <HR> have special meanings in HTML, you might wonder how I can get them to appear here on this page. To get certain special characters to appear on the page, you have to use an entity name in the HTML source code. The entity name for < is <, and the entity name for > is >. Entity names begin with & and end with a semicolon. The character & is itself a special character whose entity name is & . There are also entity names for nonstandard characters such as the accented e, é, which has the entity name é.

The rest of this page discusses several other basic HTML tags. This is not meant to be a complete discussion. But it is enough to produce interesting pages.

5.2.1 Overall Document Structure

HTML documents have a standard structure. They begin with <HTML> and end with </HTML>. Between these tags, there are two sections, the head, which is marked off by <HEAD> and </HEAD>, and the body, which – as I'm sure you have guessed – is surrounded by <BODY> and </BODY>. Often, the head contains only one item: a title for the document. This title might be shown, for example, in the title bar of a Web browser window. The title should not contain any HTML tags. The body contains the actual page contents that are displayed by the browser. So, an HTML document takes this form:

```
<HTML>
   <HEAD>
      <TITLE> page-title</TITLE>
   </HEAD>

   <BODY>

       page-contents

   </BODY>

 </HTML>
```

Web browsers are not very picky about enforcing this structure; you can probably get away with leaving out everything but the actual page contents. But it is good form to follow this structure for your pages.

The <BODY> tag can take a number of modifiers that affect the appearance of the page when it is displayed. The modifier named BGCOLOR can be used to set the background color of the page. For example,

<BODY bgcolor=white>

will ensure that the background color for the page is white. You can add modifiers to control the color of regular text (TEXT), hypertext links (LINK), and links to pages that have already been visited (VLINK). When the user clicks and holds the mouse button on a link, the link is said to be active; you can control the color of active links with the ALINK modifier. For example, how about a page with a black background, white text, blue links, red active links, and gray visited links:

<BODY bgcolor=black text=white link=blue alink=red vlink=gray>

There are several standard color names that you can use in this context, but if you want complete control, you'll have to learn how to specify colors using hexadecimal numbers. It is also possible to use an image for the background of the page, instead of a solid color. Look up the details if you are interested.

5.2.2 The Applet tag and Applet Parameters

The <APPLET> tag is 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. HEIGHT and WIDTH modifiers are required to specify the size of the applet. 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=50 WIDTH=150>
    </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 file name, not a URL.

If an applet uses a lot of.class files, it's a good idea to collect all the .class files into a single .zip or .jar file. Zip and jar files are archive files which hold a number of smaller files. Your Java development system is probably capable of creating them in some way. If your class files are in an archive, then you have to specify the name of the archive file in an ARCHIVE modifier in the <APPLET> tag. Archive files won't work on older browsers, but they should work for any browser that understands Java version 1.1 or later.

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 actually occurs in one of the PARAM tags, then getParameter returns

the associated **param-value**. If the specified paramName does not occur in any PARAM tag, then getParameter returns the value null. Parameter names are case-sensitive, so you can't use "size" in the PARAM tag and ask for "Size" in getParameter.

By the way, if you put anything besides PARAM tags between <APPLET> and </APPLET>, it will be ignored by any browser that supports Java. On the other hand, a browser that does not support Java will ignore the APPLET and PARAM tags. This means that if you put a message such as "Your browser doesn't support Java" between <APPLET> and </APPLET>, then that message will only appear in browsers that don't support Java.

Here is an example of an APPLET tag with PARAMs and some extra text for display in browsers that don't support Java:

```
<APPLET code="ShowMessage.class" WIDTH=200 HEIGHT=50>
   <PARAM NAME="message" VALUE="Goodbye World!">
   <PARAM NAME="font" VALUE="Serif">
   <PARAM NAME="size" VALUE="36">
   <p align=center>Sorry, but your browser doesn't support Java!</p>
</APPLET>
```

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

```
String display;  // Instance variable: message to be displayed.
String fontName; // Instance variable: font to use for display.

public void init() {
    String value;
    value = getParameter("message"); // Get message PARAM, if any.
    if (value == null)
       display = "Hello World!";  // default value
    else
       display = value;  // Value from PARAM tag.
    value = getParameter("font");
    if (value == null)
       fontName = "SansSerif"
    else
       fontName = value;
     .
     .
     .
```

Dealing with the size parameter would be just a little harder, since a parameter value is always a String, and the size is supposed to be an int. This means that the String value must somehow be converted to an int. We'll worry about how to do that later.

5.3 Graphics and Painting

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

An applet is an example of a GUI component. 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 – belong to subclasses of the class `javax.swing.JComponent`. Every component is responsible for drawing itself. For example, if you want to use a standard component, you only have to add it to your applet. You don't have to worry about painting it on the screen. That will happen automatically.

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.

When painting on a plain, non-Swing Applet, the drawing is done in a `paint()` method. To do custom drawing, you have to define a subclass of `Applet` and include a `paint()` method to do the drawing. However, when it comes to Swing and JApplets, things are a little more complicated. You should not draw directly on JApplets or on other top-level Swing components. Instead, you should make a separate component to use as a drawing surface, and you should add that component to the JApplet. You will have to write a class to represent the drawing surface, so programming a JApplet that does custom drawing will always involve writing at least two classes: a class for the applet itself and a class for the drawing surface. Typically, the class for the drawing surface will be defined as a subclass of `javax.swing.JPanel`, which by default is nothing but a blank area on the screen. 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 your new class, and add it to your JApplet. When the time comes for your component to be drawn on the screen, the system will call its `paintComponent()` to do the drawing. All this is not really as complicated as it might sound. We will go over this in more detail when the time comes.

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 `JComponent`. 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, each component has an instance method called `getGraphics()`. This method is a function that returns a graphics context that can be used for drawing on the component outside its `paintComponent()` method. The official line is that you should **not** do this, and I will avoid it for the most part. But I have found it convenient to use `getGraphics()` in some cases, since it can mean better performance for certain types of drawing. (Anyway, if the people who designed Java really didn't want us to use it, they shouldn't have made the `getGraphics()` method `public` !)

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 `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 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()`, this 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 when it is covered and uncovered.

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.

5.3.1 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 on the right shows a 12-by-8 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 method `getSize()`. This method returns an object that belongs to the class, `java.awt.Dimension`. A `Dimension` object has two integer instance variables,

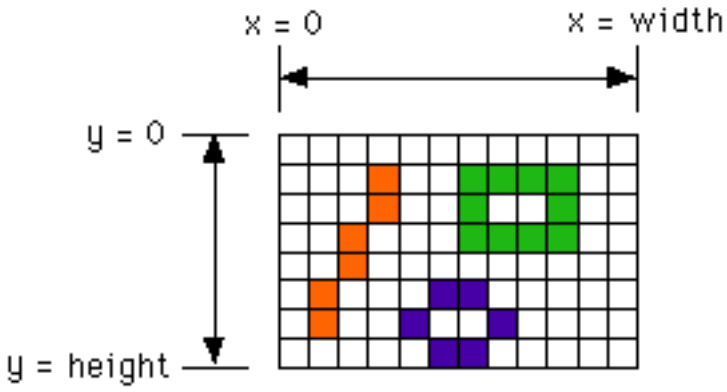


Figure 5.3: Screen Coordinates

`width` and `height`. The width of the component is `getSize().width` pixels, and its height is `getSize().height` pixels.

When you are writing an applet, you don't necessarily know the applet's size. The size of an applet is usually specified in an <APPLET> tag in the source code of a Web page, and it's easy for the Web-page author to change the specified size. In some cases, when the applet is displayed in some other kind of window instead of on a Web page, the applet can even be resized while it is running. So, it's not good form to depend on the size of the applet being set to some particular value. For other components, you have even less chance of knowing the component's size in advance. 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) {
   int width =  getSize().width;   // Find out the width of component.
   int height = getSize().height;  // Find out its height.
   . . .   // Draw the contents 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 applet.

5.3.2 Colors

Java is designed to work with more than one **colour system**. A colour system defines how a colur is represented. We'll first look at the RGB color system.

An RGB color is specified by three numbers that give the level of red, green, and blue, respectively, in that 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,

```
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. (You might 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.lightGray`, `Color.gray`, and `Color.darkGray`.

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

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

The foreground and background colors can be set by instance methods `setForeground(c)` and `setBackground(c)`, which are defined in the `Component` class and therefore are available for use with any component.

5.3.3 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". In the original Java 1.0, the font names were "TimesRoman", "Helvetica", and "Courier". You can still use the older names if you want. (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 a definite 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`. 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. It can be set with the instance method `setFont(font)`, which is defined in the `Component` class. When a graphics context is created for drawing on a component, the graphic context's current font is set equal to the font of the component.

5.3.4 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, draws the single pixel located 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, getSize().width-1, getSize().height-1);`", where `g` is a graphics context for 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, getSize().width, getSize().height);`"

- `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.

Let's use some of the material covered in this section to write a JApplet. Since we will be drawing on the applet, we will need to create a drawing surface. The drawing surface will be a JComponent belonging to a subclass of the `JPanel` class. We will define this class as a nested class inside the main applet class. All the drawing is done in the

`paintComponent()` method of the drawing surface class. I will use nested classes consistently to define drawing surfaces, although it is perfectly legal to use an independent class instead of a nested class to define the drawing surface. A nested class can be either static or non-static. In general, a non-static class must be used if it needs access to instance variables or instance methods that are defined in the main class. This will be the case in most of my examples.

The applet 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 displayed message is the string "Java!", but a different message can be specified in an applet param. The applet works OK no matter what size is specified for the applet in the <`applet`> tag. Here's the applet:

Applet Reference: see (Applet "RandomStrings")

The source for the applet is shown below. I use an instance variable called `message` to hold the message that the applet will display. There are five instance variables of type `Font` that represent different sizes and styles of text. These variables are initialized in the applet's `init()` method and are used in the drawing surface's `paintComponent()` method. I also use the `init()` method to create the drawing surface, add it to the applet, and set its background color to black.

The `paintComponent()` method for the drawing surface 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 applet. Similarly, the formula for `y` allows the string to extend beyond the top and bottom of the applet.

The drawing surface class, which is named `Display`, defines the `paintComponent()` method that draws all the strings that appear in the applet. The drawing surface is created in the applet's `init()` method as an object of type `Display`. This object is set to be the "content pane" of the applet. A JApplet's content pane fills the entire applet, except for an optional menu bar. An applet comes with a default content pane, and you can add components to that content pane. However, any JComponent can be a content pane, and in a case like this where a single component fills the applet, it makes sense to replace the content pane with the `setContentPane()` method.

`paintComponent()` in the `Display` class makes a call to `super.paintComponent(g)`. The command `super.paintComponent(g)` simply calls the `paintComponent()` method that is defined in the superclass, `JPanel`. The effect of this is to fill the component with its background color. Most `paintComponent()` methods call `super.paintComponent(g)` as the first statement, but this is not necessary if the drawing commands in the method cover the background of the component completely.

Here is the complete source code for the RandomStrings applet:

```
/* This applet displays 25 copies of a message.  The color and
   position of each message is selected at random.  The font
   of each message is randomly chosen from among five possible
   fonts.  The messages are displayed on a black background.
   Note:  This applet uses bad style, because every time
   the paintComponent() method is called, new random values are
   used.  This means that a different picture will be drawn each
   time.  This is particularly bad if only part of the applet
   needs to be redrawn, since then the applet will contain
   cut-off pieces of messages.
   When this file is compiled, it produces two classes,
   RandomStrings.class and RandomStrings$Display.class.  Both
   classes are required to use the applet.
*/

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

public class RandomStrings extends JApplet {
   String message;  // The message to be displayed.  This can be set in
                    // an applet param with name "message".  If no
                    // value is provided in the applet tag, then
                    // the string "Java!" is used as the default.
   Font font1, font2, font3, font4, font5;  // The five fonts.

   Display drawingSurface;  // This is the component on which the
                            // drawing will actually be done.  It
                            // is defined by a nested class that
                            // can be found below.

   public void init() { // Called by the system to initialize the applet.
      message = getParameter("message");
      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, 20);
      font4 = new Font("Dialog", Font.PLAIN, 30);
      font5 = new Font("Serif", Font.ITALIC, 36);

      drawingSurface = new Display();  // Create the drawing surface.
      drawingSurface.setBackground(Color.black);

      setContentPane(drawingSurface);  // Since drawingSurface will fill
                                       // the entire applet, we simply
                                       // replace the applet's content
                                       // pane with drawingSurface.
   } // end init()
```

```
    class Display extends JPanel {
         // This nested class defines a JPanel used for displaying the
         // content of the applet.  An object of this class is used as
         // the content pane of the applet.  Note that since this is a
         // nested non-static class, it has access to the instance
         // variables of the main class such as message and font1.

       public void paintComponent(Graphics g) {
          super.paintComponent(g);  // Call the paintComponent method from
                                    // the superclass, JPanel.  This simply
                                    // fills the entire component with the
                                    // component's background color.
          int width = getSize().width;   // Get this component's width.
          int height = getSize().height; // Get this component's height.

          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 nested class Display
}  // end class RandomStrings
```

5.4 Mouse Events

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

In Java, events are represented by objects. When an event occurs, the system collects all the information relevant to the event and constructs an object to contain that information. Different types of events are represented by objects belonging to different classes. For example, when the user presses one of the buttons on a mouse, an object belonging to a class called `MouseEvent` is constructed. The object contains information such as the GUI 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 subroutine. By writing that subroutine, 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 lot of processing that goes on between the time that the user presses a key or moves the mouse and the time that a subroutine in your program is called to respond to the event. Fortunately, you don't need to know much about that processing. But you should understand this much: Even though your GUI program doesn't have a `main()` routine, there is a sort of main routine running somewhere that executes a loop of the form

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

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

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. (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 RandomStrings 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.)

Every event in Java is associated with a GUI component. 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 the 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 the drawing surface of a JApplet.

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

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.*;` " 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 subroutines 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. An applet can listen for events from components that are contained in the applet. 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 nested classes to define listening objects. You will see all of these patterns in examples in this textbook.

5.4.1 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 routines – `mousePressed`, `mouseReleased`, and `mouseClicked` – will be called in that order.) In most cases, you should define `mousePressed` instead of `mouseClicked`. The `mouseEntered` and `mouseExited` methods are called when the mouse cursor enters or leaves the component. For example, if you want the component to change appearance whenever the user moves the mouse over the component, you could define these two methods.

As an example, let's look at an applet that does something when the user clicks on it. Here's an improved version of the `RandomStrings` applet. In this version, the applet will redraw itself when you click on it:

Applet Reference: see (Applet "ClickableRandomStrings")

For this version of the applet, we need to make four changes in the source code. First, add the line "`import java.awt.event.*;` " before the class definition. Second, declare that some class implements the `MouseListener` interface. If we want to use the applet itself as the listener, we would do this by saying:

```
class RandomStrings extends JApplet implements MouseListener { ...
```

Third, define the five methods of the `MouseListener` interface. Only `mousePressed` will do anything. We want to repaint the drawing surface of the applet when the user clicks the mouse. The drawing surface is represented in this applet by an instance variable named `drawingSurface`, so the `mousePressed()` needs to call `drawingSurface.repaint()` to force the drawing surface to be redrawn. The other mouse listener methods are empty. The following methods are added to the applet class definition:

```
public void mousePressed(MouseEvent evt) {
       // When user presses the mouse, tell the system to
       // call the drawing surface's paintComponent() method.
   drawingSurface.repaint();
}

// The following empty routines are required by the
// MouseListener interface:

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

Fourth and finally, the applet must be registered to listen for mouse events. Since the drawing surface fills the entire applet, it is actually the drawing surface on which the user clicks. We want the applet to listen for mouse events on the drawing surface. This can be arranged by adding this line to the applet's `init()` method:

```
drawingSurface.addMouseListener(this);
```

This calls the `addMouseListener()` method in the drawing surface object. It tells that

object where to send the mouse events that it generates. The parameter to this method is the object that will be listening for the events. In this case, the listening object is the applet itself. The special variable "`this` " is used here to refer to the applet.

We could make all these changes in the source code of the original `RandomStrings` applet. However, since we are supposed to be doing object-oriented programming, it might be instructive to write a subclass that contains the changes. This will let us build on previous work and concentrate just on the modifications. Here's the actual source code for the above applet.

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

public class ClickableRandomStrings extends RandomStrings
                                        implements MouseListener {

   public void init() {
          // When the applet is created, do the initialization
          // of the superclass, RandomStrings.  Then set this
          // applet to listen for mouse events on the
          // "drawingSurface".  (The drawingSurface variable
          // is defined in the RandomStrings class and
          // represents a component that fills the entire applet.)
      super.init();
      drawingSurface.addMouseListener(this);
   }

   public void mousePressed(MouseEvent evt) {
          // When user presses the mouse, tell the system to
          // call the drawingSurface's paintComponent() method.
      drawingSurface.repaint();
   }

   // The next four empty routines are required by the
   // MouseListener interface.

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

}  // end class ClickableRandomStrings
```

Often, when a mouse event occurs, you want to know the location of the mouse cursor. This information is available from the parameter to the event-handling method, `evt`. This parameter is an object of type `MouseEvent`, and it contains instance methods that return information about the event. To find out the coordinates of the mouse cursor, call `evt.getX()` and `evt.getY()`. These methods return integers which give the `x` and `y` coordinates where the mouse cursor was positioned. 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).

5.4.2 Anonymous Event Handlers and Adapter Classes

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 only useful 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 applet `ClickableRandomStrings` that uses an anonymous class based on `MouseAdapter` to handle mouse events:

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

public class ClickableRandomStrings2 extends RandomStrings {

   public void init() {
          // When the applet is created, do the initialization
          // of the superclass, RandomStrings.  Then add a
          // mouse listener to listen for mouse events on the
          // "drawingSurface".  (drawingSurface is defined
          // in the superclass, RandomStrings.)

      super.init();

      drawingSurface.addMouseListener( new MouseAdapter() {
            public void mousePressed(MouseEvent evt) {
               // When user presses the mouse, tell the system to
               // call the drawingSurface's paintComponent() method.
               drawingSurface.repaint();
            }
         } );
  } // end init()
}  // end class ClickableRandomStrings2
```

5.5 Introduction to Layouts and 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.

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 applet. 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. An applet's content pane is an example of a container. The standard class `JPanel`, which we have only 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. So `JPanels` are dual purpose: You can draw on them, and you can add other components to them. Because a `JPanel` is itself a component, you can add a `JPanel` to an applet's content pane or even to another `JPanel`. This makes complex nesting of components possible. `JPanels` can be used to organize complicated user interfaces.

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.

Our first example is rather simple. It's another "Hello World" applet, in which the color of the message can be changed by clicking one of the buttons at the bottom of the applet:

Applet Reference: see (Applet "HelloWorldJApplet")

Figure 5.4: HelloWorldJApplet

In the previous JApplets that we've looked at, the entire applet was filled with a JPanel that served as a drawing surface. In this example, there are two JPanels: the large black area at the top that displays the message and the smaller area at the bottom that holds the three buttons.

Let's first consider the panel that contains the buttons. This panel is created in the applet's `init()` method as a variable named `buttonBar`, of type `JPanel` :

```
JPanel buttonBar = new JPanel();
```

When a panel is to be used as a drawing surface, it is necessary to create a subclass of the `JPanel` class and include a `paintComponent()` method to do the drawing. However, when a JPanel is just being used as a container, there is no need to create a subclass. A standard JPanel is already capable of holding components of any type.

Once the panel has been created, the three buttons are created and are added to the panel. A button is just an object belonging to the class `javax.swing.JButton`. When a button is created, the text that will be shown on the button is provided as a parameter to the constructor. The first button in the panel is created with the command:

```
JButton redButton = new JButton("Red");
```

This button is added to the `buttonBar` panel with the command:

```
buttonBar.add(redButton);
```

Every JPanel comes automatically with a layout manager. This default layout manager will simply line up the components that are added to it in a row. That's exactly the behavior we want here, so there is nothing more to do. If we wanted a different kind of layout, it's possible to change the panel's layout manager.

One more step is required to make the button useful: an object must be registered with the button to listen for ActionEvents. The button will generate an ActionEvent when the user clicks on it. ActionEvents are similar to MouseEvents or KeyEvents. To use them, a class should import `java.awt.event.*`. The object that is to do the listening must implement an interface named `ActionListener`. This interface requires a definition for the method "`public void actionPerformed(ActionEvent evt);` ". Finally, the listener must be registered with the button by calling the button's `addActionListener()` method. In this case, the applet itself will act as listener, and the registration is done with the command:

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

After doing the same three commands for each of the other two buttons – and setting the background color for the sake of aesthetics – the `buttonBar` panel is ready to use. It just has to be added to the applet.

As we have seen, components are not added directly to an applet. Instead, they are added to the applet's content pane, which is itself a container. The content pane comes with a default layout manager that is capable of displaying up to five components. Four of these components are placed along the edges of the applet, in the so-called "North", "South", "East", and "West" positions. A component in the "Center" position fills in all the remaining space. This type of layout is called a BorderLayout. In our example, the button bar occupies the "South" position and the drawing area fills the "Center" position. When you add a component to a BorderLayout, you have to specify its position using a constant such as `BorderLayout.SOUTH` or `BorderLayout.CENTER`. In this example, `buttonBar` is added to the applet with the command:

```
getContentPane().add(buttonBar, BorderLayout.SOUTH);
```

The display area of the applet is a drawing surface like those we have seen in other examples. A nested class named `Display` is created as a subclass of `JPanel`, and the display area is created as an object belonging to that class. The applet class has an instance variable named `display` of type `Display` to represent the drawing surface. The display object is simply created and added to the applet with the commands:

```
        display = new Display();
        getContentPane().add(display, BorderLayout.CENTER);
```

Putting this all together, the complete `init()` method for the applet becomes:

```
public void init() {

    display = new Display();
          // The component that displays "Hello World".

    getContentPane().add(display, BorderLayout.CENTER);
          // Adds the display panel to the CENTER position of the
          // JApplet's content pane.

    JPanel buttonBar = new JPanel();
          // This panel will hold three buttons and will appear
          // at the bottom of the applet.

    buttonBar.setBackground(Color.gray);
          // Change the background color of the button panel
          // so that the buttons will stand out better.

    JButton redButton = new JButton("Red");
          // Create a new button.  "Red" is the text
          // displayed on the button.

    redButton.addActionListener(this);
          // Set up the button to send an "action event" to this applet
          // when the user clicks the button.  The parameter, this,
          // is a name for the applet object that we are creating,
          // so action events from the button will be handled by
          // calling the actionPerformed() method in this class.

    buttonBar.add(redButton);
          // Add the button to the buttonBar panel.

    JButton greenButton = new JButton("Green"); // the second button
    greenButton.addActionListener(this);
    buttonBar.add(greenButton);

    JButton blueButton = new JButton("Blue"); // the third button
    blueButton.addActionListener(this);
    buttonBar.add(blueButton);

    getContentPane().add(buttonBar, BorderLayout.SOUTH);
          // Add button panel to the bottom of the content pane.
}  // end init()
```

Notice that the variables `buttonBar`, `redButton`, `greenButton`, and `blueButton` are local to the `init()` method. This is because once the buttons and panel have been added to the applet, the variables are no longer needed. The objects continue to exist, since they

have been added to the applet. But they will take care of themselves, and there is no need to manipulate them elsewhere in the applet. The `display` variable, on the other hand, is an instance variable that can be used throughout the applet. This is because we are *not* finished with the display object after adding it to the applet. When the user clicks a button, we have to change the color of the display. We need a way to keep the variable around so that we can refer to it in the `actionPerformed()` method. In general, you don't need an instance variable for every component in an applet – just for the components that will be referred to outside the `init()` method.

The drawing surface in our example is defined by a nested class named `Display` which is a subclass of `JPanel`. The class contains a `paintComponent()` method that is responsible for drawing the message "Hello World" on a black background. The `Display` class also contains a variable that it uses to remember the current color of the message and a method that can be called to change the color. This class is more self-contained than than most of the drawing surface classes that we have looked at, and in fact it could have been defined as an independent class instead of as a nested class. Here is the definition of the nested class, `Display` :

```
class Display extends JPanel {

    // This nested class defines a component that displays
    // the string "Hello World".  The color and font for
    // the string are recorded in the variables colorNum
    // and textFont.

  int colorNum;     // Keeps track of which color is displayed;
                    //     1 for red, 2 for green, 3 for blue.

  Font textFont;    // The font in which the message is displayed.
                    // A font object represents a certain size and
                    // style of text drawn on the screen.

  Display() {
        // Constructor for the Display class.  Set the background
        // color and assign initial values to the instance
        // variables, colorNum and textFont.
     setBackground(Color.black);
     colorNum = 1;   // The color of the message is set to red.
     textFont = new Font("Serif",Font.BOLD,36);
         // Create a font object representing a big, bold font.
  }

  void setColor(int code) {
        // This method is provided to be called by the
        // main class when it wants to set the color of the
        // message.  The parameter value should be 1, 2, or 3
        // to indicate the desired color.
     colorNum = code;
     repaint(); // Tell the system to repaint this component.
  }
```

```
        public void paintComponent(Graphics g){
              // This routine is called by the system whenever this
              // panel needs to be drawn or redrawn.  It first calls
              // super.paintComponent() to fill the panel with the
              // background color.  It then displays the message
              // "Hello World" in the proper color and font.
           super.paintComponent(g);
           switch (colorNum) {          // Set the color.
              case 1:
                 g.setColor(Color.red);
                 break;
              case 2:
                 g.setColor(Color.green);
                 break;
              case 3:
                 g.setColor(Color.blue);
                 break;
           }
           g.setFont(textFont);       // Set the font.
           g.drawString("Hello World!", 25,50);    // Draw the message.
        } // end paintComponent
     } // end nested class Display
```

The main class has an instance variable named `display` of type `Display`. When the user clicks one of the buttons in the applet, this variable is used to call the `setColor()` method in the drawing surface object. This is done in the applet's `actionPerformed()` method. This method is called when the user clicks any one of the three buttons, so it needs some way to tell which button was pressed. This information is provided in the parameter to the `actionPerformed()` method. This parameter contains an "action command," which in the case of a button is just the string that is displayed on the button:

```
public void actionPerformed(ActionEvent evt) {
      // This routine is called by the system when the user clicks
      // on one of the buttons.  The response is to set the display's
      // color accordingly.

   String command = evt.getActionCommand();
            // The "action command" associated with the event
            // is the text on the button that was clicked.

   if (command.equals("Red"))       // Set the color.
      display.setColor(1);
   else if (command.equals("Green"))
      display.setColor(2);
   else if (command.equals("Blue"))
      display.setColor(3);

}  // end actionPerformed()
```

We have now looked at all the pieces of the sample applet. You can find the entire

source code in the file HelloWorldJApplet.java.

For a second example, let's look at something a little more interesting. Here's 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.) A previous section has already defined `Deck`, `Hand`, and `Card` classes that are used in this applet. 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.

Applet Reference: see (Applet "HighLowGUI")

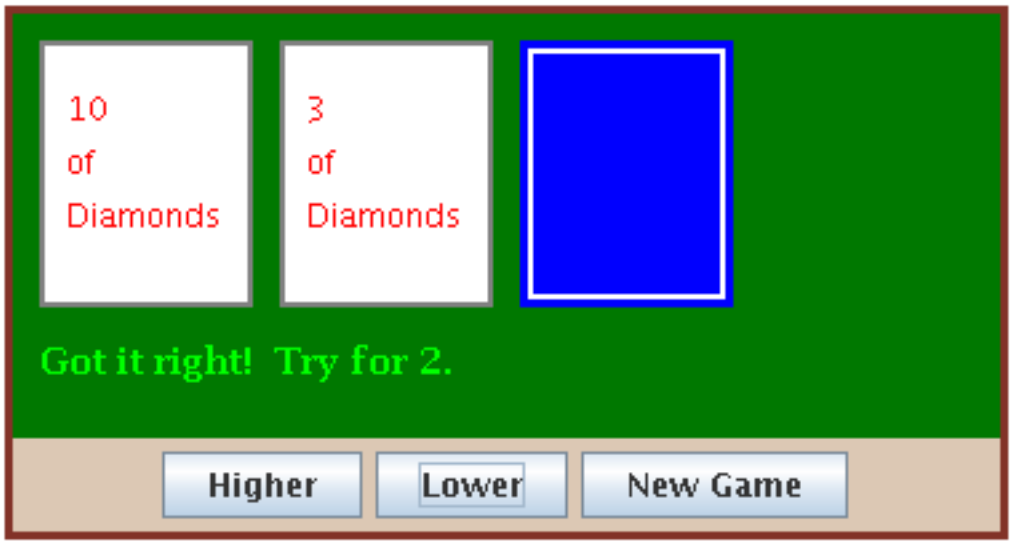


Figure 5.5: HighLowGUI Applet

The overall form of this applet is the same as that of the previous example: It has three buttons in a panel at the bottom of the applet and a large drawing surface that displays the cards and a message. However, I've organized the code a little differently in this example. In this case, it's the drawing surface object, rather than the applet, that listens for events from the buttons, and I've put almost all the programming into the display surface class. The applet object is only responsible for creating the components and adding them to the applet. This is done in the following `init()` method, which has almost the same form as the `init()` method in the previous example:

```
public void init() {

      // The init() method lays out the applet. A HighLowCanvas
      // occupies the CENTER position of the layout. On the
      // bottom is a panel that holds three buttons.  The
      // HighLowCanvas object listens for ActionEvents from the
      // buttons and does all the real work of the program.

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

   HighLowCanvas board = new HighLowCanvas();
   getContentPane().add(board, BorderLayout.CENTER);

   JPanel buttonPanel = new JPanel();
   buttonPanel.setBackground( new Color(220,200,180) );
   getContentPane().add(buttonPanel, BorderLayout.SOUTH);

   JButton higher = new JButton( "Higher" );
   higher.addActionListener(board);
   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);
}  // end init()
```

In programming the drawing surface class, `HighLowCanvas`, 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 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 canvas class uses a boolean variable named `gameInProgress` for this purpose.

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

We don't want to let the user start a new game if a game is currently in progress. That would be cheating. So, the response in the `newGame()` method is different depending on whether the state variable `gameInProgress` is true or false. If a game is in progress, the `message` instance variable should be set to 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:

```
void doNewGame() {
       // Called by the constructor, and called by actionPerformed()
       // when the user clicks the "New Game" button.  Start a new game.
   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 a deck and hand to use for this game.
   hand = new Hand();
   deck.shuffle();
   hand.addCard( deck.dealCard() );    // Deal the first card.
   message = "Is the next card higher or lower?";
   gameInProgress = true;  // State changes! A game has started.
   repaint();
}
```

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

```
void doHigher() {

         // Called by actionPerformed() when user clicks "Higher".
         // Check the user's prediction.  Game ends if user guessed
         // wrong or if the user has made three correct predictions.

   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();    // How many cards in the hand?
   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();
     }
```

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

5.6 Frames and Dialogs

APPLETS ARE A FINE IDEA. It's nice to be able to put a complete program in a rectangle on a Web page. But more serious, large-scale programs have to run in their own windows, independently of a Web browser. In Java's Swing GUI library, an independent window is represented by an object of type `JFrame`. A stand-alone GUI application can open one or more `JFrames` to provide the user interface. It is even possible for an applet to open a frame. The frame will be a separate window from the Web browser window in which the applet is running. Any frame created by an applet includes a warning message such as "Warning: Insecure Applet Window." The warning is there so that you can always recognize windows created by applets. This is just one of the security restrictions on applets, which, after all, are programs that can be downloaded automatically from Web sites that you happen to stumble across without knowing anything about them.

Here is an applet that contains just one small button. When you click this "Launch ShapeDraw" button, a `JFrame` will be opened:

Applet Reference: see (Applet "ShapeDrawLauncher")

The window in this example belongs to a class named `ShapeDrawFrame`, which is defined as a subclass of `JFrame`. The structure of a `JFrame` is almost identical to a `JApplet`, and the programming is almost the same. In fact, only a few changes had to be made to

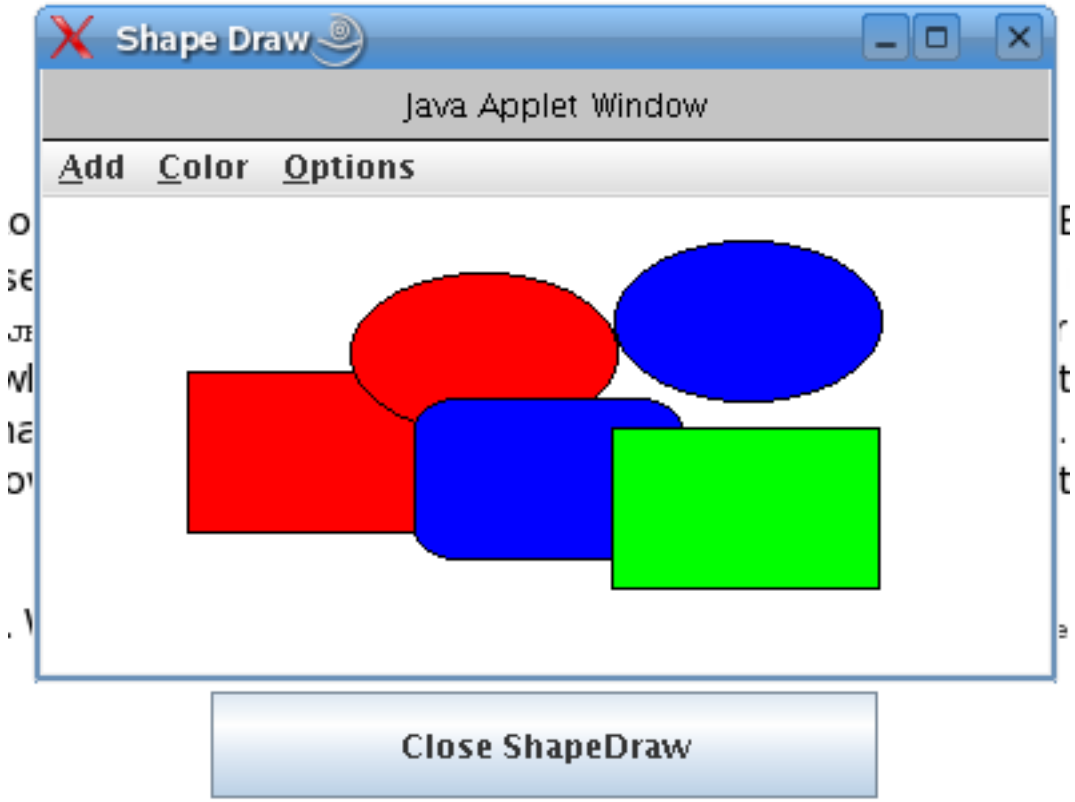


Figure 5.6: Screen Coordinates

the applet class, `ShapeDrawWithMenus`, to convert it from a `JApplet` to a `JFrame`. First, a frame does not have an `init()` method, so the initialization for `ShapeDrawFrame` is done in a constructor instead of in `init()`. In fact, the only real change necessary to convert a typical applet class into a frame class is to convert the applet's `init()` method into a constructor, and to add a bit of frame-specific initialization to the constructor. Everything that you learned about creating and programming GUI components and adding them to a content pane applies to frames as well. Menus are also handled in exactly the same way. So, we really only need to look at the additional programming that is necessary for frames.

One significant difference is that the size and location of an applet are determined externally to the applet, by the HTML code for a Web page and by the browser that displays the page. The size of a frame, on the other hand, has to be set by the frame itself or by the program that creates the frame. Often, the size is set in the frame's constructor. (If you forget to do this, the frame will have size zero and all you will see on the screen is a tiny border.) There are two methods in the `JFrame` class that can be used to set the size of a frame:

```
void setSize(int width, int height); and void pack();
```

Use `setSize()` if you know exactly what size the frame should be. The `pack()` method is more interesting. It should only be called after all GUI components have been added to the frame. Calling `pack()` will make the frame just big enough to hold all the components. It will determine the size of the frame by checking the preferred size of each of the components that it contains. (When you create your own drawing surface or custom

component, you can set its preferred size by calling its `setPreferredSize()` method or by defininining a `getPreferredSize()` method to compute the size.)

You can also set the location of the frame. This can be done with by calling the `JFrame` method:

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

The parameters, `x` and `y` give the position of the upper left corner of the frame on the screen. They are given in terms of pixel coordinates on the screen as a whole, where the upper left corner of the screen is (0,0).

Creating a frame object does not automatically make a window appear on the screen. Initially, the window is invisible, You must make it visible by calling its method

```
void show();
```

This can be called at the end of the frame's constructor, but it can also be reasonable to leave it out. In that case, the constructor will produce an invisible window, and the program that creates the frame is responsible for calling its `show()` method.

A frame has a title, a string that appears in the title bar at the top of the window. This title can be provided as an argument to the constructor or it can be set by calling the method:

```
void setTitle(String title);
```

(In the `ShapeDrawFrame` class, I set the title of the frame to be "Shape Draw". I do this by calling a constructor in the superclass with the command:

```
super("Shape Draw");
```

at the beginning of the `ShapeDrawFrame` constructor.)

Now you know how to get a frame onto the screen and how to give it a title. There is still the matter of getting rid of the frame. You can hide a frame by calling its `hide()` method. If you do this, you can make it visible again by calling `show()`. If you are completely finished with the window, you can call its `dispose()` method to close the window and free the system resources that it uses. After calling `dispose()`, you should not use the frame object again. You also have to be prepared for the fact that the user can click on the window's close box to indicate that it should be closed. By default, the system will hide the window when the user clicks its close box. However, you can tell the system to handle this event differently. Sometimes, it makes more sense to dispose of the window or even to call `System.exit()` and end the program entirely. It is even possible, as we will see below, to set up a listener to listen for the event, and to program any response you want. You can set the default response to a click in a frame's close box by calling:

```
void setDefaultCloseOperation(int operation);
```

where the parameter, `operation` is one of the constants

- `JFrame.HIDE_ON_CLOSE` – just hide the window, so it can be opened again
- `JFrame.DISPOSE_ON_CLOSE` – destroy the window, but don't end the program
- `JFrame.EXIT_ON_CLOSE` – terminate the Java interpretor by calling `System.exit()`
- `JFrame.DO_NOTHING_ON_CLOSE` – no automatic response; your program is responsible for closing the window.

If a frame is opened by a main program, and if the program has only one window, it might make sense to use EXIT_ON_CLOSE. However, note that this option is illegal for a frame that is created by an applet, since an applet is not allowed to shut down the Java interpreter.

In case you are curious, here are the lines that I added to the end of the ShapeDrawFrame constructor:

```
setDefaultCloseOperation(EXIT_ON_CLOSE); setLocation(20,50);
setSize(550,420); show();
```

I also added a main() routine to the class. This means that it is possible to run ShapeDrawFrame as a stand-alone application. In a typical command-line environment, you would do this with the command:

```
java ShapeDrawFrame
```

It has been a while since we looked at a stand-alone program with a main() routine, and we have never seen a stand-alone GUI program. It's easy for a stand-alone program to use a graphical user interface – all it has to do is open a frame. Since a ShapeDrawFrame makes itself visible when it is created, it is only necessary to create the frame object with the command "newShapeDrawFrame(); ". The complete main routine in this case looks like:

```
public static void main(String[] args) { new ShapeDrawFrame(); }
```

It might look a bit unusual to call a constructor without assigning the result to a variable, but it is perfectly legal and in this case we have no need to store the value. The main routine ends as soon as the frame is created, but the frame continues to exist and the program will continue running. Since the default close operation for the frame has been set to EXIT_ON_CLOSE, the frame will close and the program will be terminated when the user clicks the close box of the window. It might seem a bit odd to have this main() routine in the same class that defines ShapeDrawFrame, and in fact it could just as easily be in a separate class. But there is no real need to create an extra class, as long as you understand what is going on. When you type "java ShapeDrawFrame " on the command line, the system looks for a main routine in the ShapeDrawFrame class and executes it. If the main routine happens to create an object belonging to the same class, it's not a problem. It's just a command to be executed.

The source code for the frame class is in the file ShapeDrawFrame.java. The applet, shown above, which opens a frame of this type is in ShapeDrawLauncher.java.

We'll look at a few more fine points of programming with frames by looking at another example. In this case, it's a frame version of the HighLowGUI2 applet. Click on the button to open the frame:

Applet Reference: see (Applet "HighLowLauncher")

The frame in this example is defined in the file HighLowFrame.java. In many ways, this example is similar to the previous one, but there are several differences. You can resize the frame in the first example by dragging its border or corner, but if you try to resize the "High Low" frame, you will find that it is impossible. A frame is resizable by default. You can make it non-resizable by calling setResizable(false). I do this in the constructor of the HighLowFrame class. Another difference shows up if you click the frame's close box. This will not simply close the window. Instead a new window will open to ask you whether

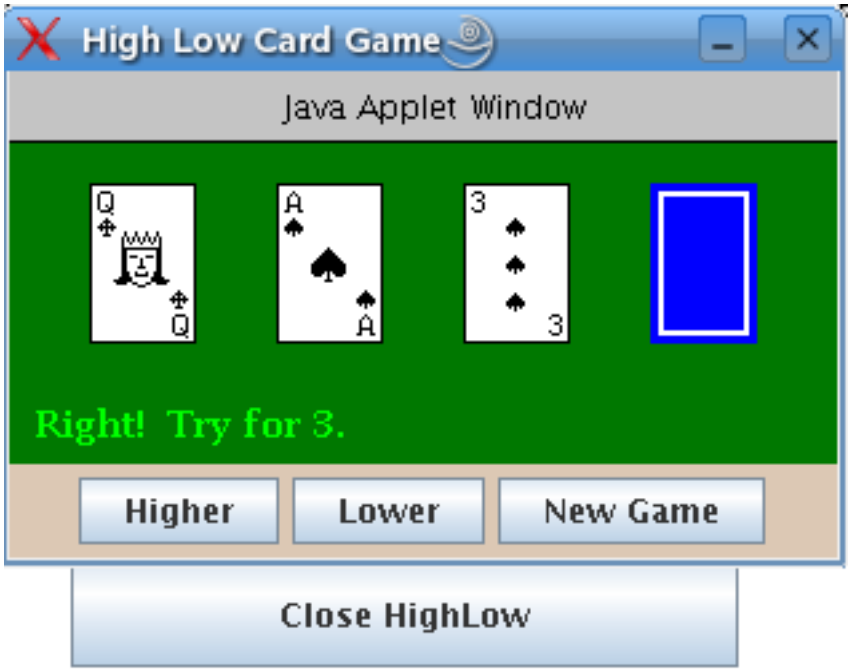


Figure 5.7: HighLowFrame Applet

you really want to close the HighLow frame. The new window is an example of a "dialog box". You have to click a button in the dialog box. If you click on "Yes", the HighLow frame will be closed; if not, the frame will remain open. (This would be more useful if, for example, you wanted to give the user a chance to save some unsaved work before closing the window.) To get this behavior, I had to turn off the system's default handling of the close box with the command:

```
setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
```

and I had to program my own response instead. I did this by registering a window listener for the frame. When certain operations are performed on a window, the window generates an event of type `WindowEvent`. You can program a response to such events in the usual way: by registering a listening object of type `WindowListener` with the frame. The `JFrame` class has an `addWindowListener()` method for this purpose. The `WindowListener` must define seven event-handling methods, including the poorly named

```
public void windowClosing(WindowEvent evt) and

public void windowClosed(WindowEvent evt)
```

The `windowClosing` event is generated when the user clicks the close box of the window. The `windowClosed` event is generated when the window is actually being disposed. The other five methods in the `WindowListener` interface are more rarely used. Fortunately, you don't have to worry about them if you use the `WindowAdapter` class. The `WindowAdapter` class implements the `WindowListener` interface, but defines all the `WindowListener` methods to be empty. You can define a `WindowListener` by creating a subclass of `WindowAdapter` and providing definitions for just those methods that you actually need. In the `HighLowFrame` class, I need a listener to respond to the `windowClosing` event that is generated when the user clicks the close box. The listener is created in the constructor using an anonymous subclass of `WindowAdapter` with a command of the form:

```
addWindowListener( new WindowAdapter() {
      // This window listener responds when the user
      // clicks the window's close box by giving the
      // user a chance to change his mind.
     public void windowClosing(WindowEvent evt) {
         .  .  //Show the dialog box, and respond to it.  .
     }
});
```

Another window listener is used in the little applet that appears on this page as the "Launch HighLow" button, above. This applet is defined in the file HighLowLauncher.java. Note that when you click on the button to open the frame, the name of the button changes to "Close HighLow". You can close the frame by clicking the button again, as well as by clicking the frame's close box. When the frame is closed, for either reason, the name of the button changes back to "Launch HighLow". The question is, how does the applet know to change the button's name, when the user closes the frame by clicking its close box? That doesn't seem to have anything to do with the applet. The trick is to use an event listener. When the applet creates the frame, it also creates a `WindowListener` and registers it with the frame. This `WindowListener` is programmed to respond to the `windowClosed` event by changing the name of the button. This is a nice example of the sort of communication between objects that can be done with events. You can check the source code to see exactly how it's done.

5.6.1 Images in Applications

Previously we saw how to load an image file into an applet. The `JApplet` class has a method, `getImage()`, that can be used for this purpose. The `JFrame` class does not provide this method, so some other technique is needed for using images in frames.

The standard class `java.awt.Toolkit` makes it possible to load an image into a standalone application. A `Toolkit` object has a `getImage()` method for reading an `Image` from an image file. To use this method, you must first obtain a `Toolkit`, and you have to do this by calling the static method `Toolkit.getDefaultToolkit()`. (Any running GUI program already has a toolkit, which is used to perform various platform-dependent functions. You don't need to construct a new toolkit. `Toolkit.getDefaultToolkit()` returns a reference to the toolkit that already exists.) Once you have a toolkit, you can use its `getImage()` method to create an `Image` object from a file. This `getImage` method takes one parameter, which specifies the name of the file. For example:

```
Toolkit toolkit = getDefaultToolkit(); Image cards =
toolkit.getImage( "smallcards.gif" );
```

or, in one line,

```
Image cards = Toolkit.getDefaultToolkit().getImage( "smallcards.gif" );
```

Once the `Image` has been created, it can be used in the same way, whether it has been created by an applet or a standalone application.

Note that an applet's `getImage()` method is used to load an image from a Web server, while a `Toolkit's getImage()` loads an image from the same computer on which the program is running. An applet cannot, in general, use a `Toolkit` to load an image, since, for security reasons, an applet is not usually allowed to read files from the computer on which it is running.

The HighLowFrame example uses an image file named "smallcards.gif" for the cards that it displays. I designed HighLowFrame with a main() routine so that it can be run as a stand-alone application. (When it is run as an application, the "smallcards.gif" file must be in the same directory with the class files.) But a HighLowFrame can also be opened by an applet, as is done in the example on this page. When it is run as an application, the image file must be loaded by a Toolkit. When it is opened by an applet, the image file must be loaded by the getImage() method of the applet. How can this be handled? I decided to do it by making the Image object a parameter to the HighLowFrame constructor. The Image must be loaded before the frame is constructed, so that it can be passed as a parameter to the constructor. The main() routine in HighLowFrame does this using a Toolkit :

```
Image cards = Toolkit.getDefaultToolkit().getImage("smallcards.gif");
HighLowFrame game = new HighLowFrame(cards);
```

The HighLowLauncher applet, on the other hand, loads the image with its own getImage() method:

```
Image cards = getImage(getCodeBase(),"smallcards.gif"); highLow = new
HighLowFrame(cards);
```

5.6.2 Dialogs

In addition to JFrame, there is another type of window in Swing. A dialog box is a type of window that is generally used for short, single purpose interactions with the user. For example, a dialog box can be used to display a message to the user, to ask the user a question, or to let the user select a color. In Swing, a dialog box is represented by an object belonging to the class JDialog.

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

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

Aside from having a parent, a JDialog can be created and used in the same way as a JFrame. However, we will not be using JDialog directly. Swing has many convenient methods for creating many common types of dialog boxes. For example, the JColorChooser class has the static method:

```
Color JColorChooser.showDialog(JFrame parent, Color initialColor)
```

When you call this method, a dialog box appears that allows the user to select a color. The first parameter specifies the parent of the dialog, or it can be null. When the dialog first appears, initialColor is selected. The dialog has a sophisticated interface that allows the user to change the selection. When the user presses an "OK" button, the dialog box closes and the selected color is returned as the value of the method. The user can also click a "Cancel" button or close the dialog box in some other way; in that case, null is

returned as the value of the method. By using this predefined color chooser dialog, you can write one line of code that will let the user select an arbitrary color.

The following applet demonstrates a `JColorChooser` dialog and three other, simpler standard dialog boxes. When you click one of the buttons, a dialog box appears. The label at the top of the applet gives you some feedback about what is happening:

Applet Reference: see (Applet "SimpleDialogDemo")

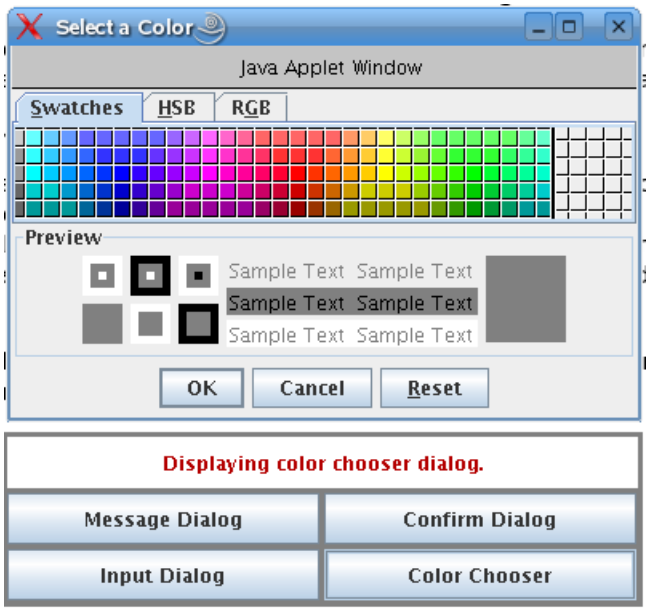


Figure 5.8: SimpleDialogDemo Applet

The three simple dialogs in this applet 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 method:

```
void JOptionPane.showMessageDialog(JFrame parent, String message)
```

The parent, as usual, can be `null`. 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(JFrame parent, String question)
```

Again, the parent can be `null`, and the question can include newline characters. The dialog box will contain an input box and 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(JFrame parent, 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 user closed the dialog in some other way.

I use a confirm dialog in the `HighLowFrame` example, earlier on this page. When the user clicks the close box of a `HighLowFrame`, I display a confirm dialog to ask whether the user really wants to quit. The frame will close if the return value is `JOptionPane.YES_OPTION`.

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

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

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

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.

5.7 Quiz Questions

THIS PAGE CONTAINS A SAMPLE quiz on material from Chapter 6 of this on-line Java textbook. You should be able to answer these questions after studying that chapter.

Question 1: Programs written for a graphical user interface have to deal with "events." Explain what is meant by the term *event*. Give at least two different examples of events, and discuss how a program might respond to those events.

Question 2: What is an *event loop?*

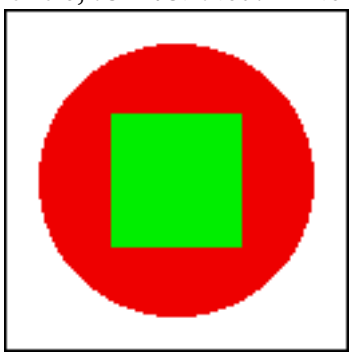
Question 3: Explain carefully what the `repaint()` method does.

Question 4: What is HTML?

Question 5: Draw the picture that will be produced by the following `paint()` method:

```
public static void paint(Graphics g) {
   for (int i=10; i <= 210; i = i + 50)
      for (int j = 10; j <= 210; j = j + 50)
         g.drawLine(i,10,j,60);
}
```

Question 6: Suppose you would like an applet that displays a green square inside a red circle, as illustrated. Write a `paint()` method that will draw the image.



Question 7: Suppose that you are writing an applet, and you want the applet to respond in some way when the user clicks the mouse on the applet. What are the four things you need to remember to put into the source code of your applet?

Question 8: Java has a standard class called `MouseEventb`. What is the purpose of this class? What does an object of type `MouseEvent` do?

Question 9: Explain what is meant by *input focus.* How is the input focus managed in a Java GUI program?

Question 10: Java has a standard class called `JPanel`. Discuss *two* ways in which JPanels can be used.

5.8 Programming Exercises

1. Write an applet that shows a pair of dice. When the user clicks on the applet, the dice should be rolled (that is, the dice should be assigned newly computed random values). Each die should be drawn as a square showing from 1 to 6 dots. Since you have to draw two dice, its a good idea to write a subroutine, "`void drawDie(Graphics g, int val, int x, int y)` ", to draw a die at the specified `(x,y)` coordinates. The second parameter, `val`, specifes the value that is showing on the die. Assume that the size of the applet is 100 by 100 pixels.

2. Improve your dice applet from the previous exercise so that it also responds to keyboard input. When the applet has the input focus, it should be hilited with a colored border, and the dice should be rolled whenever the user presses a key on the keyboard. This is in addition to rolling them when the user clicks the mouse on the applet.

3. In a previous exercise you wrote a pair-of-dice applet where the dice are rolled when the clicks on the applet. Now make a pair-of-dice applet that uses the methods discussed. Draw the dice on a JPanel, and place a "Roll" button at the bottom of the applet. The dice should be rolled when the user clicks the Roll button.

4. In Exercise 3.5, you drew a checkerboard. For this exercise, write a checkerboard applet where the user can select a square by clicking on it. Hilite the selected square by drawing a colored border around it. When the applet is first created, no square is selected. When the user clicks on a square that is not currently selected, it becomes selected. If the user clicks the square that is selected, it becomes unselected. Assume that the size of the applet is 160 by 160 pixels, so that each square on the checkerboard is 20 by 20 pixels.

5. Write an applet that shows two squares. The user should be able to drag either square with the mouse. (You'll need an instance variable to remember which square the user is dragging.) The user can drag the square off the applet if she wants; if she does this, it's gone.

6. For this exercise, you should modify the SubKiller game from Section 6.5. You can start with the existing source code, from the file SubKillerGame.java. Modify the game so it keeps track of the number of hits and misses and displays these quantities. That is, every time the depth charge blows up the sub, the number of hits goes up by one. Every time the depth charge falls off the bottom of the screen without hitting the sub, the number of misses goes up by one. There is room at the top of the applet to display these numbers. To do this exercise, you only have to add a half-dozen lines to the source code. But you have to figure out what they are and where to add them. To do this, you'll have to read the source code closely enough to understand how it works.

7. Consider the `SimpleAnimationApplet2`, a framework for writing simple animations. You can define an animation by writing a subclass and defining a `drawFrame()` method. It is possible to have the subclass implement the `MouseListener` interface. Then, you can have an animation that responds to mouse clicks.

   Write a game in which the user tries to click on a little square that jumps erratically around the applet. To implement this, use instance variables to keep track of the position of the square. In the `drawFrame()` method, there should be a certain probability that the square will jump to a new location. (You can experiment to find a probability that makes the game play well.) In your `mousePressed` method, check whether the user clicked on the square. Keep track of and display the number of times that the user hits the square and the number of times that the user misses it. Don't assume that you know the size of the applet in advance.

8. Write a Blackjack applet that lets the user play a game of Blackjack, with the computer as the dealer. The applet should draw the user's cards and the dealer's cards, just as was done for the graphical HighLow card game. You can use the source code for that game, HighLowGUI.java, for some ideas about how to write your Blackjack game. The structures of the HighLow applet and the Blackjack applet are very similar. You will certainly want to use the `drawCard()` method from that applet.

   You can find a description of the game of Blackjack in Exercise 5.5. Add the following rule to that description: If a player takes five cards without going over 21, that player wins immediately. This rule is used in some casinos. For your applet, it means that you only have to allow room for five cards. You should assume that your applet is just wide enough to show five cards, and that it is tall enough to show the user's hand and the dealer's hand.

   Note that the design of a GUI Blackjack game is very different from the design of the text-oriented program that you wrote for Exercise 5.5. The user should play the game by clicking on "Hit" and "Stand" buttons. There should be a "New Game" button that can be used to start another game after one game ends. You have to decide what happens when each of these buttons is pressed. You don't have much chance of getting this right unless you think in terms of the states that the game can be in and how the state can change.

   Your program will need the classes defined in Card.java, Hand.java, BlackjackHand.java, and Deck.java.

Chapter 6

Design II: Object Oriented Design

THE DIFFICULTIES INHERENT WITH THE DEVELOPMENT OF SOFTWARE has led many computer scientists to suggest that software development should be treated as an engineering activity. They argue for a disciplined approach where the software engineer uses carefully thought out methods and processes.

The term software engineering has several meanings (from wikipedia):

* As the broad term for all aspects of the practice of computer programming, as opposed to the theory of computer programming, which is called computer science;

* As the term embodying the advocacy of a specific approach to computer programming, one that urges that it be treated as an engineering profession rather than an art or a craft, and advocates the codification of recommended practices in the form of software engineering methodologies.

* Software engineering is

  (1) "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software," and

  (2) "the study of approaches as in (1)." IEEE Standard 610.12

6.1 Object-oriented Analysis and Design

A large programming project goes through a number of stages, starting with specification of the problem to be solved, followed by analysis of the problem and design of a program to solve it. Then comes coding, in which the program's design is expressed in some actual programming language. This is followed by testing and debugging of the program. After that comes a long period of maintenance, which means fixing any new problems that are found in the program and modifying it to adapt it to changing requirements. Together, these stages form what is called the software life cycle. (In the real world, the ideal of consecutive stages is seldom if ever achieved. During the analysis stage, it might turn out that the specifications are incomplete or inconsistent. A problem found during testing requires at least a brief return to the coding stage. If the problem is serious enough, it might even require a new design. Maintenance usually involves redoing some of the work from previous stages....)

Large, complex programming projects are only likely to succeed if a careful, systematic approach is adopted during all stages of the software life cycle. The systematic approach to programming, using accepted principles of good design, is called software engineering. The software engineer tries to efficiently construct programs that verifyably meet their specifications and that are easy to modify if necessary. There is a wide range of "methodologies" that can be applied to help in the systematic design of programs. (Most of these methodologies seem to involve drawing little boxes to represent program components, with labeled arrows to represent relationships among the boxes.)

We have been discussing object orientation in programming languages, which is relevant to the coding stage of program development. But there are also object-oriented methodologies for analysis and design. The question in this stage of the software life cycle is, How can one discover or invent the overall structure of a program? As an example of a rather simple object-oriented approach to analysis and design, consider this advice: Write down a description of the problem. Underline all the nouns in that description. The nouns should be considered as candidates for becoming classes or objects in the program design. Similarly, underline all the verbs. These are candidates for methods. This is your starting point. Further analysis might uncover the need for more classes and methods, and it might reveal that subclassing can be used to take advantage of similarities among classes.

This is perhaps a bit simple-minded, but the idea is clear and the general approach can be effective: Analyze the problem to discover the concepts that are involved, and create classes to represent those concepts. The design should arise from the problem itself, and you should end up with a program whose structure reflects the structure of the problem in a natural way.

6.1.1 Software Engineering Life-Cycles

A decades-long goal has been to find repeatable, predictable processes or methodologies that improve productivity and quality of software. Some try to systematize or formalize the seemingly unruly task of writing software. Others apply project management techniques to writing software. Without project management, software projects can easily be delivered late or over budget. With large numbers of software projects not meeting their expectations in terms of functionality, cost, or delivery schedule, effective project management is proving difficult.

Software engineering requires performing many tasks, notably the following, some of which may not seem to directly produce software.

- **Requirements Analysis** Extracting the requirements of a desired software product is the first task in creating it. While customers probably believe they know what the software is to do, it may require skill and experience in software engineering to recognize incomplete, ambiguous or contradictory requirements.

- **Specification** Specification is the task of precisely describing the software to be written, usually in a mathematically rigorous way. In reality, most successful specifications are written to understand and fine-tune applications that were already well-developed. Specifications are most important for external interfaces, that must remain stable.

- **Design and Architecture** Design and architecture refer to determining how software is to function in a general way without being involved in details. Usually this phase is divided into two sub-phases.

- **Coding** Reducing a design to code may be the most obvious part of the software engineering job, but it is not necessarily the largest portion.
- **Testing** Testing of parts of software, especially where code by two different engineers must work together, falls to the software engineer.
- **Documentation** An important (and often overlooked) task is documenting the internal design of software for the purpose of future maintenance and enhancement. Documentation is most important for external interfaces.
- **Maintenance** Maintaining and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software. Not only may it be necessary to add code that does not fit the original design but just determining how software works at some point after it is completed may require significant effort by a software engineer. About 2/3 of all software engineering work is maintenance, but this statistic can be misleading. A small part of that is fixing bugs. Most maintenance is extending systems to do new things, which in many ways can be considered new work. In comparison, about 2/3 of all civil engineering, architecture, and construction work is maintenance in a similar way.

6.1.2 Object Oriented design

OOP design rests on three principles:

- **Abstraction**: Ignore the details. In philosophical terminology, abstraction is the thought process wherein ideas are distanced from objects. In computer science, abstraction is a mechanism and practice to reduce and factor out details so that one can focus on few concepts at a time.

  Abstraction uses a strategy of simplification, wherein formerly concrete details are left ambiguous, vague, or undefined. [wikipedia:Abstraction]

- **Modularization**: break into pieces. A module can be defined variously, but generally must be a component of a larger system, and operate within that system independently from the operations of the other components. Modularity is the property of computer programs that measures the extent to which they have been composed out of separate parts called modules.

  Programs that have many direct interrelationships between any two random parts of the program code are less modular than programs where those relationships occur mainly at well-defined interfaces between modules.

- **Information hiding**: separate the implementation and the function. The principle of information hiding is the hiding of design decisions in a computer program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed. Protecting a design decision involves providing a stable interface which shields the remainder of the program from the implementation (the details that are most likely to change).

We strive for **responsibility-driven design**: each class should be responsible for its own data. We strive for **loose coupling**: each class is largely independent and communicates with other classes via a small well-defined interface. We strive for **cohesion**: each class performs one and only one task (for readability, reuse).

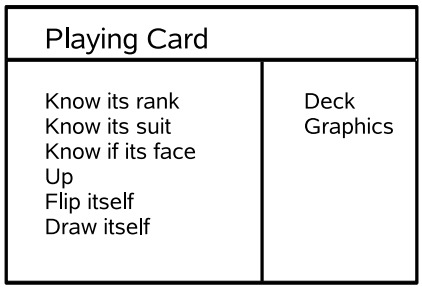
Class-Responsibility-Collaboration cards

Class-Responsibility-Collaboration cards (CRC cards) are a brainstorming tool used in the design of object-oriented software. They were proposed by Ward Cunningham. They are typically used when first determining which classes are needed and how they will interact.

CRC cards are usually created from index cards on which are written:

1. The class name.
2. The package name (if applicable).
3. The responsibilities of the class.
4. The names of other classes that the class will collaborate with to fulfill its responsibilities.

For example consider the CRC Card for a `Playing Card` class:



The responsibilities are listed on the left. The classes that the `Playing Card` class will collaborate with are listed on the right.

The idea is that class design is undertaken by a team of developers. CRC cards are used as a brainstorming technique. The team attempts to determine all the classes and their responsibilities that will be needed for the application. The team runs through various usage scenarios of the application. For e.g. one such scenario for a game of cards may be "the player picks a card from the deck and hand adds it to his hand". The team uses the CRC cards to check if this scenario can be handled by the responsibilites assigned to the classes. In this way, the design is refined until the team agrees on a set of classes and has agreed on their responsibilities.

Using a small card keeps the complexity of the design at a minimum. It focuses the designer on the essentials of the class and prevents him from getting into its details and inner workings at a time when such detail is probably counter-productive. It also forces the designer to refrain from giving the class too many responsibilities.

6.2 The Unified Modelling Language

This section will give you a quick overview of the basics of UML. It is taken from the user documentation of the UML tool `Umbrello` and wikipedia. Keep in mind that this is not a comprehensive tutorial on UML but rather a brief introduction to UML which can be read as a UML tutorial. If you would like to learn more about the Unified Modelling Language, or in general about software analysis and design, refer to one of the many books available on the topic. There are also a lot of tutorials on the Internet which you can take as a starting point.

The Unified Modelling Language (UML) is a diagramming language or notation to specify, visualize and document models of Object Oriented software systems. UML is not a development method, that means it does not tell you what to do first and what to do next or how to design your system, but it helps you to visualize your design and communicate with others. UML is controlled by the Object Management Group (OMG) and is the industry standard for graphically describing software. The OMG have recently completed version 2 of the UML standard—known as UML2.

UML is designed for Object Oriented software design and has limited use for other programming paradigms.

UML is not a method by itself, however it was designed to be compatible with the leading object-oriented software development methods of its time (e.g., OMT, Booch, Objectory). Since UML has evolved, some of these methods have been recast to take advantage of the new notation (e.g., OMT) and new methods have been created based on UML. Most well known is the Rational Unified Process (RUP) created by the Rational Software Corporation.

Modelling

There are three prominent parts of a system's model:

- Functional Model

  Showcases the functionality of the system from the user's Point of View. Includes Use Case Diagrams.

- Object Model

  Showcases the structure and substructure of the system using objects, attributes, operations, and associations. Includes Class Diagrams.

- Dynamic Model

  Showcases the internal behavior of the system. Includes Sequence Diagrams, Activity Diagrams and State Machine Diagrams.

UML is composed of many model elements that represent the different parts of a software system. The UML elements are used to create diagrams, which represent a certain part, or a point of view of the system. In UML 2.0 there are 13 types of diagrams. Some of the more important diagrams are:

- *Use Case Diagrams* show actors (people or other users of the system), use cases (the scenarios when they use the system), and their relationships
- *Class Diagrams* show classes and the relationships between them
- *Sequence Diagrams* show objects and a sequence of method calls they make to other objects.
- *Collaboration Diagrams* show objects and their relationship, putting emphasis on the objects that participate in the message exchange
- *State Diagrams* show states, state changes and events in an object or a part of the system
- *Activity Diagrams* show activities and the changes from one activity to another with the events occurring in some part of the system

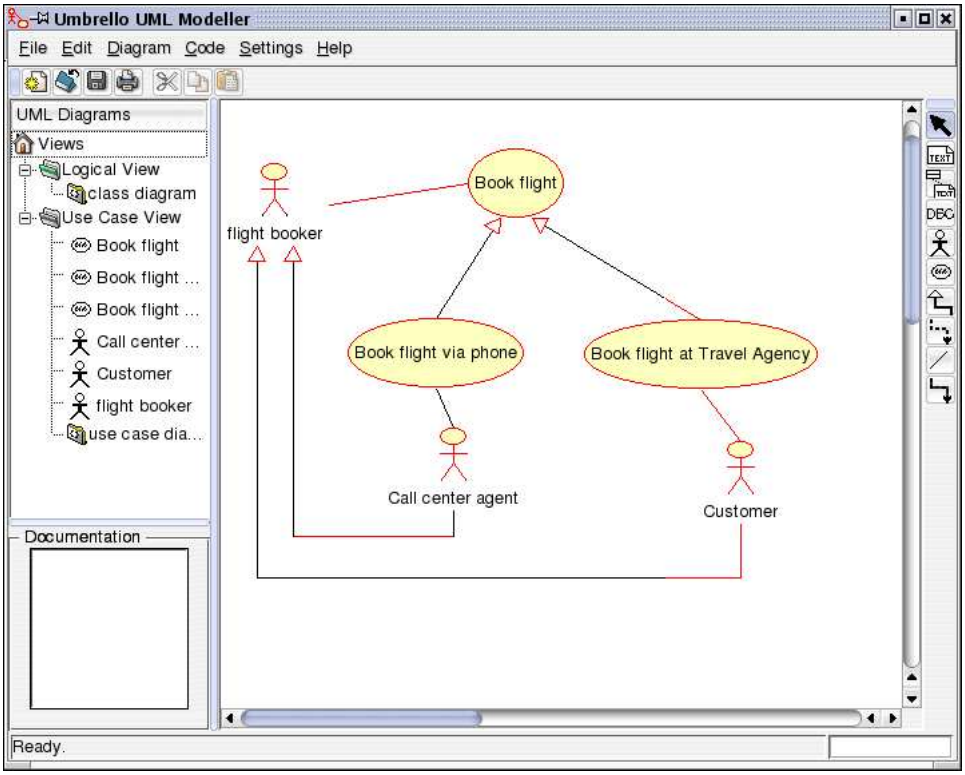


Figure 6.1: Umbrello UML Modeller showing a Use Case Diagram

- *Component Diagrams* show the high level programming components (such as KParts or Java Beans).
- *Deployment Diagrams* show the instances of the components and their relationships.

Use Case Diagrams

Use Case Diagrams describe the relationships and dependencies between a group of **Use Cases** and the Actors participating in the process.

It is important to notice that Use Case Diagrams are not suited to represent the design, and cannot describe the internals of a system. Use Case Diagrams are meant to facilitate the communication with the future users of the system, and with the customer, and are specially helpful to determine the required features the system is to have. Use Case Diagrams tell, *what* the system should do but do not–and cannot–specify *how* this is to be achieved.

A **Use Case** describes–from the point of view of the actors–a group of activities in a system that produces a concrete, tangible result.

Use Cases are descriptions of the typical interactions between the users of a system and the system itself. They represent the external interface of the system and specify a form of requirements of what the system has to do (remember, only what, not how).

When working with Use Cases, it is important to remember some simple rules:

- Each Use Case is related to at least one actor
- Each Use Case has an initiator (i.e. an actor)
- Each Use Case leads to a relevant result (a result with a business value)

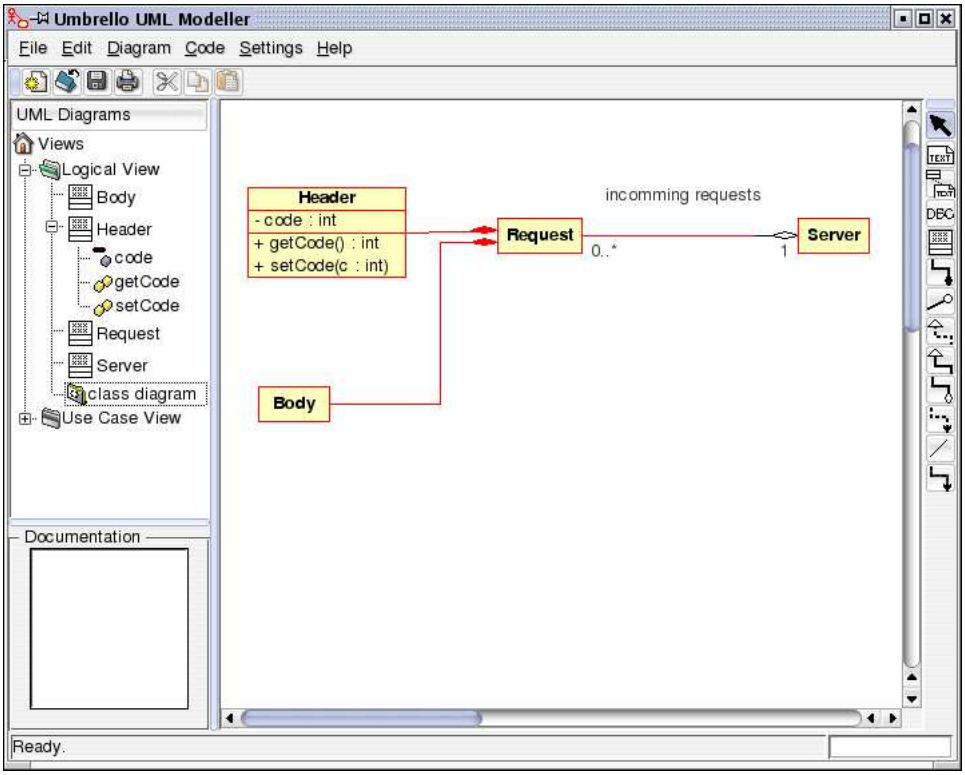


Figure 6.2: Umbrello UML Modeller showing a Class Diagram

An *a*ctor is an external entity (outside of the system) that interacts with the system by participating (and often initiating) a Use Case. Actors can be in real life people (for example users of the system), other computer systems or external events.

Actors do not represent the *physical* people or systems, but their *role* . This means that when a person interacts with the system in different ways (assuming different roles) he will be represented by several actors. For example a person that gives customer support by the telephone and takes orders from the customer into the system would be represented by an actor “Support Staff” and an actor “Sales Representative”

*U*se Case Descriptions are textual narratives of the Use Case. They usually take the form of a note or a document that is somehow linked to the Use Case, and explains the processes or activities that take place in the Use Case.

Class Diagrams

Class Diagrams show the different classes that make up a system and how they relate to each other. Class Diagrams are said to be ”static” diagrams because they show the classes, along with their methods and attributes as well as the static relationships between them: which classes ”know” about which classes or which classes ”are part” of another class, but do not show the method calls between them.

A **Class** defines the attributes and the methods of a set of objects. All objects of this class (instances of this class) share the same behavior, and have the same set of attributes (each object has its own set). The term ”Type” is sometimes used instead of Class, but it is important to mention that these two are not the same, and Type is a more general term.

In UML, Classes are represented by rectangles, with the name of the class, and can also show the attributes and operations of the class in two other ”compartments” inside

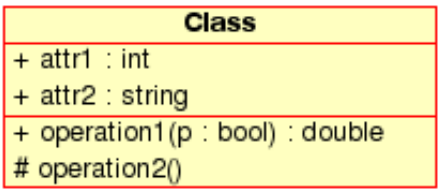


Figure 6.3: Visual representation of a Class in UML

the rectangle.

In UML, **Attributes** are shown with at least their name, and can also show their type, initial value and other properties. Attributes can also be displayed with their visibility:

- + Stands for *public* attributes
- # Stands for *protected* attributes
- - Stands for *private* attributes

Operations (methods) are also displayed with at least their name, and can also show their parameters and return types. Operations can, just as Attributes, display their visibility:

- + Stands for *public* operations
- # Stands for *protected* operations
- - Stands for *private* operations

Class Associations

Classes can relate (be associated with) to each other in different ways:

Inheritance is one of the fundamental concepts of Object Orientated programming, in which a class ”gains” all of the attributes and operations of the class it inherits from, and can override/modify some of them, as well as add more attributes and operations of its own.

- **Generalization** In UML, a *Generalization* association between two classes puts them in a hierarchy representing the concept of inheritance of a derived class from a base class. In UML, Generalizations are represented by a line connecting the two classes, with an arrow on the side of the base class.

- **Association** An association represents a relationship between classes, and gives the common semantics and structure for many types of ”connections” between objects.

  Associations are the mechanism that allows objects to communicate to each other. It describes the connection between different classes (the connection between the actual objects is called object connection, or *link* .

  Associations can have a role that specifies the purpose of the association and can be uni- or bidirectional (indicates if the two objects participating in the relationship can send messages to the other, of if only one of them knows about the other). Each end of the association also has a multiplicity value, which dictates how many objects on this side of the association can relate to one object on the other side.

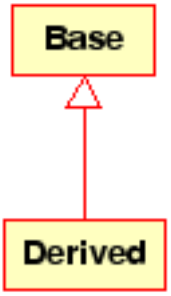


Figure 6.4: Visual representation of a generalization in UML

Figure 6.5: Visual representation of an Association in UML **Aggregation**

In UML, associations are represented as lines connecting the classes participating in the relationship, and can also show the role and the multiplicity of each of the participants. Multiplicity is displayed as a range [min..max] of non-negative values, with a star (*) on the maximum side representing infinite.

- **Aggregations** Aggregations are a special type of associations in which the two participating classes don't have an equal status, but make a "whole-part" relationship. An Aggregation describes how the class that takes the role of the whole, is composed (has) of other classes, which take the role of the parts. For Aggregations, the class acting as the whole always has a multiplicity of one.

  In UML, Aggregations are represented by an association that shows a rhomb on the side of the whole.

- **Composition** Compositions are associations that represent *very strong* aggregations. This means, Compositions form whole-part relationships as well, but the relationship is so strong that the parts cannot exist on its own. They exist only inside the whole, and if the whole is destroyed the parts die too.

  In UML, Compositions are represented by a solid rhomb on the side of the whole.

Other Class Diagram Items Class diagrams can contain several other items besides classes.

- **Interfaces** are abstract classes which means instances can not be directly created of them. They can contain operations but no attributes. Classes can inherit from

Figure 6.6: Visual representation of an Aggregation relationship in UML

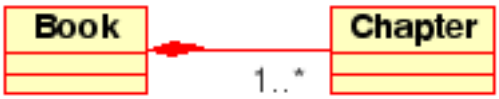


Figure 6.7:

interfaces (through a realisation association) and instances can then be made of these diagrams.

- **Datatypes** are primitives which are typically built into a programming language. Common examples include integers and booleans. They can not have relationships to classes but classes can have relationships to them.

- **Enums** are a simple list of values. A typical example is an enum for days of the week. The options of an enum are called Enum Literals. Like datatypes they can not have relationships to classes but classes can have relationships to them.

- **Packages** represent a namespace in a programming language. In a diagram they are used to represent parts of a system which contain more than one class, maybe hundereds of classes.

Sequence Diagrams

Sequence Diagrams show the message exchange (i.e. method call) between several Objects in a specific time-delimited situation. Objects are instances of classes. Sequence Diagrams put special emphasis in the order and the times in which the messages to the objects are sent.

In Sequence Diagrams objects are represented through vertical dashed lines, with the name of the Object on the top. The time axis is also vertical, increasing downwards, so that messages are sent from one Object to another in the form of arrows with the operation and parameters name.

Messages can be either synchronous, the normal type of message call where control is passed to the called object until that method has finished running, or asynchronous where control is passed back directly to the calling object. Synchronous messages have a vertical box on the side of the called object to show the flow of program control.

Collaboration Diagrams

Collaboration Diagrams show the interactions occurring between the objects participating in a specific situation. This is more or less the same information shown by Sequence Diagrams but there the emphasis is put on how the interactions occur in time while the Collaboration Diagrams put the relationships between the objects and their topology in the foreground.

In Collaboration Diagrams messages sent from one object to another are represented by arrows, showing the message name, parameters, and the sequence of the message. Collaboration Diagrams are specially well suited to showing a specific program flow or situation and are one of the best diagram types to quickly demonstrate or explain one process in the program logic.

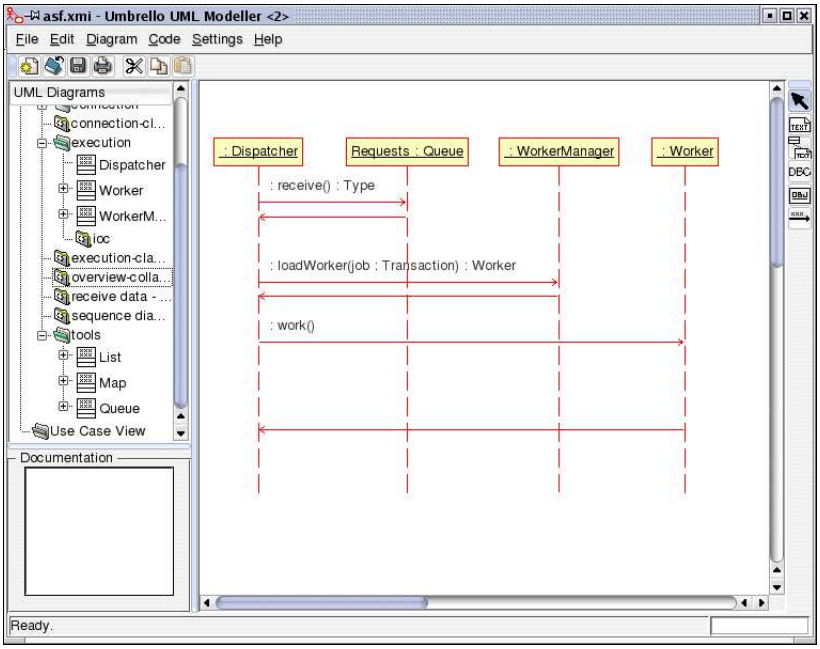


Figure 6.8: Umbrello UML Modeller showing a Sequence Diagram

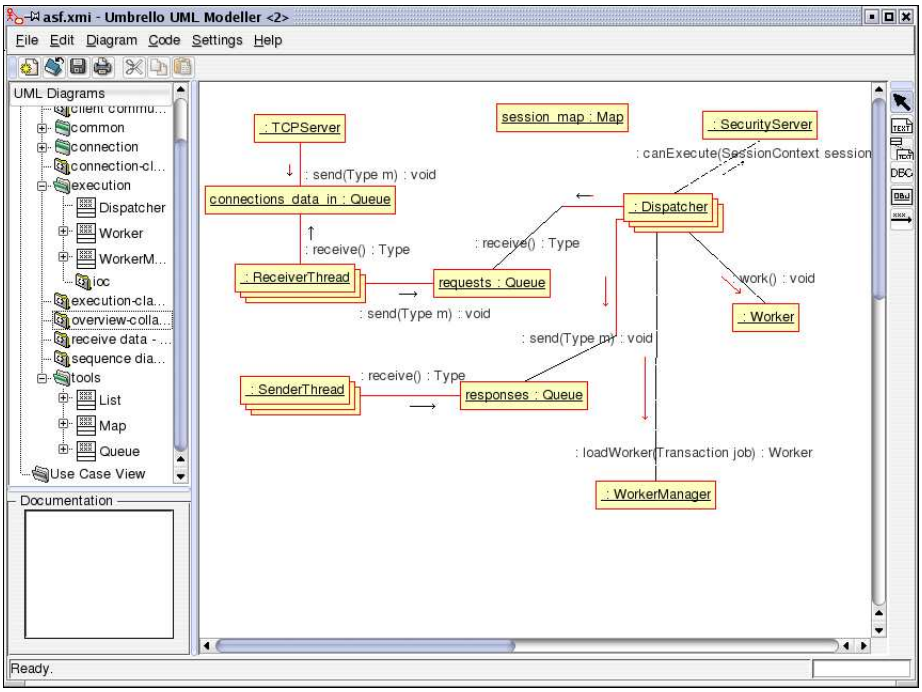


Figure 6.9: Umbrello UML Modeller showing a Collaboration Diagram

State Diagram

State Diagrams show the different states of an Object during its life and the stimuli that cause the Object to change its state.

State Diagrams view Objects as *state machines* or finite automata that can be in one of a set of finite states and that can change its state via one of a finite set of stimuli. For example an Object of type *NetServer* can be in one of following states during its life:

- Ready
- Listening
- Working
- Stopped

and the events that can cause the Object to change states are

- Object is created
- Object receives message listen
- A Client requests a connection over the network
- A Client terminates a request
- The request is executed and terminated
- Object receives message stop ... etc

Activity Diagram

Activity Diagrams describe the sequence of activities in a system with the help of Activities. Activity Diagrams are a special form of State Diagrams, that only (or mostly) contains Activities.

Activity Diagrams are always associated to a *Class* , an *Operation* or a *Use Case* .

Activity Diagrams support sequential as well as parallel Activities. Parallel execution is represented via Fork/Wait icons, and for the Activities running in parallel, it is not important the order in which they are carried out (they can be executed at the same time or one after the other)

Component Diagrams

Component Diagrams show the software components (either component technologies such as KParts, CORBA components or Java Beans or just sections of the system which are clearly distinguishable) and the artifacts they are made out of such as source code files, programming libraries or relational database tables.

Deployment Diagrams

Deployment diagrams show the runtime component instances and their associations. They include Nodes which are physical resources, typically a single computer. They also show interfaces and objects (class instances).

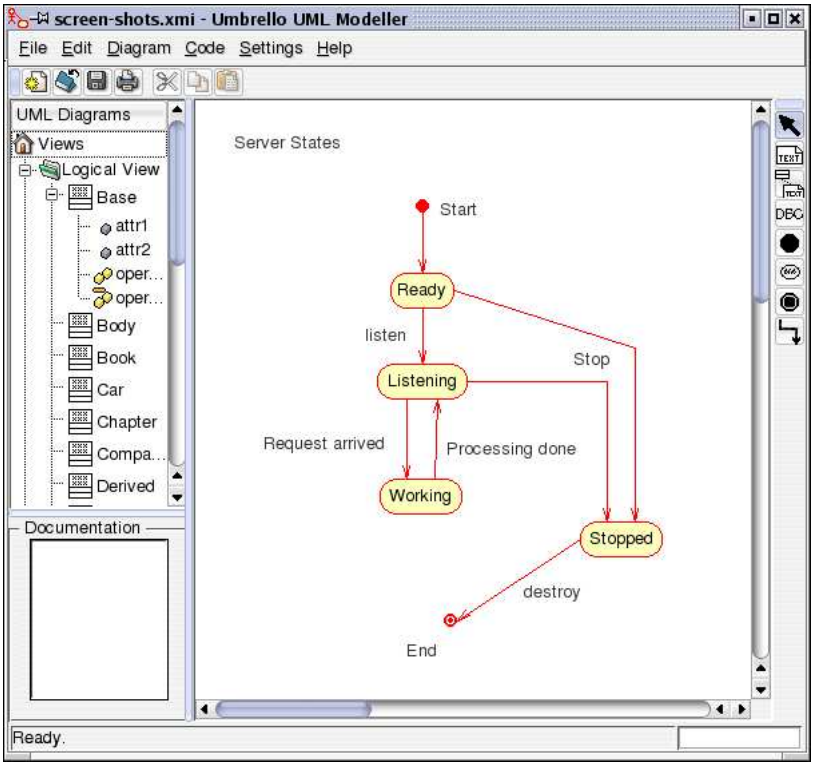


Figure 6.10: Umbrello UML Modeller showing a State Diagram

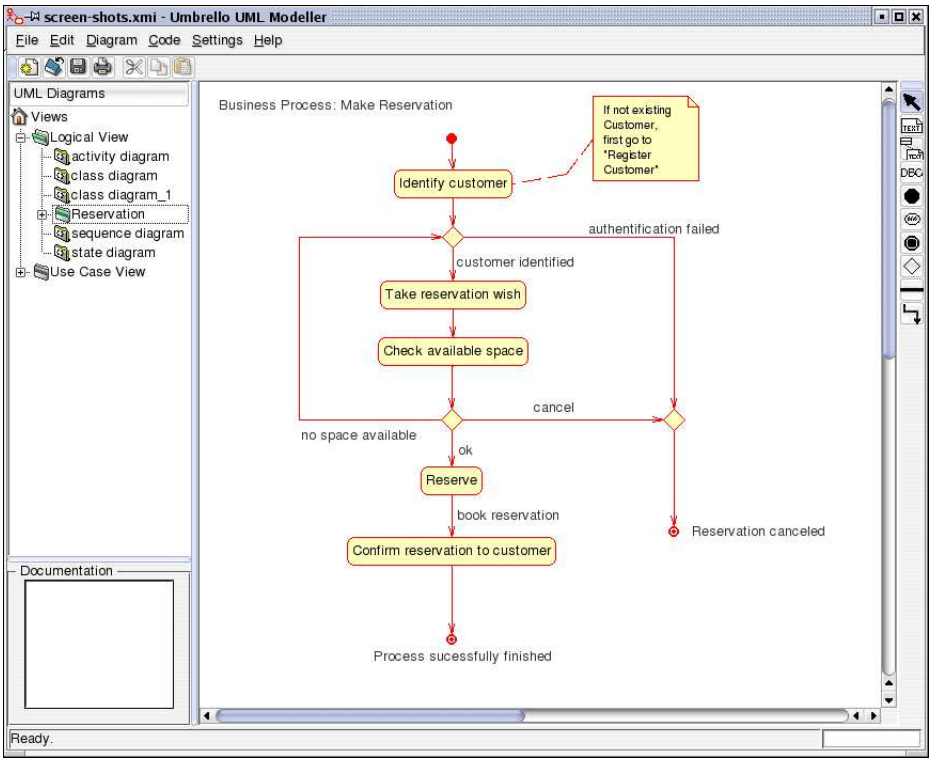


Figure 6.11: Umbrello UML Modeller showing an Activity Diagram

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.

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

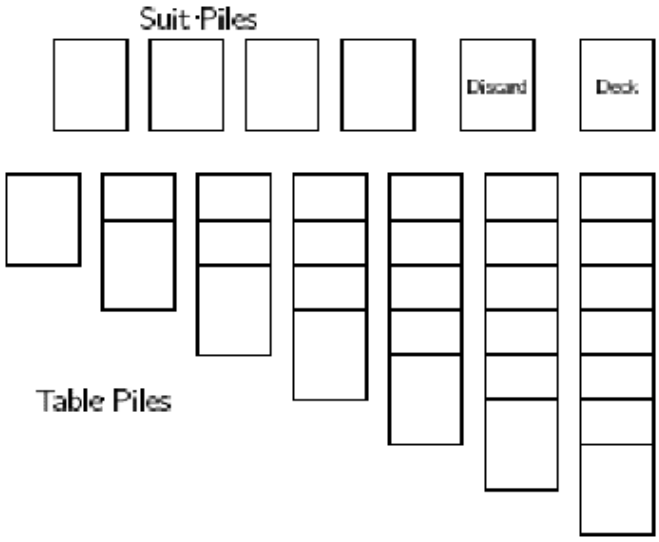


Figure 7.1: Layout of the Solitaire Game

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
```

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`).

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.

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

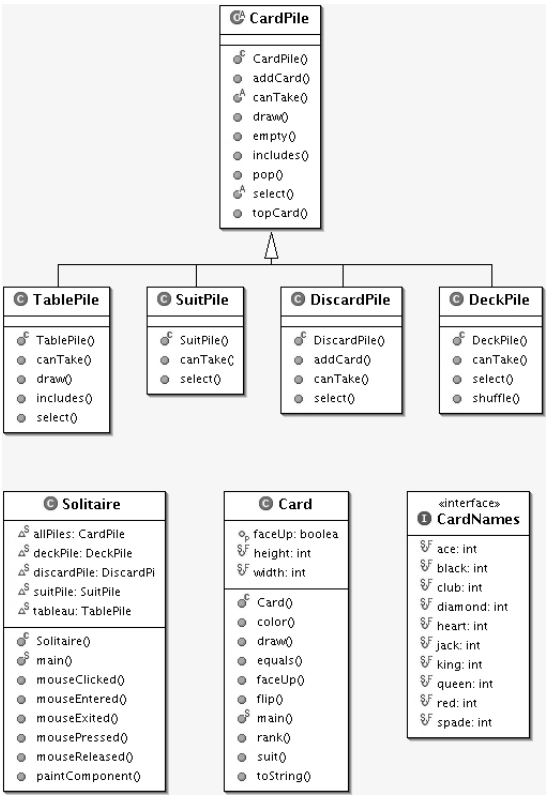


Figure 7.2: Class diagram for the Solitaire app

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:

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.

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) { }
}
```

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

Advanced GUI Programming

IT'S POSSIBLE TO PROGRAM A WIDE VARIETY of GUI applications using only the techniques covered in the previous chapter. In many cases, the basic events, components, layouts, and graphics routines covered in that chapter suffice. But the Swing graphical user interface library is far richer than what we have seen so far, and it can be used to build highly sophisticated applications. This chapter is a further introduction to Swing. Although the title of the chapter is "Advanced GUI Programming," it is still just an introduction. Full coverage of Swing would require at least another complete book.

In this chapter, we'll take a more detailed look at Swing, starting with a few more features of the `Graphics` class. We'll cover a number of new layout managers, component classes, and event types, and we'll see how to open independent windows and dialog boxes on the screen. We'll also look at two other programming techniques, timers and threads.

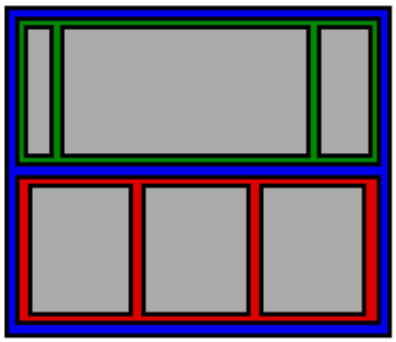
8.1 More about Layouts and Components

SWING INCLUDES A VARIETY of GUI components. We have already encountered a few of these, such as `JApplet`, `JButton`, and `JPanel`. In then next few sections, we will be studying Swing components in more detail.

Most Swing components are defined by subclasses of the `javax.swing.JComponent` class. A `JComponent` cannot stand on its own. It must be contained in some other component. We have seen, for example, that `JPanels` can act as containers for other `JComponents`. At the top level of this containment hierarchy are classes such as `JApplet`. A `JApplet` is not a `JComponent`, but it can serve as a container for `JComponents`. A `JApplet` is a top-level container that is meant to appear on a Web page. We'll see two more top-level container classes, `JFrame` and `JDialog`, which can be used to create independent windows on the computer screen.

The basic properties of components and containers are actually defined by the AWT classes `java.awt.Component` and `java.awt.Container`. Occasionally, you will see these classes used in Swing. For example, the `getContentPane()` method in a `JApplet` has a return type of `Container` rather than `JPanel` or `JComponent` as you might expect.

A `JPanel` is a container that is itself a `JComponent`. A `JPanel` can contain other components, and it can in turn be contained in another component. The fact that panels can contain other panels means that you can have many levels of components containing other components, as shown in the illustration on the right. Several other classes, such as `Box` and `TabbedPane`, also define components that can be used as containers. This leads to two questions: How are components added to a container? How are their sizes and positions controlled?

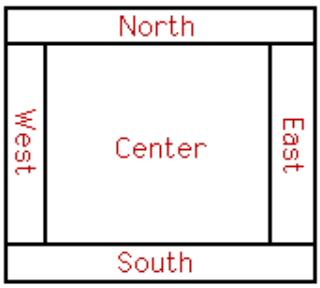


Three Panels, shown in color, containing six additional Components, shown in gray.

The sizes and positions of the components in a container are usually controlled by a layout manager. Different layout managers implement different ways of arranging components. There are several predefined layout manager classes, including `FlowLayout`, `GridLayout`, `BorderLayout`, `BoxLayout`, `CardLayout` and `GridBagLayout`. All these classes are defined in the package `java.awt`. It is also possible to define new layout managers, if none of these suit your purpose. Every container is assigned a default layout manager when it is first created. For `JPanels`, the default layout manager belongs to the class `FlowLayout`. The content pane of a `JApplet` uses a `BorderLayout` by default. You can change the layout manager of a container using its `setLayout()` method.

As for adding components to a container, that's easy. You just use one of the container's `add()` methods. There are several `add()` methods. Which one you should use depends on what type of `LayoutManager` is being used by the container, so I will discuss the appropriate `add()` methods as I go along.

8.1.1 BorderLayout



A `BorderLayout` places one component in the center of a container. This central component is surrounded by up to four other components that border it to the "North", "South", "East", and "West", as shown in the diagram at the right. Each of the four bordering components is optional. The layout manager first allocates space to the bordering components. Any space that is left over goes to the center component.

If a container uses a `BorderLayout`, then components should be added to the container using a version of the `add()` method that has two parameters. The first parameter is the component that is being added to the container. The second parameter specifies where the component is to be placed. It must be one of the constants `BorderLayout.CENTER`,

`BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, or `BorderLayout.WEST`. If the second parameter is omitted, then `BorderLayout.CENTER` is used by default. For example, the following code creates a panel with `drawArea` as its center component and with scroll bars to the right and below:

```
JPanel panel = new JPanel();
  // To use BorderLayout with a JPanel, you have
  // to change the panel's layout manager; otherwise,
  // a FlowLayout is used.  Alternatively, you
  // can provide the layout manager as a
  // parameter to the constructor:
  // panel = new JPanel( new BorderLayout() );
panel.setLayout(new BorderLayout());
  // Assume drawArea already exists.
panel.add(drawArea, BorderLayout.CENTER);
  // Assume hScroll is a horizontal scroll bar
  // component that already exists.
panel.add(hScroll, BorderLayout.SOUTH);
  // Assume vScroll is a vertical scroll bar
  // component that already exists.
panel.add(vScroll, BorderLayout.EAST);
```

Sometimes, you want to leave space between the components in a container. You can specify horizontal and vertical gaps in the constructor of a `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 horizontal gap is inserted between the center and west components and between the center and east components; the vertical gap is inserted between the center and north components and between the center and south components. (The default layout for a JApplet's content pane is a `BorderLayout` with no horizontal or vertical gap.)

8.1.2 GridLayout

A `GridLayout` lays out components in a grid of equal sized rectangles. The illustration shows how the components would be arranged in a grid layout with 3 rows and 2 columns. If a container uses a `GridLayout`, the appropriate `add` method takes a single parameter of type `Component` (for example: `add(myButton)`). 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` with `R` rows and `C` columns takes the form "`new GridLayout(R,C)` ". If you want to leave horizontal gaps of `H` pixels between columns and vertical gaps of `V` pixels between rows, use "`new GridLayout(R,C,H,V)` " instead.

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

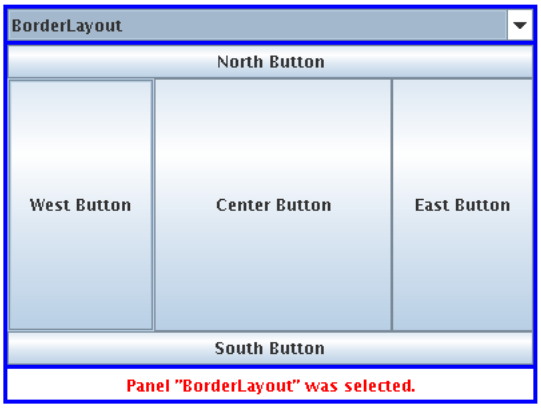
Horizontal grids, with a single row, and vertical grids, with a single column, are very common. For example, suppose that `button1`, `button2`, and `button3` are buttons and that you'd like to display them in a horizontal row in a panel. If you use a horizontal grid for the panel, then the buttons will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

```
JPanel buttonBar = new JPanel();
buttonBar.setLayout(new GridLayout(1,3));
   // (Note: The "3" here is pretty much ignored, and
   // you could also say "new GridLayout(1,0)".
   // To leave gaps between the buttons, you could use
   // "new GridLayout(1,0,5,5)".)
buttonBar.add(button1);
buttonBar.add(button2);
buttonBar.add(button3);
```

8.1.3 An Example

To finish this survey of layout managers, here is an applet that demonstrates layout managers of various types:

Applet Reference: see (Applet "LayoutDemo")



The applet itself uses a `BorderLayout` with vertical gaps of 3 pixels. These gaps show up in blue. The Center component of the applet is a `JPanel`, which uses a `CardLayout` as its layout manager. The layout contains eight cards. Each card is itself another panel

that contains several buttons. Each card uses a different type of layout manager (several of which are extremely stupid choices for laying out buttons).

The North component of the applet is a `JComboBox`, which contains the names of the eight panels in the card layout. The user can switch among the cards by selecting items from this menu. The South component of the applet is a `JLabel` that displays an appropriate message whenever the user clicks on a button or chooses an item from the `JComboBox`.

The source code for this applet is in the file LayoutDemo.java. It consists mainly of a long `init()` method that creates all the buttons, panels, and other components and lays out the applet.

8.2 Basic Components and Their Events

THIS SECTION DISCUSSES some of the GUI interface elements that are represented by subclasses of `JComponent`. The treatment here is brief and covers only the basic uses of each component type. After you become familiar with the basics, you might want to consult a Java reference for more details. I will give some examples of programming with components in the next section.

The exact appearance of a Swing component and the way that the user interacts with the component are not fixed. They depend on the look-and-feel of the user interface. While Swing supports a default look-and-feel, which is probably the one that you will see most often, it is possible to change the look-and-feel. For example, a Windows look-and-feel could be used to make a Java program that is running on a Windows computer look more like a standard Windows program. While this can improve the user's experience, it means that some of the details that I discuss will have to be qualified with the phrase "depending on the look-and-feel."

The `JComponent` class itself defines many useful methods that can be used with components of any type. We've already used some of these in examples. Let `comp` be a variable that refers to any JComponent. Then the following methods are available (among many others):

- `comp.getSize()` is a function that returns a `Dimension` object. This object contains two instance variables, `comp.getSize().width` and `comp.getSize().height`, that give the current size of the component. You can also get the height and width more directly by calling `comp.getHeight()` and `comp.getWidth()`. 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 changes, and the user cannot do anything with it. There is a boolean function, `comp.getEnabled()` that you can call to discover whether the component is enabled.

- `comp.setVisible(true)` and `comp.setVisible(false)` can be called to hide or show the component.

- `comp.setBackground(color)` and `comp.setForeground(color)` set the background and foreground colors for the component. If no colors are set for a component, the colors are determined by the look-and-feel.

- `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. In the default look-and-feel, 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.setFont(font)` sets the font that is used for text displayed on the component. The parameter is an object of type `java.awt.Font`.

- `comp.setToolTipText(string)` sets the specified string as a "tool tip" for the component. The tool tip is displayed if 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.setCursor(cursor)` sets the cursor image that represents the mouse position when the mouse cursor is inside this component. The parameter is an object belonging to the class `java.awt.Cursor` . Generally, this parameter takes the form `Cursor.getPredefinedCursor(id)` where `id` is one of several constants defined in the `Cursor` class. The most useful values are probably `Cursor.HAND_CURSOR`, `Cursor.CROSSHAIR_CURSOR`, `Cursor.WAIT_CURSOR`, and `Cursor.DEFAULT_CURSOR`. For example, if you would like the cursor to appear as a little pointing hand when the mouse is in the component `comp`, you can use:
  `comp.setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));`

- `comp.setPreferredSize(size)` sets the size at which the component should be displayed, if possible. The parameter is of type `Dimension`, and a call to this method usually looks something like "`setPreferredSize(newDimension(100,50))` ". The preferred size is used as a hint by layout managers, but will not be respected in all cases. In a BorderLayout, for example, the preferred size of the Center component is irrelevant, but the preferred sizes of the North, South, East, and West components are used by the layout manager to decide how much space to allow for those components. 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.

- `comp.getParent()` is a function that returns a value of type `java.awt.Container`. The return value is the container that directly contains the component, if any. For a top-level component such as a `JApplet`, the value will be null.

- `comp.getLocation()` is a function that returns the location of the top-left corner of the component. The location is specified in the coordinate system of the component's parent. The returned value is an object of type `Point`. An object of type `Point` contains two instance variables, `x` and `y`.

For the rest of this section, we'll look at subclasses of `JComponent` that represent common GUI components. Remember 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.

8.2.1 The JButton Class

An object of class `JButton` is a push button. You've already seen buttons used in the previous chapter, but we can use a review of `JButtons` as a reminder of what's involved in using components, events, and listeners. (Some of the methods described here are new.)

- **Constructors** : The `JButton` class has a constructor that takes a string as a parameter. This string becomes the text displayed on the button. For example: `stopGoButton = new JButton(''Go'')`

- **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.

- **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. 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** : There are several useful methods 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.

`JButtons` also 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!"

By the way, it's possible for a `JButton` to display an icon instead of or in addition to the text that it displays. An icon is simply a small image. Several other components can also display icons. However, I will not cover this aspect of Swing in this book. Consult a Java reference if you are interested.

8.2.2 The JLabel Class

`JLabels` are certainly the simplest type of component. An object of type `JLabel` is just a single 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 the `JLabel` class is a subclass of `Component`, you can use methods such as `setForeground()` with labels. If you want the background color to have any effect, you should call `setOpaque(true)` on the label, since otherwise the `JLabel` might not fill in its background (depending on the look-and-feel). For example:

```
JLabel message = new JLabel("Hello World!");
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 bold font.
message.setOpaque(true); // Make sure background is filled in.
```

8.2.3 The JCheckBox Class

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. The current state of a checkbox is set using its `setSelected(boolean)` method. You can check or uncheck a checkbox `showTime` programmatically by saying: "`showTime.setSelected(true);`'' or "`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 changes, 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 of a checkbox, you must register an `ActionListener` with the checkbox. (Note that if you change the state by calling the `setSelected()` method, no `ActionEvent` is generated. However, there is another method in the `JCheckBox` class, `doClick()`, which simulates a user click on the checkbox and does generate an `ActionEvent`.)

When handling an `ActionEvent`, you 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).getSelected();
            ... // respond to the change of state }
     else if (source == cb2) {
        boolean newState = ((JCheckBox)cb2).getSelected();
            ... // 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.

8.2.4 The JRadioButton and ButtonGroup Classes

Closely related to checkboxes are radio buttons. Radio buttons occur in groups. At most one radio button in a group can be selected at any given time. In Java, a radio button is represented by an object of type `JRadioButton`. When used in isolation, a `JRadioButton` acts just like a `JCheckBox`, and it has the same methods and events. Ordinarily, however, a `JRadioButton` is used in a group. A group of radio buttons is represented by an object belonging to the class `ButtonGroup`. A `ButtonGroup` is *not* a component and does not itself have a visible representation on the screen. A `ButtonGroup` works behind the scenes to organize a group of radio buttons, so that at most one button in the group can be selected at any given time.

To use a group of radio buttons, you must create a `JRadioButton` object for each button in the group, and you must create one object of type `ButtonGroup` to organize the individual buttons into a group. Each `JRadioButton` must be added individually to some container, so that it will appear on the screen. (A `ButtonGroup` plays no role in the placement of the buttons on the screen.) Each `JRadioButton` must also be added to the `ButtonGroup`, which has an `add()` method for this purpose. If you want one of the buttons to be selected at start-up, you can call `setSelected(true)` for that button. If you don't do this, then none of the buttons will be selected until the user clicks on one of them.

As an example, here is how you could set up a set of radio buttons that can be used to select a color:

```
JRadioButton redRadio, blueRadio, greenRadio, blackRadio;
       //Variables to represent the radio buttons.
       // These should probably be instance variables, so
       // that they can be used throughout the program.

ButtonGroup colorGroup = new ButtonGroup();

redRadio = new JRadioButton("Red"); // Create a button.
colorGroup.add(redRadio); // Add it to the group.

blueRadio = new JRadioButton("Blue");
colorGroup.add(blueRadio);
```

```
greenRadio = new JRadioButton("Green");
colorGroup.add(greenRadio);

blackRadio = new JRadioButton("Black");
colorGroup.add(blackRadio);

redRadio.setSelected(true); // Make an initial selection.
```

The individual buttons must still be added to a container if they are to appear on the screen. If you want to respond immediately when the user clicks on one of the radio buttons, you should register an `ActionListener` for each button. Here is an applet that demonstrates this. When you click one of the radio buttons, the background color of the label is changed:

Applet Reference: see (Applet "RadioButtonDemo")

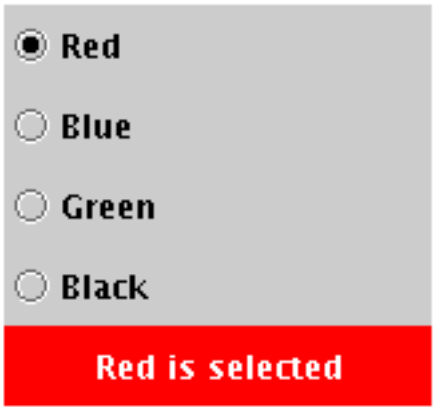


Figure 8.1: RadioButtonDemo

The source code for the applet is in the file RadioButtonDemo.java. Just as for checkboxes, it is not always necessary to register listeners for radio buttons. In many cases, you can simply check the state of each button when you need to know it, using the `isSelected()` method.

8.2.5 The JTextField and JTextArea Classes

`JTextFields` and `JTextAreas` are boxes where the user can type in and edit text. The difference between them is that a `JTextField` contains a single line of editable text, while a `JTextArea` can display 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.

`JTextField` and `JTextArea` are subclasses of `javax.swing.text.JTextComponent`, which defines their common behavior. The `JTextComponent` class supports the idea of a selection. A selection is a subset of the characters in the `JTextComponent`, including all the characters from some starting position to some ending position. The selection is highlighted on the screen. The user selects text by dragging the mouse over it. Some useful methods in class `JTextComponent` include the following. They can, of course, be used for both `JTextFields` and `JTextAreas`.

```
void setText(String newText);  // substitute newText
                               //   for current contents

        String getText();  // return a copy of the current contents

        String getSelectedText();  // return the selected text

        void select(int start, int end);  // change the selection;
           // characters in the range  start <= pos < end  are
           // selected; characters are numbered starting from zero

        void selectAll();  // select the entire text
        int getSelectionStart();  // get starting point of selection
        int getSelectionEnd();  // get end point of selection

        void setEditable(boolean canBeEdited);
          // specify whether or not the text in the component
          // can be edited by the user
```

The `requestFocus()` method, inherited from `JComponent`, is also useful for text components. The constructor for a `JTextField` takes the form

```
JTextField(int columns);
```

where `columns` 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, the number of characters visible in the text field might not be exactly 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 the maximum size 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:

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

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

The constructors for a `JTextArea` are

```
JTextArea();
JTextArea(int lines, int columns);
JTextArea(String contents);
JTextArea(String contents, int lines, int columns);
```

The parameter `lines` specifies how many lines of text should be visible in the text area. This determines the preferred height of the text area. (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 procedures to those inherited from `JTextComponent`:

```
void append(String text);
      // add the specified text at the end of the current
      // contents; line breaks can be inserted by using the
      // special character \n
void insert(String text, int pos);
      // insert the text, starting at specified position
void replaceRange(String text, int start, int end);
      // delete the text from position start to position end
      //    and then insert the specified text in its place
void setLineWrap(boolean wrap);
      // If wrap is true, then a line that is too long to be
      // displayed in the text area will be "wrapped" onto
      // the next line.  The default value is false.
```

A `JTextField` generates an `ActionEvent` when the user presses return while typing in the `JTextField`. The `JTextField` class includes an `addActionListener()` method that can be used to register a listener with a `JTextField`. In the `actionPerformed()` method, the `evt.getActionCommand()` method will return a copy of the text from the `JTextField`. It is also common to use a `JTextField` by checking its contents, when needed, with the `getText()` method. `JTextAreas` do not generate action events.

8.2.6 Other Components

This section has introduced many, but not all, Swing components. Some Swing components will not be covered at all. These include:

- JSlider
- JScrollbar and JScrollPane
- JList – displays a list of items to the user, and allows the user to select one or several items from the list.
- JTable – displays a two-dimensional table of items, and possibly allows the user to edit them.
- JTree – displays hierarchical data in a tree-like structure.
- JToolBar – holds a row of tools, such as icons and buttons. The user can drag the tool bar away from the window that contains it, and it becomes a separate, floating tool window.
- JSplitPane – a container that displays two components. The user can drag the dividing line between the components to adjust their relative sizes.
- JTabbedPane – a container that displays one of a set of panels. The user selects which panel is visible by clicking on a “tab” at the top of the pane.

8.3 Programming with Components

THE TWO PREVIOUS SECTIONS described some raw materials that are available in the form of layout managers and standard GUI components. This section presents some programming examples that make use of those raw materials.

8.3.1 An Example with Text Input Boxes

As a first example, let's look at a simple calculator applet. This example demonstrates typical uses of `JTextFields`, `JButtons`, and `JLabels`, and it uses several layout managers. In the applet, you can enter two real numbers in the text-input boxes and click on one of the buttons labeled "+", "-", "*", and "/". The corresponding operation is performed on the numbers, and the result is displayed in a `JLabel` at the bottom of the applet. If one of the input boxes contains an illegal entry – a word instead of a number, for example – an error message is displayed in the `JLabel`.

Applet Reference: see (Applet "SimpleCalculator")

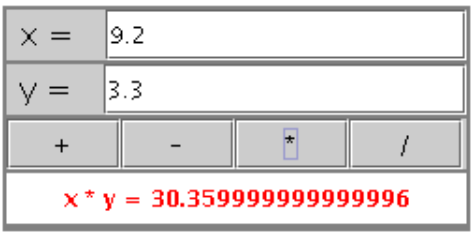


Figure 8.2: SimpleCalculator

When designing an applet such as this one, you should start by asking yourself questions like: How will the user interact with the applet? What components will I need in order to support that interaction? What events can be generated by user actions, and what will the applet do in response? What data will I have to keep in instance variables to keep track of the state of the applet? What information do I want to display to the user? Once you have answered these questions, you can decide how to lay out the components. You might want to draw the layout on paper. At that point, you are ready to begin writing the program.

In the simple calculator applet, the user types in two numbers and clicks a button. The computer responds by doing a computation with the user's numbers and displaying the result. The program uses two `JTextField` components to get the user's input. The `JTextFields` do a lot of work on their own. They respond to mouse, focus, and keyboard events. They show blinking cursors when they are active. They collect and display the characters that the user types. The program only has to do three things with each `JTextField` : Create it, add it to the applet, and get the text that the user has input by calling its `getText()` method. The first two things are done in the applet's `init()` method. The third – getting the user's input from the input boxes – is done in an `actionPerformed()` method, which responds when the user clicks on one of the buttons. When a component is created in one method and used in another, as the input boxes are in this case, we need an instance variable to refer to it. In this case, I use two instance variables, `xInput` and `yInput`, of type `JTextField` to refer to the input boxes. The `JLabel` that is used to display the result is treated similarly: A `JLabel` is created and added to the applet in the `init()` method. When an answer is computed in the `actionPerformed()` method, the `JLabel's setText()` method is used to display the answer in the label. I use an instance variable named `answer`, of type `JLabel`, to refer to the label.

The applet also has four `JButtons` and two more `JLabels` . (The two extra labels display the strings "x=" and "y=".) I use local variables rather than instance variables for these components because I don't need to refer to them outside the `init()` method.

The applet as a whole uses a GridLayout with four rows and one column. The bottom row is occupied by the JLabel, answer . The other three rows each contain several components. Each of the first three rows is filled by a JPanel, which has its own layout manager and contains several components. The row that contains the four buttons is a JPanel which uses a GridLayout with one row and four columns. The JPanels that contain the input boxes use BorderLayouts. The input box occupies the Center position of the BorderLayout, with a JLabel on the West. (This example shows that BorderLayouts are more versatile than it might appear at first.) All the work of setting up the applet is done in its init() method:

```
public void init() {

   /* Since I will be using the content pane several times, declare a
      variable to represent it.  Note that the return type of
      getContentPane() is Container. */

   Container content = getContentPane();

   /* Assign a background color to the applet and its content panel.
      This color will show through between components and around the
      edges of the applet. */

   setBackground(Color.gray); content.setBackground(Color.gray);

   /* Create the input boxes, and make sure that their background color
      is white.  (They are likely to be white by default.) */

   xInput = new JTextField("0"); xInput.setBackground(Color.white);
   yInput = new JTextField("0"); yInput.setBackground(Color.white);

   /* Create panels to hold the input boxes and labels "x =" and "y = ".
      By using a BorderLayout with the JTextField in the Center position,
      the JTextField will take up all the space left after the label is
      given its preferred size. */

   JPanel xPanel = new JPanel(); xPanel.setLayout(new BorderLayout());
   xPanel.add( new Label(" x = "), BorderLayout.WEST );
   xPanel.add(xInput, BorderLayout.CENTER);

   JPanel yPanel = new JPanel(); yPanel.setLayout(new BorderLayout());
   yPanel.add( new Label(" y = "), BorderLayout.WEST );
   yPanel.add(yInput, BorderLayout.CENTER);

   /* Create a panel to hold the four buttons for the four operations.  A
      GridLayout is used so that the buttons will all have the same size
      and will fill the panel.  The applet serves as ActionListener for
      the buttons. */

   JPanel buttonPanel = new JPanel(); buttonPanel.setLayout(new
   GridLayout(1,4));
```

```
        JButton plus = new JButton("+"); plus.addActionListener(this);
        buttonPanel.add(plus);

        JButton minus = new JButton("-"); minus.addActionListener(this);
        buttonPanel.add(minus);

        JButton times = new JButton("*"); times.addActionListener(this);
        buttonPanel.add(times);

        JButton divide = new JButton("/"); divide.addActionListener(this);
        buttonPanel.add(divide);

        /* Create the label for displaying the answer in red on a white
           background.  The label is set to be "opaque" to make sure that the
           white background is painted. */

        answer = new JLabel("x + y = 0", JLabel.CENTER);
        answer.setForeground(Color.red); answer.setBackground(Color.white);
        answer.setOpaque(true);

        /* Set up the layout for the applet, using a GridLayout, and add all
            the components that have been created. */

        content.setLayout(new GridLayout(4,1,2,2)); content.add(xPanel);
        content.add(yPanel); content.add(buttonPanel); content.add(answer);

        /* Try to give the input focus to xInput, which is the natural place
           for the user to start. */

        xInput.requestFocus();

    } // end init()
```

The action of the applet takes place in the `actionPerformed()` method. The algorithm for this method is simple:

```
get the number from the input box xInput
get the number from the input box yInput
get the action command (the name of the button)
if the command is "+"
   add the numbers and display the result in the answer label
else if the command is "-"
   subtract the numbers and display the result in the label
else if the command is "*"
   multiply the numbers and display the result in the label
else if the command is "/"
   divide the numbers and display the result in the label
```

There is one problem with this. When we call `xInput.getText()` and `yInput.getText()` to get the contents of the input boxes, the results are `Strings`, not numbers. We need a

method to convert a string such as "42.17" into the number that it represents. The standard class `Double` contains a static method, `Double.parseDouble(String)` for doing just that. So we can get the first number entered by the user with the commands:

```
String xStr = xInput.getText(); x = Double.parseDouble(xStr);
```

where `x` is a variable of type `double`. Similarly, if we wanted to get an integer value from the string, `xStr`, we could use a static method in the standard `Integer` class: `x=Integer.parseInt(xStr)`. This makes it fairly easy to get numerical values from a `JTextField`, but one problem remains: We can't be sure that the user has entered a string that represents a legal real number. We could ignore this problem and assume that a user who doesn't enter a valid input shouldn't expect to get an answer. However, a more friendly program would notice the error and display an error message to the user. This requires using a "`try...catch` " statement, which is not covered until Chapter9 of this book. My program does in fact use a `try...catch` statement to handle errors, so you can get a preview of how it works. Here is the `actionPerformed()` method that responds when the user clicks on one of the buttons in the applet:

```
public void actionPerformed(ActionEvent evt) {
        // When the user clicks a button, get the numbers
        // from the input boxes and perform the operation
        // indicated by the button.  Put the result in
        // the answer label.  If an error occurs, an
        // error message is put in the label.
   double x, y;  // The numbers from the input boxes.

   /* Get a number from the xInput JTextField.  Use
      xInput.getText() to get its contents as a String.
      Convert this String to a double.  The try...catch
      statement will check for errors in the String.  If
      the string is not a legal number, the error message
      "Illegal data for x." is put into the answer and
      the actionPerformed() method ends. */
   try {
      String xStr = xInput.getText();
      x = Double.parseDouble(xStr);
   }
   catch (NumberFormatException e) { //xStr is not a legal number.
      answer.setText("Illegal data for x.");
      return;
   }

   /* Get a number from yInput in the same way. */
   try {
      String yStr = yInput.getText();
      y = Double.parseDouble(yStr);
   }
   catch (NumberFormatException e) {
      answer.setText("Illegal data for y.");
      return;
   }
```

```
        /* Perform the operation based on the action command
           from 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 the applet can be found in the file SimpleCalculator.java. (It contains very little in addition to the two methods shown above.)

8.4 Menus and Menubars

ANY USER OF A GRAPHICAL USER INTERFACE is accustomed to selecting commands from menus, which can be found in a menu bar at the top of a window (or sometimes at the top of a screen). In Java, menu bars, menus, and the items in menus are `JComponents`, just like all the other Swing components. Java makes it easy to add a menu bar to a `JApplet` or to a `JFrame`. Here is a sample applet that uses menus:

Applet Reference: see (Applet "ShapeDrawWithMenus".)

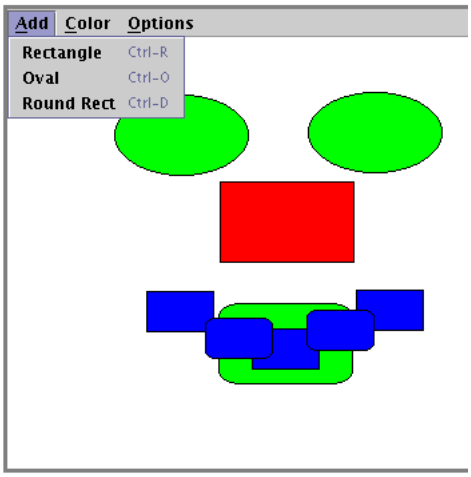


Figure 8.3: ShapeDrawWithMenus

This is a much improved version of the ShapeDraw applet. You can add shapes to the large white drawing area and drag them around. To add a shape, select one of the commands in the "Add" menu. The other menus allow you to control the properties of the shapes and set the background color of the drawing area.

This applet illustrates many ideas related to menus. There is a menu bar. A menu bar serves as a container for menus. In this case, there are three menus in the menu bar. The menus have titles: "Add", "Color", and "Options". When you click on one of these titles, the menu items in the menu appear. Each menu has an associated mnemonic, which is a character that is underlined in the name. Instead of clicking on the menu, you can select it by pressing the mnemonic key while holding down the ALT key. (This assumes that the applet has the keyboard focus.)

Once the menu has appeared, you can select an item in the menu by clicking on it, or by using the arrow keys to select the item and then pressing return. It is possible to assign mnemonics to individual items in a menu, but I haven't done that in this example. The commands in the "Add" menu and the "Clear" command do have accelerators. An accelerator is a key or combination of keys that can be pressed to invoke a menu item without ever opening the menu. The accelerator is shown in the menu, next to the name of the item. For example, the accelerator for the "Rectangle" command in the "Add" menu is "Ctrl-R". This means that you can invoke the command by holding down the Control key and pressing the R key.

The commands in the "Color" menu act like a set of radio buttons. Only one item in the menu can be selected at a given time. The selected item in this menu determines the color of newly added shapes. Similarly, two of the commands in the "Options" menu act just like checkboxes. The first of these items determines whether newly added shapes will be large or small. The second determines whether newly added shapes will have a black border drawn around them.

The last item in the "Options" menu is actually another menu. This is called a sub-menu. When you select this item, the sub-menu will appear. Select an item from the sub-menu to set the background color of the drawing area.

This applet also demonstrates a pop-up menu. The pop-up menu appears when you click on one of the shapes in just the right way. The exact action you have to take depends on the look-and-feel and is called the pop-up trigger. The pop-up trigger is probably either clicking with the right mouse button, clicking with the middle mouse button, or clicking while holding down the Control key. The pop-up menu in this example contains commands for editing the shape on which you clicked.

In the rest of this section, we'll look at how all this can be programmed. If you would like to see the complete source code of the applet, you will find it in the file ShapeDrawWithMenus.java. The source code is just over 600 lines long. The menus are created and configured in a very long `init()` method.

8.4.1 Menu Bars and Menus

A menu bar is just an object that belongs to the class `JMenuBar` . Since `JMenuBar` is a subclass of `JComponent`, a menu bar could actually be used anywhere in any container. In practice though, a `JMenuBar` is generally added to a top-level container such as a `JApplet`. This can be done in the `init()` method of a `JApplet` using commands of the form

```
JMenuBar menubar = new JMenuBar(); setMenuBar(menubar);
```

The applet's `setMenuBar()` method does *not* add the menu bar to the applet's content pane. The menu bar appears in a separate area, above the content pane.

A menu bar is a container for menus. The type of menu that can appear in a menu bar is an object belonging to the class `JMenu`. The constructor for a `JMenu` specifies a title for the menu, which appears in the menu bar. A menu is added to a menu bar using the menu bar's `add()` method. For example, the following commands will create a menu with title "Options" and add it to the `JMenuBar`, `menubar` :

```
JMenu optionsMenu = new JMenu("Options");
menubar.add(optionsMenu);
```

8.5 Timers, Animation, and Threads

JAVA IS A MULTI-THREADED LANGUAGE, which means that several different things can be going on, in parallel. A thread is the basic unit of program execution. A thread executes a sequence of instructions, one after the other. When the system executes a stand-alone program, it creates a thread. (Threads are usually called processes in this context, but the differences are not important here.) The commands of the program are executed sequentially, from beginning to end, by this thread. The thread "dies" when the program ends. In a typical computer system, many threads can exist at the same time. At a given time, only one thread can actually be running, since the computer's Central Processing Unit (CPU) can only do one thing at a time. (An exception to this is a multi-processing computer, which has several CPUs. At a given time, every CPU can be executing a different thread.) However, the computer uses time sharing to give the illusion that several threads are being executed at the same time, "in parallel." Time sharing means that the CPU executes one thread for a while, then switches to another thread, then to another..., and then back to the first thread – typically about 100 times per second. As far as users are concerned, the threads might as well be running at the same time.

To say that Java is a multi-threaded language means that a Java program can create one or more threads which will then run in parallel with the program. This is a fundamental, built-in part of the language, not an option or add-on like it is in some languages. Still, programming with threads can be tricky, and should be avoided unless it is really necessary. Ideally, even then, you can avoid using threads directly by using well-tested classes and libraries that someone has written for you.

8.5.1 Animation In Swing

One of the places where threads are used in GUI programming is to do animation. An animation is just a sequence of still images displayed on the screen one after the other. If the images are displayed quickly enough and if the changes from one image to the next are small enough, then the viewer will perceive continuous motion. To program an animation, you need some way of displaying a sequence of images.

A GUI program already has at least one thread, an event-handling thread, which detects actions taken by the user and calls appropriate subroutines in the program to handle each event. An animation, however, is not driven by user actions. It is something that happens by itself, as an independent process. In Java, the most natural way to accomplish this is to create a separate thread to run the animation. Before the introduction of the Swing GUI, a Java programmer would have to deal with this thread directly. This made

animation much more complicated than it should have been. The good news in Swing is that it is no longer necessary to program directly with threads in order to do simple animations. The neat idea in Swing is to integrate animation with event-handling, so that you can program animations using the same techniques that are used for the rest of the program.

In Swing, an animation can be programmed using an object belonging to the class `javax.swing.Timer`. A `Timer` object can generate a sequence of events on its own, without any action on the part of the user. To program an animation, all your program has to do is create a `Timer` and respond to each event from the timer by displaying another frame in the animation. Behind the scenes, the `Timer` runs a separate thread which is responsible for generating the events, but you never need to deal with the thread directly.

The events generated by a `Timer` are of type `ActionEvent` . The constructor for a `Timer` specifies two things: The amount of time between events and an `ActionListener` that will be notified of each event:

```
Timer(int delayTime, ActionListener listener)
```

The `listener` should be programmed to respond to events from the `Timer` in its `actionPerformed()` method. The delay time between events is specified in milliseconds (where one second equals 1000 milliseconds). The actual delay time between two events can be longer than the requested delay time, depending on how long it takes to process the events and how busy the computer is with other things. In a typical animation, somewhere between ten and thirty frames should be displayed every second. These rates correspond to delay times between 100 and 33.

A `Timer` does not start running automatically when it is created. To make it run, you must call its `start()` method. A timer also has a `stop()` method, which you can call to make it stop generating events. If you have stopped a timer and want to start it up again, you can call its `restart()` method. (The `start()` method should be called only once.) None of these methods have parameters.

Let's look at an example. In the following applet, you can start an animation running by clicking the "Start" button. When you do this, the text on the button changes to "Stop", and you can stop the animation by clicking the button again. This is yet another applet that says "Hello World." The animation simply cycles the color of the message through all possible hues:

Applet Reference: see (Applet "HelloWorldSpectrum".)

Figure 8.4: HelloWorldSpectrum

Here is part of the source code for this applet, omitting the definition of the nested class that defines the drawing surface:

```
public class HelloWorldSpectrum extends JApplet {


   Display display;  // A JPanel belonging to a nested "Display"
                     // class; used for displaying "Hello World."
                     // It defines a method "setColor(Color)" for
                     // setting the color of the displayed message.


   JButton startStopButton; // The button that will be used to
                            // start and stop the animation.


   Timer timer;  // The timer that drives the animation.  A timer
                 // is started when the user starts the animation.
                 // Each time an ActionEvent is received from the
                 // timer, the color of the message will change.
                 // The value of this variable is null when the
                 // animation is not in progress.


   int colorIndex;  // This is be a number between 0 and 100 that
                    // will be used to determine the color.  It will
                    // increase by 1 each time a timer event is
                    // processed.


   public void init() {
          // This is called by the system to initialize the applet.
          // It adds a button to the "south" position in the applet's
          // content pane, and it adds a display panel to the "center"
          // position so that it will fill the rest of the content pane.

       display = new Display();
             // The component that displays "Hello World".

       getContentPane().add(display, BorderLayout.CENTER);
             // Adds the display panel to the CENTER position of the
             // JApplet's content pane.

       JPanel buttonBar = new JPanel();
             // This panel will hold the button and appears
             // at the bottom of the applet.
       buttonBar.setBackground(Color.gray);
       getContentPane().add(buttonBar, BorderLayout.SOUTH);

       startStopButton = new JButton("Start");
       buttonBar.add(startStopButton);
```

```
    startStopButton.addActionListener( new ActionListener() {
              // The action listener that responds to the
              // button starts or stops the animation.  It
              // checks the value of timer to find out which
              // to do.  Timer is non-null when the animation
              // is running, so if timer is null, the
              // animation needs to be started.
          public void actionPerformed(ActionEvent evt) {
             if (timer == null)
                startAnimation();
             else
                stopAnimation();
          }
       });

} // end init()

ActionListener timerListener = new ActionListener() {
         // Define an action listener to respond to events
         // from the timer.  When an event is received, the
         // color of the display is changed.
      public void actionPerformed(ActionEvent evt) {
          colorIndex++;  // A number between 0 and 100.
          if (colorIndex > 100)
             colorIndex = 0;
          float hue = colorIndex / 100.0F;  // Between 0.0F and 1.0F.
          display.setColor( Color.getHSBColor(hue,1,1) );
      }
   };


void startAnimation() {
       // Start the animation, unless it is already running.
       // We can check if it is running since the value of
       // timer is non-null when the animation is running.
       // (It should be impossible for this to be called
       // when an animation is already running... but it
       // doesn't hurt to check!)
    if (timer == null) {
          // Start the animation by creating a Timer that
          // will fire an event every 50 milliseconds, and
          // will send those events to timerListener.
       timer = new Timer(50, timerListener);
       timer.start();  // Make the time start running.
       startStopButton.setText("Stop");
    }
}
```

```
void stopAnimation() {
      // Stop the animation by stopping the timer, unless the
      // animation is not running.
   if (timer != null) {
      timer.stop();   // Stop the timer.
      timer = null;   // Set timer variable to null, so that we
                      //   can tell that the animation isn't running.
      startStopButton.setText("Start");
   }
}


public void stop() {
      // The stop() method of an applet is called by the system
      // when the applet is about to be stopped, either temporarily
      // or permanently.  We don't want a timer running while
      // the applet is stopped, so stop the animation.  (It's
      // harmless to call stopAnimation() if the animation is not
      // running.)
   stopAnimation();
}
```

This applet responds to `ActionEvents` from two sources: the button and the timer that drives the animation. I decided to use a different `ActionListener` object for each source. Each listener object is defined by an anonymous nested class. (It would, of course, be possible to use a single object, such as the applet itself, as a listener, and to determine the source of an event by calling `evt.getSource()` . However, Java programmers tend to be fond of anonymous classes.)

The applet defines methods `startAnimation()` and `stopAnimation()`, which are called when the user clicks the button. Each time the user clicks "Start", a new timer is created and started:

```
timer = new Timer(50,timerListener); timer.start();
```

Remember that without `timer.start()`, the timer won't do anything at all. The constructor specifies a delay time of 50 milliseconds, so there should be about 20 action events from the timer every second. The second parameter is an `ActionListener` object. The timer events will be processed by calling the `actionPerformed()` method of this object. The `stopAnimation()` method calls `timer.stop()`, which ends the flow of events from the timer. The `startAnimation()` and `stopAnimation()` methods also change the text on the button, so that it reads "Stop" when the animation is running and "Start" when it is not running.

The `actionPerformed()` method in `timerListener` responds to a timer event by setting the color of the display. The color is computed from a number, `colorIndex` that is changed for each frame. (The color is specified as an "HSB" color.)

8.5.2 JApplet's start() and stop() Methods

There is one other method in the `HelloWorldSpectrum` class that needs some explanation: the `stop()` method. Every applet has several methods that are meant to be called by the

system at various times in the applet's life cycle. We have already seen that `init()` is called when the applet is first created, before it appears on the screen. Another method, `destroy()` is called just before the applet is destroyed, to give it a chance to clean things up. Two other applet methods, `start()` and `stop()`, are called by the system between `init()` and `destroy()` The `start()` method is always called by the system just after `init()`, and `stop()` is always called just before `destroy()`. However, `start()` and `stop()` can also be called at other times. The reason is that an applet is not necessarily active for the whole time that it exists.

Suppose that you are viewing a Web page that contains an applet, and suppose you follow a link to another page. The applet still exists. It will be there if you hit your browser's Back button to return to the page that contains the applet. However, the page containing the applet is not visible, so the user can't see the applet or interact with it. The applet will not receive any events from the user, and since it is not visible, it will not be asked to paint itself. The system calls the applet's `stop()` method when user leaves the page that contains the applet. The system will call the applet's `start()` method if the user returns to that page. This lets the applet keep track of when it is active and when it is inactive. It might want to do this so that it can avoid using system resources when it is inactive.

In particular, an applet should probably not leave a timer running when it is inactive. The `HelloWorldSpectrum` applet defines the `stop()` method to call `stopAnimation()`, which, in turn, will stop the timer if it is running. When the applet is about to become inactive, the system will call `stop()`, and the animation – if it was running – will be stopped. You can try it, if you are reading this page in a Web browser: Start the animation running in the above applet, go to a different page, and then come back to this page. You should see that the animation has been stopped.

The animation in the `HelloWorldSpectrum` applet is started and stopped under the control of the user. In many cases, we want an animation to run for the entire time that an applet is active. In that case, the animation can be started in the applet's `start()` method and stopped in the applet's `stop()` method. It would also be possible to start the animation in the `init()` method and stop it in the `destroy()` method, but that would leave the animation running, uselessly, while the applet is inactive. In the following example, a message is scrolled across the page. It uses a timer which churns out events for the whole time the applet is active:

Applet Reference: see (Applet "ScrollingHelloWorld".)

Figure 8.5: HelloWorldApplet2

You can find the source code in the file ScrollingHelloWorld.java, but the relevant part here is the `start()` and `stop()` methods:

```
public void start() {
      // Called when the applet is being started or restarted.
      // Create a new timer, or restart the existing timer.
   if (timer == null) {
         // This method is being called for the first time,
         // since the timer does not yet exist.
      timer = new Timer(300, this);  // (Applet listens for events.)
      timer.start();
   }
   else {
      timer.restart();
   }
}


public void stop() {
      // Called when the applet is about to be stopped.
      // Stop the timer.
   timer.stop();
}
```

These methods can be called several times during the lifetime of an applet. The first time `start()` is called, it creates and starts a timer. If `start()` is called again, the timer already exists, and the existing timer is simply restarted.

8.5.3 Other Useful Timer Methods

Although timers are most often used to generate a sequence of events, a timer can also be configured to generate a single event, after a specified amount of time. In this case, the timer is being used as an alarm which will signal the program after a specified amount of time has passed. To use a `Timer` object in this way, call `setRepeats(false)` after constructing it. For example:

```
Timer alarm = new Timer(5000, listener); alarm.setRepeats(false);
alarm.start();
```

The timer will send one action event to the listener five seconds (5000 milliseconds) after it is started. You can cancel the alarm before it goes off by calling its `stop()` method.

Here's one final way to control the timing of events: When you start a repeating timer, the time until the first event is the same as the time between events. Sometimes, it would be convenient to have a longer or shorter delay before the first event. If you start a timer in the `init()` method of an applet, for example, you might want to give the applet some time to appear on the screen before receiving any events from the timer. You can set a separate delay for the first event by calling `timer.setInitialDelay(delay)`, where `timer` is the `Timer` and `delay` is specified in milliseconds as usual.

8.5.4 Using Threads

Although `Timers` can replace threads in some cases, there are times when direct use of threads is necessary. When a `Timer` is used, the processing is done in an event handler. This means that the processing must be something that can be done quickly. An event

handler should always finish its work quickly, so that the system can move on to handle the next event. If an event handler runs for a long time, it will block other events from being processed, and the program will become unresponsive. So, in a GUI application, any computation or process that will take a long time to complete should be run in a separate thread. Then the event-handling thread can continue to run at the same time, responding to user actions as they occur.

As a short and incomplete introduction to threads, we'll look at one example. The example requires some explanation, but the main point for our discussion of threads is that it is a realistic example of a long computation that requires a separate thread. The example is based on the Mandelbrot set, a mathematical curiosity that has become familiar because it can be used to produce a lot of pretty pictures. The Mandelbrot set has to do with the following simple algorithm:

```
Start with a point (x,y) in the plane, where x and y are real numbers.
Let zx = x, and let zy = y.
Repeat the following:
   Replace (zx,zy) with ( zx*zx - zy*zy + x, 2*zx*zy + y )
```

The question is, what will happen to the point `(zx,zy)` as the loop is repeated over and over? The answer depends on the initial point `(x,y)`. For some initial points `(x,y)`, the point `(zx,zy)` will, sooner or later, move arbitrarily far away from the origin, `(0,0)`. For other starting points, the point `(zx,zy)` will stay close to `(0,0)` no matter how many times you repeat the loop. The Mandelbrot set consists of the `(x,y)` points for which `(zx,zy)` stays close to `(0,0)` forever. This would probably not be very interesting, except that the Mandelbrot set turns out to have an incredibly intricate and quite pretty structure.

To get a pretty picture from the Mandelbrot set, we change the question, just a bit. Given a starting point `(x,y)`, we ask, how many steps does it take, up to some specified maximum number, before the point `(zx,zy)` moves some set distance away from `(0,0)` ? We then assign the point a color, depending on the number of steps. If we do this for each `(x,y)` , we get a kind of picture of the set. For a point in the Mandelbrot set, the count always reaches the maximum (since for such points, `(zx,zy)` *never* moves far away from zero). For other points, in general, the closer the point is to the Mandelbrot set, the more steps it will take.

With all that said, here is an applet that computes a picture of the Mandelbrot set. It will begin its computation when you press the "Start" button. (Eventually, the color of every pixel in the applet will be computed, but the applet actually computes the colors progressively, filling the applet with smaller and smaller blocks of color until it gets down to the single pixel level.) The applet represents a region of the plane with `-1.25<=x<=1.0` and `-1.25<=y<=1.25`. The Mandelbrot set is colored purple. Points outside the set have other colors. Try it:

Applet Reference: see (Applet "Mandelbrot")

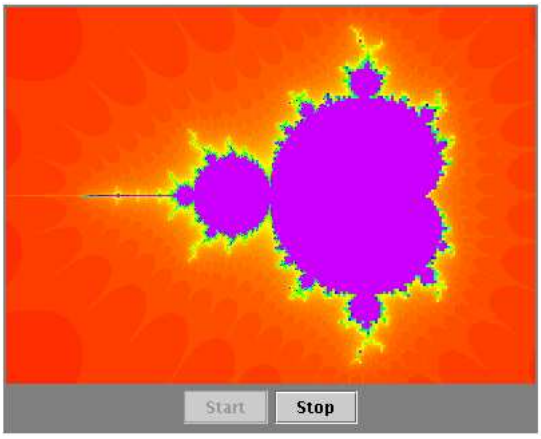


Figure 8.6: Mandelbrot

The algorithm for computing the colors in this applet is:

```
For square sizes 64, 32, 16, 8, 4, 2, and 1:
   For each square in the applet:
      Let (a,b) be the pixel coords of the center of the square.
      Let (x,y) be the real numbers corresponding to (a,b).
      Let (zx,zy) = (x,y).
      Let count = 0.
      Repeat until count is 80 or (zx,zy) is "big":
          Let new_zx = zx*zx - zy*zy + x.
          Let zy = 2*zx*zy + y.
          Let zx = new_zx.
          Let count = count + 1.
      Let color = Color.getHSBColor( count/100.0F, 0.0F, 0.0F )
      Fill the square with that color.
```

The point is that this is a *long* computation. When you click the "Start" button of the applet, the applet creates a separate thread to do this computation.

In Java, a thread is an object belonging to the class `java.lang.Thread`. The purpose of a thread is to execute a single subroutine from beginning to end. In the Mandelbrot applet, that subroutine implements the above algorithm. The subroutine for a thread is usually an instance method

```
public void run()
```

that is defined in an object that implements the interface named `Runnable`. The `Runnable` interface defines `run()` as its only method. The `Runnable` object is provided as a parameter to the constructor of the thread object. (It is also possible to define a thread by declaring a subclass of class `Thread`, and defining a `run()` method in the subclass, but it is more common to use a `Runnable` object to provide the `run()` method for a thread.) If `runnableObject` is an object that implements the `Runnable` interface, then a thread can be constructed with the command:

```
Thread runner = new Thread(runnableObject);
```

The job of this thread is to execute the `run()` method in `runnableObject`. Just as with a `Timer`, it is not enough to construct a thread. You must also start it running by

calling its start method: `runner.start()`. When this method is called, the thread will begin executing the `run()` method in the `runnableObject`, and it will do this in parallel with the rest of the program. When the subroutine ends, the thread will die, and it cannot be restarted or reused. There is no `stop` method for stopping a thread. (Actually, there is one, but it is deprecated, meaning that you are not supposed to call it.) If you want to be able to stop the thread, you need to provide some way of telling the thread to stop itself. I often do this with an instance variable named `running` that is visible both in the `run()` method and elsewhere in the program. When the program wants the thread to stop, it just sets the value of `running` to `false`. In the `run()` method, the thread checks the value of `running` regularly. If it sees that the value of `running` has become `false`, then the `run()` method should end. Here, for example, are the methods that the Mandelbrot applet uses to start and to stop the thread:

```
void startRunning() {
       // A simple method that starts the computational thread,
       // unless it is already running.  (This should be
       // impossible since this method is only called when
       // the user clicks the "Start" button, and that button
       // is disabled when the thread is running.)
     if (running)
        return;
     runner = new Thread(this);
          // Creates a thread that will execute the run()
          // method in this class, which implements Runnable.
     running = true;
     runner.start();
  }

  void stopRunning() {
       // A simple method that is called to stop the computational
       // thread.  This is done by setting the value of the
       // variable, running.  The thread checks this value
       // regularly and will terminate when running becomes false.
     running = false;
  }
```

There are a few more details of threads that you need to understand before looking at the `run()` method from the applet. First, on some platforms, once a thread starts running it grabs control of the CPU, and no other thread can run until it yields control. In Java, a thread can yield control, and allow other threads to run, by calling the static method `Thread.yield()`. I do this regularly in the `run()` method of the applet. If I did not do this, then, on some platforms, the computational thread would block the rest of the program from running. Another way for a thread to yield control is to go to sleep for a specified period of time by calling `Thread.sleep(time)`, where `time` is the number of milliseconds for which the thread will be inactive. The thread in a `Timer`, for example, sleeps between events, since it has nothing else to do.

8.6 Quiz Questions

THIS PAGE CONTAINS A SAMPLE quiz on material from Chapter 7 of this on-line Java textbook. You should be able to answer these questions after studying that chapter.

Question 1:

Question 2:

Question 3: One of the main classes in Swing is the `JComponent` class. What is meant by a component? What are some examples?

Question 4: What is the function of a *LayoutManager* in Java?

Question 5: What does it mean to use a `null` layout manager, and why would you want to do so?

Question 6: What is a `JCheckBox` and how is it used?

Question 7: What is a *thread*

Question 8: Explain how `Timers` are used to do animation.

Question 9: Menus can contain *sub-menus*. What does this mean, and how are sub-menus handled in Java?

Question 10: What is the purpose of the `JFrame` class?

8.7 Programming Exercises

1. StatCalc.java, computes some statistics of a set of numbers. Write an applet that uses the `StatCalc` class to compute and display statistics of numbers entered by the user. The applet will have an instance variable of type `StatCalc` that does the computations. The applet should include a `JTextField` where the user enters a number. It should have four labels that display four statistics for the numbers that have been entered: the number of numbers, the sum, the mean, and the standard deviation. Every time the user enters a new number, the statistics displayed on the labels should change. The user enters a number by typing it into the `JTextField` and pressing return. There should be a "Clear" button that clears out all the data. This means creating a new `StatCalc` object and resetting the displays on the labels. My applet also has an "Enter" button that does the same thing as pressing the return key in the `JTextField`. (Recall that a `JTextField` generates an `ActionEvent` when the user presses return, so your applet should register itself to listen for `ActionEvents` from the `JTextField`.)

2. Write an applet with a `JTextArea` where the user can enter some text. The applet should have a button. When the user clicks on the button, the applet should count the number of lines in the user's input, the number of words in the user's input, and the number of characters in the user's input. This information should be displayed on three labels in the applet. Recall that if `textInput` is a `JTextArea`, then you can get the contents of the `JTextArea` by calling the function `textInput.getText()`. This function returns a `String` containing all the text from the `JTextArea`. The number of characters is just the length of this `String`. Lines in the `String` are separated by the new line character, '\n', so the number of lines is just the number of new line characters in the `String`, plus one. Words are a little harder to count. Exercise 3.4 has some advice about finding the words in a `String`. Essentially, you want to count the number of characters that are first characters in words. Don't forget to put your `JTextArea` in a `JScrollPane` . Scrollbars should appear when the user types more text than will fit in the available area. Here is my applet:

3. The RGBColorChooser applet lets the user set the red, green, and blue levels in a color by manipulating sliders. Something like this could make a useful custom component. Such a component could be included in a program to allow the user to specify a drawing color, for example. Rewrite the `RGBColorChooser` as a component. Make it a subclass of `JPanel` instead of `JApplet`. Instead of doing the initialization in an `init()` method, you'll have to do it in a constructor. The component should have a method, `getColor()`, that returns the color currently displayed on the component. It should also have a method, `setColor(Color c)`, to set the color to a specified value. Both these methods would be useful to a program that uses your component.

   In order to write the `setColor(Color c)` method, you need to know that if `c` is a variable of type `Color`, then `c.getRed()` is a function that returns an integer in the range 0 to 255 that gives the red level of the color. Similarly, the functions `c.getGreen()` and `c.getBlue()` return the blue and green components.

   Test your component by using it in a simple applet that sets the component to a random color when the user clicks on a button, like this one:

4. In the Blackjack game BlackjackGUI.java from Exercise 6.8, the user can click on the "Hit", "Stand", and "NewGame" buttons even when it doesn't make sense to do so. It would be better if the buttons were disabled at the appropriate times. The "New Game" button should be disabled when there is a game in progress. The "Hit" and "Stand" buttons should be disabled when there is not a game in progress. The instance variable `gameInProgress` tells whether or not a game is in progress, so you just have to make sure that the buttons are properly enabled and disabled whenever this variable changes value. Make this change in the Blackjack program. This applet uses a nested class, `BlackjackCanvas` , to represent the board. You'll have to do most of your work in that class. In order to manipulate the buttons, you will have to use instance variables to refer to the buttons.

   I strongly advise writing a subroutine that can be called whenever it is necessary to set the value of the `gameInProgress` variable. Then the subroutine can take responsibility for enabling and disabling the buttons. Recall that if `bttn` is a variable of type `JButton` , then `bttn.setEnabled(false)` disables the button and `bttn.setEnabled(true)` enables the button.

5. Building on your solution to the preceding exercise, make it possible for the user to place bets on the Blackjack game. When the applet starts, give the user $100. Add a `JTextField` to the strip of controls along the bottom of the applet. The user can enter the bet in this `JTextField`. When the game begins, check the amount of the bet. You should do this when the game begins, not when it ends, because several errors can occur: The contents of the `JTextField` might not be a legal number. The bet that the user places might be more money than the user has, or it might be $<=$ 0. You should detect these errors and show an error message instead of starting the game. The user's bet should be an integral number of dollars. You can convert the user's input into an integer, and check for illegal, non-numeric input, with a `try...catch` statement of the form

```
try {
     betAmount = Integer.parseInt( betInput.getText() );
} catch (NumberFormatException e) {
      . . .
        // The input is not a number.
        // Respond by showing an error message and
        // exiting from the doNewGame() method.
}
```

It would be a good idea to make the `JTextField` uneditable while the game is in progress. If `betInput` is the `JTextField`, you can make it editable and uneditable by the user with the commands `betAmount.setEditable(true)` and `betAmount.setEditable(false)`.

In the `paintComponent()` method, you should include commands to display the amount of money that the user has left.

There is one other thing to think about: The applet should not start a new game when it is first created. The user should have a chance to set a bet amount before the game starts. So, in the constructor for the canvas class, you should not call `doNewGame()`. You might want to display a message such as "Welcome to Blackjack" before the first game starts.

6. The `StopWatch` component displays the text "Timing..." when the stop watch is running. It would be nice if it displayed the elapsed time since the stop watch was started. For that, you need to create a `Timer`. Add a `Timer` to the original source code, StopWatch.java, to display the elapsed time in seconds. Create the timer in the `mousePressed()` routine when the stop watch is started. Stop the timer in the `mousePressed()` routine when the stop watch is stopped. The elapsed time won't be very accurate anyway, so just show the integral number of seconds. You only need to set the text a few times per second. For my `Timer` method, I use a delay of 100 milliseconds for the timer. Here is an applet that tests my solution to this exercise:

7. The applet `KaleidaAnimate` shows animations of moving symmetric patterns that look something like the image in a kaleidescope. Symmetric patterns are pretty. Make the `SimplePaint3` applet do symmetric, kaleidoscopic patterns. As the user draws a figure, the applet should be able to draw reflected versions of that figure to make symmetric pictures.

   The applet will have several options for the type of symmetry that is displayed. The user should be able to choose one of four options from a `JComboBox` menu. Using the "No symmetry" option, only the figure that the user draws is shown. Using "2-way symmetry", the user's figure and its horizontal reflection are shown. Using "4-way symmetry", the two vertical reflections are added. Finally, using "8-way symmetry", the four diagonal reflections are also added. Formulas for computing the reflections are given below.

   The source code SimplePaint3.java already has a `drawFigure()` subroutine that draws all the figures. You can add a `putMultiFigure()` routine to draw a figure and some or all of its reflections. `putMultiFigure` should call the existing `drawFigure` to draw the figure and any necessary reflections. It decides which reflections to draw based on the setting of the symmetry menu. Where the `mousePressed`,

mouseDragged, and mouseReleased methods call drawFigure, they should call putMultiFigure instead. The source code also has a repaintRect() method that calls repaint() on a rectangle that contains two given points. You can treat this in the same way as drawFigure(), adding a repaintMultiRect() that calls repaintRect() and replacing each call to repaintRect() with a call to repaintMultiRect(). Alternatively, if you are willing to let your applet be a little less efficient about repainting, you could simply replace each call to repaintRect() with a simple call to repaint(), without parameters. This just means that the applet will redraw a larger area than it really needs to.

If (x,y) is a point in a component that is width pixels wide and height pixels high, then the reflections of this point are obtained as follows:

The horizontal reflection is (width - x, y)

The two vertical reflections are (x, height - y) and (width - x, height - y)

To get the four diagonal reflections, first compute the diagonal reflection of (x,y) as

```
a = (int)( ((double)y / height) * width );
b = (int)( ((double)x / width) * height );
```

Then use the horizontal and vertical reflections of the point (a,b) :

```
(a, b) (width - a, b) (a, height - b) (width - a, height - b)
```

(The diagonal reflections are harder than they would be if the canvas were square. Then the height would equal the width, and the reflection of (x,y) would just be (y,x).)

To reflect a figure determined by two points, (x1,y1) and (x2,y2), compute the reflections of both points to get the reflected figure.

This is really not so hard. The changes you have to make to the source code are not as long as the explanation I have given here.

8. Turn your applet from the previous exercise into a stand-alone application that runs as a JFrame. (If you didn't do the previous exercise, you can do this exercise with the original SimplePaint3.java.) To make the exercise more interesting, remove the JButtons and JComboBoxes and replace them with a menubar at the top of the frame. You can design the menus any way you like, but you should have at least the same functionality as in the original program.

   As an improvement, you might add an "Undo" command. When the user clicks on the "Undo" button, the previous drawing operation will be undone. This just means returning to the image as it was before the drawing operation took place. This is easy to implement, as long as we allow just one operation to be undone. When the off-screen canvas, OSI, is created, make a second off-screen canvas, undoBuffer, of the same size. Before starting any drawing operation, copy the image from OSI to undoBuffer . You can do this with the commands

   ```
   Graphics undoGr = undoBuffer.getGraphics();
   undoGr.drawImage(OSI, 0,0, null);
   ```

When the user clicks "Undo", just swap the values of `OSI` and `undoBuffer` and repaint. The previous image will appear on the screen. Clicking on "Undo" again will "undo the undo."

As another improvement, you could make it possible for the user to select a drawing color using a `JColorChooser` dialog box.

Here is a button that opens my program in its own window. (You don't have to write an applet to launch your frame. Just create the frame in the program's `main()` routine.)

Chapter 9

Generic Programming and Collection Classes

HOW TO AVOID REINVENTING the wheel? Many data structures and algorithms have been studied, programmed, and re-programmed by generations of computer science students. This is a valuable learning experience. Unfortunately, they have also been programmed and re-programmed by generations of working computer professionals, taking up time that could be devoted to new, more creative work. A programmer who needs a list or a binary tree shouldn't have to re-code these data structures from scratch. They are well-understood and have been programmed thousands of times before. The problem is how to make pre-written, robust data structures available to programmers. In this chapter, we'll look at Java's attempt to address this problem.

9.1 Generic Programming

GENERIC PROGRAMMING refers to writing code that will work for many types of data.

An `ArrayList` is essentially a dynamic array of values of type `Object`. Since every class is a sub-class of `Object`, objects belonging to any class can be stored in an `ArrayList`. This is an example of generic programming: The source code for the `ArrayList` class was written once, and it works for objects of any type. (It does, not, however, work for data belonging to the primitive types, such as `int` and `double`.)

The `ArrayList` class is just one of several classes and interfaces that are used for generic programming in Java. We will spend this chapter looking at these classes and how they are used. All the classes discussed in this chapter 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 `import java.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.)

It is no easy task to design a library for generic programming. Java's solution has many nice features but is certainly not the only possible approach. It is almost certainly not the best, but in the context of the overall design of Java, it might be close to optimal. To get some perspective on generic programming in general, it might be useful to look very briefly at generic programming in two other languages.

9.1.1 Generic Programming in Smalltalk

Smalltalk was one of the very first object-oriented programming languages. It is still used today. Although it has not achieved anything like the popularity of Java or C++, it is the source of many ideas used in these languages. In Smalltalk, essentially all programming is generic, because of two basic properties of the language.

First of all, variables in Smalltalk are typeless. A data value has a type, such as integer or string, but variables do not have types. Any variable can hold data of any type. Parameters are also typeless, so a subroutine can be applied to parameter values of any type. Similarly, a data structure can hold data values of any type. For example, once you've defined a binary tree data structure in SmallTalk, you can use it for binary trees of integers or strings or dates or data of any other type. There is simply no need to write new code for each data type.

Secondly, all data values are objects, and all operations on objects are defined by methods in a class. This is true even for types that are "primitive" in Java, such as integers. When the "+" operator is used to add two integers, the operation is performed by calling a method in the integer class. When you define a new class, you can define a "+" operator, and you will then be able to add objects belonging to that class by saying "a+b" just as if you were adding numbers. Now, suppose that you write a subroutine that uses the "+" operator to add up the items in a list. The subroutine can be applied to a list of integers, but it can also be applied, automatically, to any other data type for which "+" is defined. Similarly, a subroutine that uses the "<" operator to sort a list can be applied to lists containing any type of data for which "<" is defined. There is no need to write a different sorting subroutine for each type of data.

Put these two features together and you have a language where data structures and algorithms will work for any type of data for which they make sense, that is, for which the appropriate operations are defined. This is real generic programming. This might sound pretty good, and you might be asking yourself why all programming languages don't work this way. This type of freedom makes it easier to write programs, but unfortunately it makes it harder to write programs that are correct and robust. (See Chapter9.) Once you have a data structure that can contain data of any type, it becomes hard to ensure that it only holds the type of data that you want it to hold. If you have a subroutine that can sort any type of data, it's hard to ensure that it will only be applied to data for which the "<" operator is defined. More particularly, there is no way for a *compiler* to ensure these things. The problem will show up at run time when an attempt is made to apply some operation to a data type for which it is not defined, and the program will crash.

9.1.2 Generic Programming in C++

Unlike Smalltalk, C++ is a very strongly typed language, even more so than Java. Every variable has a type, and can only hold data values of that type. This means that the type of generic programming used in Smalltalk is impossible. Furthermore, C++ does not have anything corresponding to Java's `Object` class. That is, there is no class that is a superclass of all other classes. This means that C++ can't use Java's style of generic programming either. Nevertheless, C++ has a powerful and flexible system of generic programming. It is made possible by a language feature known as templates. In C++, instead of writing a different sorting subroutine for each type of data, you can write a single subroutine template. The template is not a subroutine; it's more like a factory for making subroutines. We can look at an example, since the syntax of C++ is very similar to Java's:

```
template<class ItemType>
      void sort( ItemType A[], int count ) {
            // Sort count items in the array, A, into increasing order.
            // The algorithm that is used here is selection sort.
         for (int i = count-1; i > 0; i--) {
            int position_of_max = 0;
            for (int j = 1; j <= count ; j++)
               if ( A[j] > A[position_of_max] )
                  position_of_max = j;
            ItemType temp = A[count];
            A[count] = A[position_of_max];
            A[position_of_max] = temp;
         }
      }
```

This piece of code defines a subroutine template. If you were to remove the first line, "template<class ItemType>", and substitute the word "int" for the word "ItemType" in the rest of the template, you get a subroutine for sorting arrays of ints. (Even though it says "class ItemType", you can actually substitute any type for ItemType, including the primitive types.) If you substitute "string" for "ItemType", you get a subroutine for sorting arrays of strings. This is pretty much what the compiler does with the template. If your program says "sort(list,10)" where list is an array of ints, the compiler uses the template to generate a subroutine for sorting arrays of ints. If you say "sort(cards,10)" where cards is an array of objects of type Card, then the compiler generates a subroutine for sorting arrays of Cards. At least, it tries to. The template uses the ">" operator to compare values. If this operator is defined for values of type Card, then the compiler will successfully use the template to generate a subroutine for sorting Cards. If ">" is not defined for Cards, then the compiler will fail – but this will happen at compile time, not, as in Smalltalk, at run time where it would make the program crash.

C++ also has templates for making classes. If you write a template for binary trees, you can use it to generate classes for binary trees of ints, binary trees of strings, binary trees of dates, and so on – all from one template. The most recent version of C++ comes with a large number of pre-written templates called the Standard Template Library or STL. The STL is quite complex. Many people would say that its much too complex. But it is also one of the most interesting features of C++.

9.1.3 Generic Programming in Java

Like C++, Java is a strongly typed language. However, generic programming in Java is closer in spirit to Smalltalk than it is to C++. As I've already noted, generic programming in Java is based on the fact that class `Object` is a superclass of every other class. To some extent, this makes Java similar to Smalltalk: A data structure designed to hold `Objects` can hold values belonging to any class. There is no need for templates or any other new language feature to support generic programming.

Of course, primitive type values, such as integers, are not objects in Java and therefor cannot be stored in generic data structures. In fact, there is no way to do generic programming with the primitive data types in Java. The Smalltalk approach doesn't work except for objects, and the C++ approach is not available. Furthermore, generic subroutines are more problematic in Java than they are in either Smalltalk or C++. In Smalltalk, a subroutine can be called with parameter values of any type, and it will work fine as long as

all the operations used by the subroutine are supported by the actual parameters. In Java, parameters to a subroutine must be of a specified type, and the subroutine can only use operations that are defined for that type. A subroutine with a parameter of type `Object` can be applied to objects of any type, but the subroutine can only use operations that are defined in class `Object`, and there aren't many of those! For example, there is no comparison operation defined in the `Object` class, so it is not possible to write a completely generic sorting algorithm. We'll see below how Java addresses this problem.

Because of problems like these, some people (including myself) claim that Java does not really support true generic programming. Other people disagree. But whether it's true generic programming or not, that doesn't prevent it from being very useful.

Java 1.5 Note: Java 1.5 introduces templates that are similar to C++ class templates. As in C++, this makes it possible to do generic programming in a more type-safe way. For example, if *Shape* is a class, then the type `ArrayList<Shape>` represents a list that can only hold values of type *Shape*. The type name `ArrayList<Shape>` is used like any other type. For example, to declare a variable and create an object of this type, you could say "`ArrayList<Shape> shapeList = new ArrayList<Shape>()` ". The variable *shapeList* can then be used like any other ArrayList, except that it can only hold values of type *Shape*. Since the compiler knows this, it can enforce this condition at compile time. Java 1.5 templates still only work with object types, and not with primitive types (but see the note later on this page about automatic conversion between primitive types and "wrapper types").

9.2 An Example: ArrayLists

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[]`. If a subroutine has a formal parameter of type `Object`, than any object can be passed to the subroutine as an actual parameter. The `ArrayList` class is in the package `java.util`, so if you want to use the `ArrayList` class in a program, you should put the directive "`import java.util.ArrayList;` " or "`import java.util.*;` " 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`.

- `list.size()` – This function 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 `newArrayList()` creates an `ArrayList` of size zero.

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

- `list.get(N)` – This function 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 occurs. Calling this function is similar to referring to `A[N]` for an array, `A`, except that 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 function 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 function 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 again 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 function `players.get(i)` to get the value stored at position `i` in the `ArrayList`. The return type of this function is `Object`. In this case the object that is returned by the function 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`:

```
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 all the players 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.

9.3 Collections

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. 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. A map is similar to what I called an "association list" in Section8.4.

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. A set is a collection in which no object can appear more than once.

Note that the terms "collection," "list," "set," and "map" tell you nothing about how the data is stored. A list could be represented as an array, as a linked list, or, for that matter, as a map that associates the elements of the list to the numbers 0, 1, 2,. . . . In fact, these terms are represented in Java not by classes but by interfaces. The interfaces `Collection`, `List`, `Set`, and `Map` specify the basic operations on data structures of these types, but do not specify how the data structures are to be represented or how the operations are to be implemented. That will be specified in the classes that implement the interfaces. Even when you use these classes, you might not know what the implementation is unless you go look at the source code. Java's generic data structures are abstract data types. They are defined by the operations that can be performed on them, not by the physical layout of the data in the computer.

9.3.1 Generic Algorithms and Iterators

The `Collection` interface includes methods for performing some basic operations on collections 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 any object that implements the `Collection` interface. Here are some of the operations that are defined:

- `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 is0.
- `coll.clear()` – removes all objects from the collection.

- `coll.contains(object)` – returns a boolean value that is true if `object` is in the collection.
- `coll.add(object)` – adds `object` to the collection. The parameter can be any `Object`. Some collections can contain the value `null`, while others cannot. 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.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.
- `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 the collection `coll2` to `coll`.
- `coll.removeAll(coll2)` – removes every `object` from `coll` that also occurs in the collection `coll2`.
- `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`.
- `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. 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` 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 exception is `UnsupportedOperationException`.

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 you need can be implemented most efficiently. We'll see specific examples of this in the next two sections.

The `Collection` interface 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. For a binary tree, you can use a recursive

subroutine to do an infix traversal. Collections can be represented in any of these forms and many others besides. With such a variety of traversal mechanisms, how can we 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 different types of iterators, 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 `Collection` interface 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 an interface named `Iterator`. This interface defines just three methods. If `iter` refers to an `Iterator`, then:

- `iter.next()` – returns the next item, and advances the iterator. The return value is of type `Object`. 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. 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. 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 that `coll` is of type `Collection`. Then we can say:

```
Iterator iter = coll.iterator();
while ( iter.hasNext() ) {
   Object item = iter.next();
   System.out.println(item);
}
```

The same general form will work for other types of processing. For example, here is a subroutine that will remove all `null` values from any collection (as long as that collection supports removal of values):

```
void removeNullValues( Collection coll ) {
     Iterator iter = coll.iterator();
     while ( iter.hasNext() ) {
        Object item = iter.next();
        if (item == null)
           iter.remove();
     }
}
```

Collections can hold objects of any type, so the return value of `iter.next()` is `Object`. Now, there's not very much you can do with a general `Object`. In practical situations, a

collection is used to hold objects belonging to some more specific class, and objects from the collection are type-cast to that class before they are used. Suppose, for example, that we are working with `Shapes`, where `Shape` is a class that represents geometric figures. Suppose that the `Shape` class includes a `draw()` method for drawing the figure. Then we can write a generic method for drawing all the figures in a collection of `Shapes` :

```
void drawAllShapes( Collection shapeCollection ) {
      // Precondition: Every item in shapeCollection is non-null
      // and belongs to the class Shape.
     Iterator iter = shapeCollection.iterator();
     while ( iter.hasNext() ) {
           Shape figure = (Shape)iter.next();
           figure.draw();
     }
 }
```

The precondition of this method points out that the method will fail if the method contains an item that does not belong to class `Shape`. When that item is encountered, the type-cast "`(Shape)iter.next()` " will cause an exception of type `ClassCastException`. Although it's unfortunate that we can't have a "Collection of Shapes" in Java, rather than a "Collection of Objects", it's not a big problem in practice. You just have to be aware of what type of objects you are storing in your collections.

Java 1.5 Note: Java 1.5 introduces a new variation on the for loop that makes Iterators unnecessary in many cases. An iterator is often used to apply the same operation to all the elements in a Collection. In Java 1.5, this can be done with a for loop something like this: "for (Object x : c) applyOperation(x)", where *c* is the Collection and *x* is the for-loop variable. The notation "for (Object x : c)" has the meaning "for every Object *x* in the Collection c do the following." Using this notation, the *drawAllShapes* example above could be written simply as:

```
void drawAllShapes( Collection shapeCollection ) {
     // Precondition: Every item in shapeCollection is non-null
     // and belongs to the class Shape.
    for (Object obj : shapeCollection) {
        Shape figure = (Shape)obj;
        figure.draw();
    }
}
```

Using the template notation discussed in the previous Java 1.5 Note, this becomes even nicer:

```
void drawAllShapes( Collection<Shape> shapeCollection ) {
     //Precondition: Every item in shapeCollection is non-null.
    for (Shape figure : shapeCollection)
        figure.draw();
}
```

Note that this works not just for Collection but for other container classes such as ArrayList and Vector. In fact, it even works for applying the same operation to all the elements in an array.

9.3.2 Equality and Comparison

The discussion of methods in the `Collection` interface had an unspoken assumption: It was assumed that it's known what it means for two objects to be "equal." 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 a boolean-valued method `equals(Object)` for testing whether one object is equal to another. For the purposes of collections, `obj1` and `obj2` are considered to be equal if they are both `null`, or if they are both non-`null` and `obj1.equals(obj2)` is `true`. 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:

```
public class Card {  // Class to represent playing cards.
     int suit;  // Number from 0 to 3 :spades, diamonds, clubs or hearts.
     int value; // Number from 1 to 13 that represents the value.
     public boolean equals(Object obj) {
         if (obj == null || ! (obj instanceof Card) ) {
               // obj can't be equal to this Card if obj
               // is not a Card, or if it is null.
            return false;
         }
         else {
            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;
         }
     }
      ... // other methods and constructors
  }
```

Without the `equals()` method in this class, methods such as `contains()` and `remove()` from the `Collection` interface will not work as expected for values of type `Card`.

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`. This interface defines one method: `public int compareTo(Object obj)`

The value returned by `obj1.compareTo(obj2)` should be zero if and only if the objects are equal (that is, if `obj1.equals(obj2)` is true). It should be negative if and only if `obj1` comes before `obj2`, when the objects are arranged in ascending order. And it should be positive if and only if `obj1` comes after `obj2` . In general, it should be considered an error to call `obj1.compareTo(obj2)` if `obj2` is not of the same type as `obj1`. The `String` class implements the `Comparable` interface and defines `compareTo` in a reasonable way. 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:

```
class FullName implements Comparable {
 //Represents a full name: a first name and a last name.
   String firstName, lastName;
   public boolean equals(Object obj) {
      if (obj == null || ! (obj instanceof FullName)) {
         return false;
      }
      else {
         FullName other = (FullName)obj;
         return firstName.equals(other.firstName)
                     && lastName.equals(other.lastName);
      }
   }
   public void compareTo(Object obj) {
      Fullname other = (FullName)obj;
      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.
         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.
         return 1;
      }
      else {
             // Last names are the same, so base the comparison
             // on the first names.
         return firstName.compareTo(other.firstName);
      }
   }
    ... // other methods and constructors
}
```

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 `java.util.Comparator`, which defines the method:

```
public int compare(Object obj1, Object obj2)
```

This method compares two objects and returns a value that is negative, or zero, or positive depending on whether `obj1` comes before `obj2`, or is the same as `obj2`, or comes after `obj2`. 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.

9.3.3 Wrapper Classes

As noted above, Java's generic programming does not apply to the primitive types. Before leaving this section, we should try to address this problem.

You can't store an integer in a generic data structure designed to hold `Objects`. On the other hand, there is nothing to stop you from making an object that *contains* an integer and putting that object into the data structure. In the simplest case, you could define a class that does *nothing but* contain an integer:

```
public class IntContainer { public int value; }
```

In fact, Java already has a class similar to this one. An object belonging to the class `java.lang.Integer` contains a single `int`. It is called a wrapper for that `int`. The `int` value is provided as a parameter to the constructor. For example,

```
Integer val = new Integer(17);
```

creates an `Integer` object that "wraps" the number 17. The `Integer` object can be used in generic data structures and in other situations where an object is required. The `int` value is stored in a `private final` instance variable of the `Integer` object. If `val` refers to an object of type `Integer`, you can find out what `int` it contains by calling the instance method `val.intValue()`. There is no way to change that value. We say that an `Integer` is an immutable object. That is, after it has been constructed, there is no way to change it. (Similarly, an object of type `String` is immutable.)

There are wrapper classes for all the primitive types. All objects belonging to these classes are immutable. The wrapper class for values of type `double` is `java.lang.Double`. The value stored in an object of type `Double` can be retrieved by calling the instance method `doubleValue()`.

The wrapper classes define a number of useful methods. Some of them exist to support generic programming. For example, the wrapper classes all define instance methods `equals(Object)` and `compareTo(Object)` in a reasonable way. Other methods in the wrapper classes are utility functions for working with the primitive types. For example, we encountered the static methods `Integer.parseInt(String)` and `Double.parseDouble(String)` in Section7.4. These functions are used to convert strings such as "42" or "2.71828" into the numbers they represent.

Java 1.5 Note: Java 1.5 makes it easier to use the wrapper classes by introducing automatic conversions between the primitive types and the wrapper types. For example, it is possible to assign a value of type *int* to a variable of type *Integer*, and *vice versa*. In Java

1.5, the statement "`Integer val = new Integer(17)` " could be replaced by "`Integer val = 17` ". Similarly, it is possible to pass a value of type *double* to a function that has a parameter of type *Double*. All this is especially convenient when working with templates, which were mentioned in the first Java 1.5 Note on this page. For example, if *integerList* is a variable of type *ArrayList<Integer>*, you can say "`integerList.add(42)` " and it will be automatically interpreted by the compiler as "`integerList.add(newInteger(42))` ".

9.4 List Classes

IN THE PREVIOUS SECTION, we looked at the general properties of collection classes in Java. In this section, we look at some specific collection classes and how to use them. These classes can be divided into two categories: lists and sets. 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. A set is a collection that has no duplicate entries. The elements of a set might or might not be arranged into some definite order. As with all of Java's collection classes, the items in a list or set are of type `Object`.

9.4.1 The ArrayList and LinkedList Classes

There are two obvious ways to represent a list: as a dynamic array and as a linked list. We've encountered these already in Sections 8.3 and 11.2. Both of these options are available in generic form as the collection's `java.util.ArrayList` and `java.util.LinkedList`. That is, a `ArrayList` represents an ordered sequence of objects stored in an array that will grow in size as new items are added, and a `LinkedList` represents an ordered sequence of objects stored in nodes that are linked together with pointers. Both of these classes implement an interface `java.util.List`, which specifies operations that are available for all lists.

Both list classes support the basic list operations, 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. In a linked list, nodes can be added or removed at any position by changing a few pointer values. The `ArrayList` class is more efficient when random access to items is required. Random access means accessing the n-th item in the list, for any integern. 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 n 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 `Collection` methods discussed in the previous section, including `size()`, `isEmpty()`, `add(Object)`, `remove(Object)`, and `clear()`. The `add(Object)` 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 `List` interface

adds some methods for accessing list items according to their numerical positions in the list. For an object, `list`, of type `List`, these methods include:

- `list.get(index)` – returns the `Object` 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 an object `obj` at position number `index` in the list, replacing the object that was there previously. 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`. 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` can be in the range 0 to `list.size()`, inclusive.
- `list.remove(index)` – removes the object at position number `index`. Items after this position move up one space in the list to fill the hole.
- `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`. If `obj` occurs more than once in the list, the index of the first occurrence is returned.

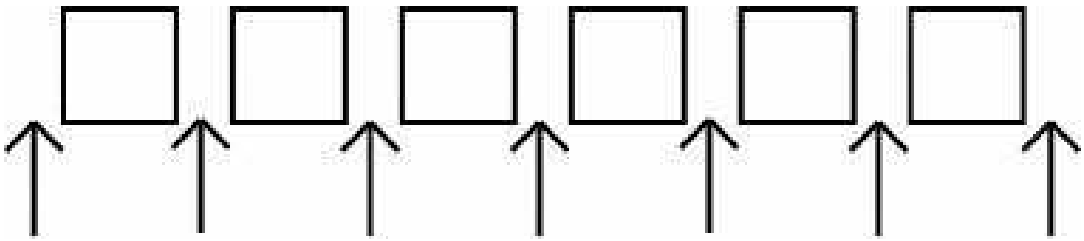
These methods are defined in both the `ArrayList` class and in the `LinkedList` class, although they are only efficient for `ArrayLists`. The `LinkedList` class adds a few additional methods, which are not defined for an `ArrayList`. If `linkedlist` is an object of type `LinkedList`, then

- `linkedlist.getFirst()` – returns the `Object` that is the first item in the list. The list is not modified.
- `linkedlist.getLast()` – returns the `Object` 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` as its return value.
- `linkedlist.removeLast()` – removes the last item from the list, and returns that `Object` as its return value.
- `linkedlist.addFirst(obj)` – adds the `Object`, `obj`, to the beginning of the list.
- `linkedlist.addLast(obj)` – adds the `Object`, `obj`, 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.)

These methods are apparently defined to make it easy to use a `LinkedList` as if it were a stack or a queue. (See Section11.3.) For example, we can use a `LinkedList` as a queue by adding items onto one end of the list (using the `addLast()` method) and removing them from the other end (using the `removeFirst()` method).

If `list` is an object of type `List`, then the method `list.iterator()`, defined in the `Collection` interface, 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. The method `list.listIterator()` returns a `ListIterator` for `list`.

A `ListIterator` has the usual `Iterator` methods `hasNext()` and `next()`, but it also has methods `hasPrevious()` and `previous()` that make it possible to move backwards in the list. 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 a `ListIterator()`, `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(Object)` that adds the specified object to the list at the current position of the iterator. This can be between two existing items or at the beginning of the list or at the end of the list.

(By the way, the lists that are used in the `LinkedList` class are doubly linked lists. That is, each node in the list contains two pointers – one to the next node in the list and one to the previous node. This makes it possible to implement effeciently both the `next()` and `previous()` methods of a `ListIterator`. Also, to make the `addLast()` and `getLast()` methods of a `LinkedList` efficient, the `LinkedList` class includes an instance variable that points to the last node in 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. The idea is to start at the beginning of the list and to move the iterator forward past all the items that are bigger than the item that is being inserted. At that point, the iterator's `add()` method can be used to insert the item at its correct position in the list. In order to say what it means for one item to be "bigger" than another, we assume that the items in the list implement the `Comparable` interface and define the `compareTo()` method. (This interface was discussed in the previous section.) Here is a method that will do this:

```
static void orderedInsert(List list, Comparable newItem) {

      // Precondition:  The items in list are sorted into ascending
      //                order, according to the compareTo method.
      //                newitem.compareTo(item) must be defined for
      //                each item in the list.
      //
      // Postcondition: newItem has been added to the list in its
      //                correct position, so that the list is still
      //                sorted into ascending order.

   ListIterator iter = list.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()) {
           Object 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);

     } // end orderedInsert()
```

Since the parameter in this method is of type `List`, it can be applied to both `ArrayLists` and `LinkedLists`, and it will be about equally efficient for both types of lists. You would probably find it easier to write an `orderedInsert` method using array-like indexing with the methods `get(index)` and `add(index,obj)`. However, that method would be horribly inefficient for `LinkedLists` because `get(index)` is so inefficient for such lists. You can find a program that tests this method in the file ListInsert.java.

9.4.2 Sorting

Sorting a list is a fairly common operation, and there should really be a sorting method in the `List` interface. For some reason, there is not, 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 command

```
Collections.sort(list);
```

can be used to sort a list into ascending order. The items in the list must implement the `Comparable` interface. This method will work, for example, for lists of `Strings`. If a `Comparator` is provided as a second argument:

```
Collections.sort(list,comparator);
```

then 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. We'll see an example of using a `Comparator` in Section4.

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 command:

```
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)` 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`.

9.4.3 The TreeSet and HashSet Classes

A set is a collection of `Objects` in which no object occurs more than once. Objects `obj1` and `obj2` are considered to be the same if `obj1.equals(obj2)` is true, as discussed in the previous section. Sets implement all the general `Collection` methods, but do so in a way that ensures that no element occurs twice in the set. For example, if `set` is an object of type `Set`, then `set.add(obj)` will have no effect on the set if `obj` is already an element of the set. Java has two classes that implement the `Set` interface: `java.util.TreeSet` and `java.util.HashSet`.

In addition to being a `Set`, a `TreeSet` has the property that the elements of the set are arranged into ascending sorted order. An `Iterator` for a `TreeSet` will always visit the elements of the set in ascending order.

A `TreeSet` cannot hold arbitrary objects, since there must be a way to determine the sorted order of the objects it contains. Ordinarily, this means that the objects in a `TreeSet` should implement the `Comparable` interface and that `obj1.compareTo(obj2)` should be defined in a reasonable way for any two objects `obj1` and `obj2` in the set. Alternatively, a `Comparator` can be provided as a parameter to the constructor when the `TreeSet` is created. In that case, the `Comparator` will be used to compare objects that are added to the set.

In the implementation of a `TreeSet`, the elements are stored in something like a binary sort tree. (See Section11.4.) The actually type of tree that is used is balanced in the sense that all the leaves of the tree are at about the same distance from the root of the tree. The number of operations required to find an item in a sorted tree is the same as the distance from the root of the tree to the item. Using a balanced tree ensures that all items are as close to the root as possible. This makes finding an item very efficient. Adding and removing elements are equally efficient.

As an example, suppose that `coll` is any `Collection` of `Strings` (or any other type for which `compareTo()` is properly defined). We can use a `TreeSet` to sort the items of `coll` and remove the duplicates simply by saying:

```
TreeSet set = new TreeSet(); set.addAll(coll);
```

The second statement adds all the elements of the collection to the set. Since it's a `Set`, duplicates are ignored. Since it's a `TreeSet`, the elements of the set are sorted. If you would like to have the data in some other type of data structure, it's easy to copy the data from the set. For example, to place the answer in an `ArrayList`, you could say:

```
TreeSet set = new TreeSet();
set.addAll(coll);
ArrayList list = new  ArrayList();
list.addAll(set);
```

Now, in fact, every one of Java's collection classes has a constructor that takes a `Collection` as an argument. All the items in that `Collection` are added to the new collection when it is created. So, `newTreeSet(coll)` creates a `TreeSet` that contains the same elements as the `Collection`, `coll`. This means that we can abbreviate the four lines in the above example to the single command:

```
ArrayList list = new ArrayList( new TreeSet(coll) );
```

This makes a sorted list of the elements of `coll` with no duplicates. A nice example of the power of generic programming. (It seems, by the way, there is no equally easy way to get the list *with* duplicates. To do this, we would need something like a `TreeSet` that allows duplicates. The C++ programming language has such a thing and refers to it as a multiset. The Smalltalk language has something similar and calls it a bag. Java, for the time being at least, lacks this data type.)

9.5 Quiz Questions

THIS PAGE CONTAINS A SAMPLE quiz on material from Chapter 12 of this on-line Java textbook. You should be able to answer these questions after studying that chapter.

Question 1: What is meant by *generic programming* and what is the alternative?

Question 2: Java does not support generic programming with the primitive types. Why not? What is it about generic programming in Java that prevents it from working with primitive types such as `int` and `double`.

Question 3: What is an *iterator* and why are iterators necessary for generic programming?

Question 4: Suppose that `integers` is a variable of type `Collection` and that every object in the collection belongs to the wrapper class `Integer`. Write a code segment that will compute the sum of all the integer values in the collection.

Question 5: Interfaces such as `List`, `Set`, and `Map` define *abstract data types*. Explain what this means.

Question 6: What is the fundamental property that distinguishes `Sets` from other types of `Collections` ?

Question 7: What is the essential difference in functionality between a `TreeMap` and a `HashMap` ?

Question 8: Explain what is meant by a *hash code*.

Question 9: Modify the following `Date` class so that it implements the `Comparable` interface. The ordering on objects of type `Date` should be the natural, chronological ordering.

```
class Date {
   int month;  // Month number in range 1 to 12.
   int day;    // Day number in range 1 to 31.
   int year;   // Year number.
   Date(int m, int d, int y) { // Convenience constructor.
      month = m;
      day = d;
      year = y;
   }
}
```

Question 10: Suppose that `syllabus` is a variable of type `TreeMap`, the keys in the map are objects belonging to the `Date` class from the previous problem, and the values are of type `String`. Write a code segment that will write out the value string for every key that is in the month of September, 2002.

9.6 Programming Exercises

1. In Section 12.2, I mentioned that a `LinkedList` can be used as a queue by using the `addLast()` and `removeFirst()` methods to enqueue and dequeue items. But, if we are going to work with queues, it's better to have a `Queue` class. The data for the queue could still be represented as a `LinkedList`, but the `LinkedList` object would be hidden as a private instance variable in the `Queue` object. Use this idea to write a generic `Queue` class for representing queues of `Objects`. Also write a generic `Stack` class that uses either a `LinkedList` or an `ArrayList` to store its data. (Stacks and queues were introduced in Section11.3.)

2. In mathematics, several operations are defined on sets. The union of two sets A and B is a set that contains all the elements that are in A together with all the elements that are in B. The intersection of A and B is the set that contains elements that are in both A and B. The difference of A and B is the set that contains all the elements of A *except* for those elements that are also inB.

   Suppose that `A` and `B` are variables of type `set` in Java. The mathematical operations on `A` and `B` can be computed using methods from the `Set` interface. In particular: The set `A.addAll(B)` is the *union* of `A` and `B` ; `A.retainAll(B)` is the *intersection* of `A` and `B` ; and `A.removeAll(B)` is the *difference* of `A` and `B`. (These operations change the contents of the set `A`, while the mathematical operations create a new set without changing `A`, but that difference is not relevant to this exercise.)

   For this exercise, you should write a program that can be used as a "set calculator" for simple operations on sets of non-negative integers. (Negative integers are not allowed.) A set of such integers will be represented as a list of integers, separated by commas and, optionally, spaces and enclosed in square brackets. For example:

`[1,2,3]` or `[17,42,9,53,108]`. The characters `+`, `*`, and `-` will be used for the union, intersection, and difference operations. The user of the program will type in lines of input containing two sets, separated by an operator. The program should perform the operation and print the resulting set. Here are some examples:

```
 Input                                  Output
-------------------------             -------------------
 [1, 2, 3] + [3,  5,  7]               [1, 2, 3, 5, 7]
 [10,9,8,7] * [2,4,6,8]                [8]
 [ 5, 10, 15, 20 ] - [ 0, 10, 20 ]     [5, 15]
```

To represent sets of non-negative integers, use `TreeSets` containing objects belonging to the wrapper class `Integer`. Read the user's input, create two `TreeSets`, and use the appropriate `TreeSet` method to perform the requested operation on the two sets. Your program should be able to read and process any number of lines of input. If a line contains a syntax error, your program should not crash. It should report the error and move on to the next line of input. (Note: To print out a `Set`, `A`, of `Integers`, you can just say `System.out.println(A)`. I've chosen the syntax for sets to be the same as that used by the system for outputting a set.)

3. The fact that Java has a `HashMap` class means that no Java programmer has to write an implementation of hash tables from scratch – unless, of course, you are a computer science student.

   Write an implementation of hash tables from scratch. Define the following methods: `get(key)`, `put(key,value)`, `remove(key)`, `containsKey(key)`, and `size()`. Do not use *any* of Java's generic data structures. Assume that both keys and values are of type `Object`, just as for `HashMaps`. Every `Object` has a hash code, so at least you don't have to define your own hash functions. Also, you do *not* have to worry about increasing the size of the table when it becomes too full.

   You should also write a short program to test your solution.

4. A predicate is a boolean-valued function with one parameter. Some languages use predicates in generic programming. Java doesn't, but this exercise looks at how predicates might work in Java.

   In Java, we could use "predicate objects" by defining an interface:

   ```
   public interface Predicate { public boolean test(Object obj); }
   ```

   The idea is that an object that implements this interface knows how to "test" objects in some way. Define a class `Predicates` that contains the following generic methods for working with predicate objects:

   ```
   public static void remove(Collection coll, Predicate pred)
      // Remove every object, obj, from coll for which
      // pred.test(obj) is true.

   public static void retain(Collection coll, Predicate pred)
      // Remove every object, obj, from coll for which
      // pred.test(obj) is false.  (That is, retain the
      // objects for which the predicate is true.)
   ```

```
public static List collect(Collection coll, Predicate pred)
   // Return a List that contains all the objects, obj,
   // from the collection, coll, such that pred.test(obj)
   // is true.

public static int find(ArrayList list, Predicate pred)
   // Return the index of the first item in list
   // for which the predicate is true, if any.
   // If there is no such item, return -1.
```

(In C++, methods similar to these are included as a standard part of the generic programming framework.)

5. One of the examples in Section 12.4 concerns the problem of making an index for a book. A related problem is making a concordance for a document. A concordance lists every word that occurs in the document, and for each word it gives the line number of every line in the document where the word occurs. All the subroutines for creating an index that were presented in Section12.4 can also be used to create a concordance. The only real difference is that the integers in a concordance are line numbers rather than page numbers.

   Write a program that can create a concordance. The document should be read from an input file, and the concordance data should be written to an output file. The names of the input file and output file should be specified as command line arguments when the program is run. You can use the indexing subroutines from Section12.4, modified to write the data to a file instead of to `System.out`. You will also need to make the subroutines `static`. If you need some help with using files and command-line arguments, you can find an example in the sample program WordCount.java, which was also discussed in Section12.4.

   As you read the file, you want to take each word that you encounter and add it to the concordance along with the current line number. Your program will need to keep track of the line number. The end of each line in the file is marked by the newline character,'\n'. Every time you encounter this character, add one to the line number.

   Because it is so common, don't include the word "the" in your concordance. Also, do not include words that have length less than3.

Chapter 10

Correctness and Robustness

COMPUTER PROGRAMS THAT FAIL are much too common. Programs are fragile. A tiny error can cause a program to misbehave or crash. Most of us are familiar with this from our own experience with computers. And we've all heard stories about software glitches that cause spacecraft to crash, telephone service to fail, and, in a few cases, people to die.

Programs don't have to be as bad as they are. It might well be impossible to guarantee that programs are problem-free, but careful programming and well-designed programming tools can help keep the problems to a minimum. This chapter will look at issues of correctness and robustness of programs. We'll also look at exceptions, one of the tools that Java provides as an aid in writing robust programs.

10.1 Introduction to Correctness and Robustness

A PROGRAM IS correct if 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 at all 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.

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 on-going 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.

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 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 Windows computers have so many memory leaks that

the computer will run out of memory after a few days of use and will 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.

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, but the value of N has become");
         System.out.println("too large for your computer!");
         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 times 10 to the power 308. Numbers beyond this range do not "wrap around" to negative values. Instead, they are represented by special values that have no numerical equivalent. The values `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY` represent numbers outside the range of legal values. For example, 20 * 1e308 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 function `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.

Not all possible errors are 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 programmer still needs to learn techniques for avoiding and dealing with errors. These are the topics of the rest of this chapter.

10.2 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.

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 some of 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 the execution of the program. 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

```
Scanner keyboard = new Scanner(System.in);
do {
   System.out.println("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 A*x2 + B*x + 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 subroutine. As the terms are being used here, a precondition of a subroutine is just a precondition of the code that makes up the definition of the subroutine, and the postcondition of a subroutine 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:

```
Scanner sc = new Scanner(System.in);
do {
   System.out.println("Enter A, B, and C.  B*B-4*A*C must be >= 0.");
   System.out.print("A = ");
   A = sc.nextDouble();
   System.out.print("B = ");
   B = sc.nextDouble();
   System.out.print("C = ");
   C = sc.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. See the discussion at the end of the previous section.)

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

```
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);
```

10.3 Exceptions and the `try...catch` Statement

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. 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.

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. (More exactly, the thread that throws the exception will crash. In a multithreaded program, it is possible for other threads to continue even after one crashes.)

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 the case of an applet, only the current operation – such as the response to a button – will be terminated. Parts of the applet might continue to function even when other parts are non-functional because of exceptions.) In many other programming languages, a crashed program will often 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 subroutine call stack, which is a list of the subroutines that were being executed when the exception was thrown. (Since one subroutine can call another, several subroutines 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. The object thrown by an exception must be an instance of the standard class `java.lang.Throwable` or of one of its subclasses. In general, each different type of exception is represented by its own subclass of `Throwable`. `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. You should not try to catch and handle such errors. An example is the `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 usually best just to terminate the program, 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 such as: `ArithmeticException`, which occurs for example when there is an attempt to divide an integer by zero, `ArrayIndexOutOfBoundsException`, which occurs when an out-of-bounds index is used in an array, and `NullPointerException`, which occurs when there is an attempt to use a `null` reference in a context when an actual object reference is required. A `RuntimeException` generally indicates a bug in the program, which the programmer should fix. `RuntimeExceptions` and `Errors` share the prop-

erty 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.

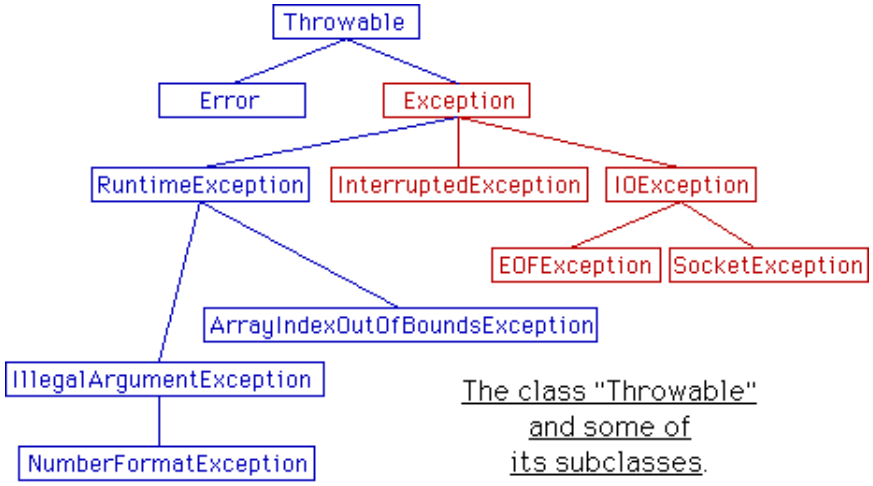


Figure 10.1: HelloWorldApplet2

To catch exceptions in a Java program, you need a `try` statement. The idea is that you tell the computer to "try" to execute some commands. If it succeeds, all well and good. But if an exception is thrown during the execution of those commands, you can catch the exception and handle it. 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 ( ArrayIndexOutOfBoundsException e ) {
   System.out.println("M is the wrong size to have a determinant.");
}
```

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 `ArrayIndexOutOfBoundsException` occurs, then the computer jumps immediately to the block of statements labeled "`catch (ArrayIndexOutOfBoundsException e)` ". 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.

You might notice that there is another possible source of error in this `try` statement. If the value of the variable `M` is `null`, then a `NullPointerException` will be thrown when the attempt is made to reference the array. In the above `try` statement, `NullPointerExceptions` are not caught, so they will be processed in the ordinary way (by terminating the program, unless the exception is handled elsewhere). You could catch `NullPointerExceptions` by adding another `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:  " + );
   System.out.println( e.getMessage() );
}
```

This example shows how to use multiple `catch` clauses in one `try` block. It also shows what that little "e" is doing in the `catch` clauses. The `e` is actually a variable name. (You can use any name you like.) Recall that when an exception occurs, it is actually an object that is thrown. Before executing a `catch` clause, the computer sets this variable to refer to the exception object that is being caught. This object contains information about the exception. For example, an error message describing the exception can be retrieved using the object's `getMessage()` method, as is done in the above example. Another useful method in every exception object, `e`, is `e.printStackTrace()` . This method will print out the list of subroutines that were being executed when the exception was thrown. This information can help you to track down the part of your program that caused the error.

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 e ) {
   System.out.println("Sorry, an error has occurred.");
   e.printStackTrace();
}
```

Since objects of type `ArrayIndexOutOfBoundsException` or `NullPointerException` is also of type `RuntimeException`, this will catch array index errors and null pointer errors as well as any other type of runtime exception. 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.

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.)

The syntax of a `try` statement is a little more complicated than I've indicated so far. The syntax can be described as

```
try {
   statements
}
optional-catch-clauses

optional-finally-clause
```

Note that this is a case where a block of statements, enclosed between { and }, is required. You need the { and } even if they enclose just one statement. The `try` statement can include zero or more `catch` clauses and, optionally, a `finally` clause. (The `try` statement must include either a `finally` clause or at least one `catch` clause.) 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.

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.

To throw an exception, use a `throw` statement. 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. (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? The answer is a little surprising: 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.)

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 subroutine and the exception is not handled in the same subroutine, then that subroutine is terminated (after the execution of any pending `finally` clauses). Then the routine that called that subroutine gets a chance to handle the exception. That is, if the subroutine 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 that routine does not handle the exception, then it also is terminated and the routine that called **it** gets the next shot at the exception. The exception will crash the program only if it passes up through the entire chain of subroutine calls without being handled.

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

```
static double root(double A, double B, double C)
                            throws IllegalArgumentException {
      // Returns the larger of the two roots of
      // the quadratic equation A*x*x + B*x + C = 0.
      // (Throws an exception if A == 0 or B*B-4*A*C < 0.)
   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 subroutine has the preconditions that `A!=0` and `B*B-4*A*C>=0`. The subroutine throws an exception of type `IllegalArgumentException` when either of these preconditions is violated. When an illegal condition is found in a subroutine, throwing an exception is often a reasonable response. If the program that called the subroutine 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.

10.3.1 Mandatory Exception Handling

In the preceding example, declaring that the subroutine `root()` can throw an `IllegalArgumentException` is just a courtesy to potential readers of this routine. This

is because handling of IllegalArgumentExceptions is not "mandatory". A routine can throw an IllegalArgumentException without announcing the possibility. And a program that calls that routine 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 subroutine can throw such an exception, that fact **must** be announced in a throws clause in the routine 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 a program can generate an exception that requires mandatory handling. The statement could be a throw statement, which throws the exception directly, or it could be a call to a subroutine 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. The second way is to declare that the subroutine that contains the statement can throw the exception. This is done by adding a "throws " clause to the subroutine heading. If the throws clause is used, then any other routine that calls the subroutine will be responsible for handling the exception. If you don't handle the possible exception in one of these two ways, it will be considered a syntax error, and the compiler will not accept your program.

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. A robust program has to be prepared to handle such conditions. The design of Java makes it impossible for programmers to ignore such conditions.

Among the exceptions that require mandatory handling are several that can occur when using Java's input/output routines. This means that you can't even use these routines unless you understand something about exception-handling. The next chapter deals with input/output and uses exception-handling extensively.

10.4 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.

10.4.1 Writing New Exception Classes

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, the new class will extend RuntimeException (or one of its subclasses) if the programmer does not want to require mandatory exception handling. 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) {
         // Constructor.  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`. See Section 5.5.) Of course the class inherits the `getMessage()` and `printStackTrace()` routines from its superclass. If `e` refers to an object of type `ParseError`, then the function 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 meaning of a string. A `ParseError` would indicate, presumably, that some string 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 subroutine that contains the `throw` statement must declare that it can throw a `ParseError`. It does this by adding the clause "`throws ParseError` " to the subroutine 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 routine 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:  Create a ShipDestroyed object
         // carrying an error message and 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.

10.4.2 Exceptions in Subroutines and Classes

The ability to throw exceptions is particularly useful in writing general-purpose subroutines and classes that are meant to be used in more than one program. In this case, the person writing the subroutine or class often has no reasonable way of handling the error, since that person has no way of knowing exactly how the subroutine 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 subroutine 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. Exceptions are a cleaner way for a subroutine to react when it encounters an error.

10.4.3 Assertions

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, 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.

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 that must 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. Assertions are not available in Java 1.3, but an assertion facility similar to the C/C++ version has been added to the language as of Java 1.4.

Even in versions of Java before 1.4, you can do something similar to assertions: You can test the condition using an if statement and throw an exception if the condition does not hold.

```
if (condition == false)
    throw new IllegalArgumentException("Assertion Failed.");
```

Of course, you could use a better error message. And it would be better style to define a new exception class instead of using the standard class `IllegalArgumentException`. This sort of test is most useful during testing and debugging of the program. Once you are sure that the program is correct, the test in the if statement might be seen as a waste of the computer's time. One advantage of 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. The nice part is that the source code doesn't have to be modified to produce the release version.

The assertion facility in Java 1.4 and later takes all this into account. A new **assert** statement is introduced into the language that has the syntax

```
assert condition: error-message;
```

The **condition** in this statement is a boolean-valued expression. The idea is that this condition is something that is supposed to be true at that point in the program, if the program is correct. The **error-message** is generally a string (though in fact it can be an expression of any type). When an *assert* statement is executed, the **expression** in the statement is evaluated. If the condition is true, the assertion has no effect and the program proceeds with the next statement. If the condition is false, then an error of type **AssertionError** is thrown, and this will cause the program to crash. The **error-message** is passed to the AssertionError object and becomes part of the error message that is printed when the program is terminated. (Of course, it's possible to catch the AssertionError to stop the program from crashing, but the whole point of an assertion is to **make** the program crash if it has gotten into some state where a necessary condition is false.)

By default, however, assert statements are **not executed**. Remember that assertions should only be executed during testing and debugging, so there has to be some way to turn them on and off. In C/C++, this is done at compile time; in Java, it is done at run time. When you run a program in the ordinary way using the *java* command, assertions in the program are ignored. To have an effect, they must be **enabled**. This is done by adding an option to the *java* command. The form of the option is "`-enableassertions:class-name`

" to enable all the assertions in a specified class or "`-enableassertions:package-name...` " to enable all the assertions in a package and in its sub-packages. 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 "`-enableassertions:...` ". You can abbreviate "-enableassertions" as "-ea", and you can use this option several times in the same command. For example, to run a Java program named "MegaPaint" with assertions enabled for the packages named "paintutils" and "drawing", you would use the command:

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

Remember that you would use the "-ea" options during development of the program, but your customers would not have to use them when they run your program.

10.5 Quiz Questions

THIS PAGE CONTAINS A SAMPLE quiz on material from Chapter 9 of this on-line Java textbook. You should be able to answer these questions after studying that chapter.

Question 1: What does it mean to say that a program is *robust?*

Question 2: Why do programming languages require that variables be declared before they are used? What does this have to do with correctness and robustness?

Question 3: What is "`Double.NaN` "?

Question 4: What is a *precondition?* Give an example.

Question 5: Explain how preconditions can be used as an aid in writing correct programs.

Question 6: Java has a predefined class called `Throwable`. What does this class represent? Why does it exist?

Question 7: Write a subroutine that prints out a `3N+1` sequence starting from a given integer, `N`. The starting value should be a parameter to the subroutine. If the parameter is less than or equal to zero, throw an `IllegalArgumentException`. If the number in the sequence becomes too large to be represented as a value of type `int`, throw an `ArithmeticException`.

Question 8: Some classes of exceptions require *mandatory exception handling*. Explain what this means.

Question 9: Consider a subroutine `processData` that has the header

```
static void processData() throws IOException
```

Write a `try...catch` statement that calls this subroutine and prints an error message if an `IOException` occurs.

Question 10: Why should a subroutine throw an exception when it encounters an error? Why not just terminate the program?

10.6 Programming Exercises

1. Write a program that uses the following subroutine, from Section 3, to solve equations specified by the user.

```
static double root(double A, double B, double C)
                              throws IllegalArgumentException {
      // Returns the larger of the two roots of
```

```
        // the quadratic equation A*x*x + B*x + C = 0.
        // (Throws an exception if A == 0 or B*B-4*A*C < 0.)
     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);
     }
  }
```

Your program should allow the user to specify values for `A`, `B`, and `C`. It should call the subroutine to compute a solution of the equation. If no error occurs, it should print the root. However, if an error occurs, your program should catch that error and print an error message. After processing one equation, the program should ask whether the user wants to enter another equation. The program should continue until the user answers no.

2. As discussed in Section 1, values of type `int` are limited to 32 bits. Integers that are too large to be represented in 32 bits cannot be stored in an `int` variable. Java has a standard class, `java.math.BigInteger`, that addresses this problem. An object of type `BigInteger` is an integer that can be arbitrarily large. (The maximum size is limited only by the amount of memory on your computer.) Since `BigIntegers` are objects, they must be manipulated using instance methods from the `BigInteger` class. For example, you can't add two `BigIntegers` with the + operator. Instead, if `N` and `M` are variables that refer to `BigIntegers`, you can compute the sum of `N` and `M` with the function call `N.add(M)`. The value returned by this function is a new `BigInteger` object that is equal to the sum of `N` and `M`.

   The `BigInteger` class has a constructor `new BigInteger(str)`, where `str` is a string. The string must represent an integer, such as "3" or "39849823783783283733". If the string does not represent a legal integer, then the constructor throws a `NumberFormatException`.

   There are many instance methods in the `BigInteger` class. Here are a few that you will find useful for this exercise. Assume that `N` and `M` are variables of type `BigInteger`.

   `N.add(M)` – a function that returns a `BigInteger` representing the sum of `N` and `M`.

   `N.multiply(M)` – a function that returns a `BigInteger` representing the result of multiplying `N` times `M`.

   `N.divide(M)` – a function that returns a `BigInteger` representing the result of dividing `N` by `M`.

   `N.signum()` – a function that returns an ordinary `int`. The returned value represents the sign of the integer `N`. The returned value is 1 if `N` is greater than zero. It is -1 if `N` is less than zero. And it is 0 if `N` is zero.

   `N.equals(M)` – a function that returns a `boolean` value that is `true` if `N` and `M` have the same integer value.

`N.toString()` – a function that returns a `String` representing the value of `N`.

`N.testBit(k)` – a function that returns a `boolean` value. The parameter `k` is an integer. The return value is `true` if the `k` -th bit in `N` is 1, and it is `false` if the `k` -th bit is 0. Bits are numbered from right to left, starting with 0. Testing "`if (N.testBit(0))` " is an easy way to check whether `N` is even or odd. `N.testBit(0)` is `true` if and only if `N` is an odd number.

For this exercise, you should write a program that prints `3N+1` sequences with starting values specified by the user. In this version of the program, you should use `BigIntegers` to represent the terms in the sequence.

If the user's input is legal, print out the `3N+1` sequence. Count the number of terms in the sequence, and print the count at the end of the sequence. Exit the program when the user inputs an empty line.

3. A Roman numeral represents an integer using letters. Examples are XVII to represent 17, MCMLIII for 1953, and MMMCCCIII for 3303. By contrast, ordinary numbers such as 17 or 1953 are called Arabic numerals. The following table shows the Arabic equivalent of all the single-letter Roman numerals:

M	1000	X	10
D	500	V	5
C	100	I	1
L	50		

When letters are strung together, the values of the letters are just added up, with the following exception. When a letter of smaller value is followed by a letter of larger value, the smaller value is subtracted from the larger value. For example, IV represents 5 - 1, or 4. And MCMXCV is interpreted as M + CM + XC + V, or 1000 + (1000 - 100) + (100 - 10) + 5, which is 1995. In standard Roman numerals, no more than thee consecutive copies of the same letter are used. Following these rules, every number between 1 and 3999 can be represented as a Roman numeral made up of the following one- and two-letter combinations:

M	1000	X	10
CM	900	IX	9
D	500	V	5
CD	400	IV	4
C	100	I	1
XC	90		
L	50		
XL	40		

Write a class to represent Roman numerals. The class should have two constructors. One constructs a Roman numeral from a string such as "XVII" or "MCMXCV". It should throw a `NumberFormatException` if the string is not a legal Roman numeral. The other constructor constructs a Roman numeral from an `int`. It should throw a `NumberFormatException` if the `int` is outside the range 1 to 3999.

In addition, the class should have two instance methods. The method `toString()` returns the string that represents the Roman numeral. The method `toInt()` returns the value of the Roman numeral as an `int`.

At some point in your class, you will have to convert an `int` into the string that represents the corresponding Roman numeral. One way to approach this is to gradually "move" value from the Arabic numeral to the Roman numeral. Here is the beginning of a routine that will do this, where `number` is the `int` that is to be converted:

```
String roman = "";
int N = number;
while (N >= 1000) {
      // Move 1000 from N to roman.
   roman += "M";
   N -= 1000;
}
while (N >= 900) {
      // Move 900 from N to roman.
   roman += "CM";
   N -= 900;
}
.
.  // Continue with other values from the above table.
.
```

(You can save yourself a lot of typing in this routine if you use arrays in a clever way to represent the data in the above table.)

Once you've written your class, use it in a main program that will read both Arabic numerals and Roman numerals entered by the user. If the user enters an Arabic numeral, print the corresponding Roman numeral. If the user enters a Roman numeral, print the corresponding Arabic numeral. The program should end when the user inputs an empty line.

4. The file Expr.java defines a class, `Expr`, that can be used to represent mathematical expressions involving the variable `x`. The expression can use the operators `+`, `-`, `*`, `/`, and `^`, where `^` represents the operation of raising a number to a power. It can use mathematical functions such as `sin`, `cos`, `abs`, and `ln`. See the source code file for full details. The `Expr` class uses some advanced techniques which have not yet been covered in this textbook. However, the interface is easy to understand. It contains only a constructor and two public methods.

   The constructor `new Expr(def)` creates an `Expr` object defined by a given expression. The parameter, `def`, is a string that contains the definition. For example, `new Expr(''x^2'')` or `new Expr(''sin(x)+3*x'')`. If the parameter in the constructor call does not represent a legal expression, then the constructor throws an `IllegalArgumentException`. The message in the exception describes the error.

   If `func` is a variable of type `Expr` and `num` is of type `double`, then `func.value(num)` is a function that returns the value of the expression when the number `num` is substituted for the variable `x` in the expression. For example, if `Expr` represents the

expression `3*x+1`, then `func.value(5)` is `3*5+1`, or 16. If the expression is undefined for the specified value of `x`, then the special value `Double.NaN` is returned.

Finally, `func.getDefinition()` returns the definition of the expression. This is just the string that was used in the constructor that created the expression object.

For this exercise, you should write a program that lets the user enter an expression. If the expression contains an error, print an error message. Otherwise, let the user enter some numerical values for the variable `x`. Print the value of the expression for each number that the user enters. However, if the expression is undefined for the specified value of `x`, print a message to that effect. You can use the `boolean` -valued function `Double.isNaN(val)` to check whether a number, `val`, is `Double.NaN`.

The user should be able to enter as many values of `x` as desired. After that, the user should be able to enter a new expression. Here is an applet that simulates my solution to this exercise, so that you can see how it works:

5. This exercises uses the class `Expr`, which was described in Exercise9.4. For this exercise, you should write an applet that can graph a function, `f(x)`, whose definition is entered by the user. The applet should have a text-input box where the user can enter an expression involving the variable `x`, such as `x^2` or `sin(x-3)/x`. This expression is the definition of the function. When the user presses return in the text input box, the applet should use the contents of the text input box to construct an object of type `Expr`. If an error is found in the definition, then the applet should display an error message. Otherwise, it should display a graph of the function. (Note: A `JTextField` generates an `ActionEvent` when the user presses return.)

   The applet will need a `JPanel` for displaying the graph. To keep things simple, this panel should represent a fixed region in the xy-plane, defined by `-5<=x<=5` and `-5<=y<=5`. To draw the graph, compute a large number of points and connect them with line segments. (This method does not handle discontinuous functions properly; doing so is very hard, so you shouldn't try to do it for this exercise.) My applet divides the interval `-5<=x<=5` into 300 subintervals and uses the 301 endpoints of these subintervals for drawing the graph. Note that the function might be undefined at one of these `x` -values. In that case, you have to skip that point.

   A point on the graph has the form `(x,y)` where `y` is obtained by evaluating the user's expression at the given value of `x`. You will have to convert these real numbers to the integer coordinates of the corresponding pixel on the canvas. The formulas for the conversion are:

```
a  =  (int)( (x + 5)/10 * width );
b  =  (int)( (5 - y)/10 * height );
```

   where `a` and `b` are the horizontal and vertical coordinates of the pixel, and `width` and `height` are the width and height of the canvas.

Chapter 11

Input/Output

COMPUTER PROGRAMS ARE ONLY USEFUL if they interact with the rest of the world in some way. This interaction is referred to as input/output, or I/O. Up until now, the only type of interaction that has been covered in this textbook is interaction with the user, through either a graphical user interface or a command-line interface. But the user is only one possible source of information and only one possible destination for information. In this chapter, we'll look at others, including files and network connections. In Java, input/output involving files and networks is based on streams, which are objects that support the same sort of I/O commands that you have already used to communicate with the user in a command-line interface. In fact, standard output (`System.out`) and standard input (`System.in`) are examples of streams.

Working with files and networks requires familiarity with exceptions, which were introduced in the previous chapter. Many of the subroutines that are used can throw exceptions that require mandatory exception handling. This generally means calling the subroutine in a `try...catch` statement that can deal with the exception if one occurs.

11.1 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 I/O abstractions are called streams. This section is an introduction to streams, but it is not meant to cover them in full detail. See the official Java documentation for more information.

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 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.

Every 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 `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.)

It's usually easy to decide whether to use byte streams or character streams. If you want the data to be human-readable, use character streams. Otherwise, use byte streams. I should note that Java 1.0 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, as of Java 1.1, 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 putting the directive "`import java.io.*;` " at the beginning of your source file. Streams are not used in Java's graphical user interface, which has its own form of I/O. But they 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 (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 `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 subroutine 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

```
public void write(int b) throws IOException
```

but again, in practice, you will almost always use higher-level output operations defined in some subclass of OutputStream.

The Reader and Writer classes provide very similar low-level read and write operations. But in these character-oriented classes, the I/O operations read and write char values rather than bytes. In practice, you will use sub-classes of Reader and Writer, as discussed below.

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 PrintWriter's advanced data output methods, 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, the output methods of the PrintWriter class include:

```
public void print(String s)   // Methods for outputting
public void print(char c)     //    standard data types
public void print(int i)      //    to the stream, in
public void print(long l)     //    human-readable form.
public void print(float f)
public void print(double d)
public void print(boolean b)

public void println()  // Output a carriage return to the stream.

public void println(String s)    // These methods are identical
public void println(char c)      //    to the previous set,
public void println(int i)       //    except that the output
public void println(long l)      //    value is followed by
```

```
public void println(float f)     //    a carriage return.
public void println(double d)
public void println(boolean b)
```

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 every time you use a `PrintWriter` method.

When you use `PrintWriter` methods 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 `OutputStream`, you could say

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

to wrap `byteSink` in a `DataOutputStream`, `dataSink`.

For machine-readable data, such as that created by writing to a `DataOutputStream`, `java.io` provides the class `DataInputStream`. You can wrap any `InputStream` opbject in a `DataInputStream` object to provide it with the ability to read data of various types from the byte-stream. The methods in the `DataInputStream` 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.

The classes `PrintWriter`, `DataInputStream`, and `DataOutputStream` allow you to easily input and output all of Java's primitive data types. But what happens when you want to read and write objects? Traditionally, you would have to come up with some way of encoding your object as a sequence of data values belonging to the primitive types, which can then be output as bytes or characters. This is called serializing the object. On input, you have to read the serialized data and somehow reconstitute a copy of the original object. For complex objects, this can all be a major chore. However, you can get Java to do all the work for you by using the classes `ObjectInputStream` and `ObjectOutputStream`. These are subclasses of `InputStream` and `Outputstream` that can be used for writing and reading serialized objects.

`ObjectInputStream` and `ObjectOutputStream` are wrapper classes that can be wrapped around arbitrary `InputStreams` and `OutputStreams`. This makes it possible to do object

input and output on any byte-stream. The methods for object I/O are `readObject()`, in `ObjectInputStream`, and `writeObject(Object obj)`, in `ObjectOutputStream`. Both of these methods can throw `IOExceptions`. Note that `readObject()` returns a value of type `Object`, which generally has to be type-cast to a more useful type.

`ObjectInputStream` and `ObjectOutputStream` only work with objects that implement an interface named `Serializable`. Furthermore, all of the instance variables in the object must be serializable. However, there is little work involved in making an object serializable, since the `Serializable` interface does not declare any methods. It exists only as a marker for the compiler, to tell it that the object is meant to be writable and readable. You only need to add the words "`implements Serializable` " to your class definitions. Many of Java's standard classes are already declared to be serializable, including all the component classes in Swing and in the AWT. This means, in particular, that GUI components can be written to `ObjectOutputStreams` and read from `ObjectInputStreams`.

11.2 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 floppy disk, 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 is 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 generally not allowed to access files. 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.

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. This exception type requires mandatory exception handling, so you have to call the constructor in a `try` statement (or inside a routine that is declared to throw `FileNotFoundException`). 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` object or in some other wrapper class.

```
Scanner data;

try {
     data = new Scanner(new FileReader("data.dat"));
}
catch  (FileNotFoundException e) {
     ... // handle the exception
}
```

Once you have a `Scanner` named `data`, you can read from it using such methods as `data.nextInt()` and `data.next()`, exactly as you would from any other `Scanner`.

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` 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. An `IOException` might occur 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. (If you forget to do this, the file will ordinarily be closed automatically when the program terminates or when the file stream object is garbage collected, but it's best to close a file as soon as you are done with 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; however,`PrintWriter` overrides this method so that it cannot throw such exceptions.)

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, and that there are no more than 1000 numbers altogether. 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.*;
public class ReverseFile {

   public static void main(String[] args) {

      Scanner data;     // Character input stream for reading data.
      PrintWriter result;  // Character output stream for writing data.

      double[] number = new double[1000];  // An array to hold all
                                           //   the numbers that are
                                           //   read from the file.

      int numberCt;  // Number of items actually stored in the array.

      try {  // Create the input stream.
         data = new Scanner(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(e.toString());
         data.close();  // Close the input file.
         return;        // End the program.
      }
```

```
        try {

            // Read the data from the input file.

            numberCt = 0;
            while (data.hasNext()) {  // Read until end-of-file.
               number[numberCt] = data.nextDouble();
               numberCt++;
            }

            // Output the numbers in reverse order.

            for (int i = numberCt-1; i >= 0; i--)
               result.println(number[i]);

            System.out.println("Done!");

         }
         catch (IOException e) {
            // Some problem reading the data from the input file.
            System.out.println("Input Error: " + e.getMessage());
         }
         catch (IndexOutOfBoundsException e) {
            // Must have tried to put too many numbers in the array.
            System.out.println("Too many numbers in data file.");
            System.out.println("Processing has been aborted.");
         }
         finally {
            // Finish by closing the files,
            //     whatever else may have happened.
            data.close();
            result.close();
         }

      }  // end of main()

   } // end of class
```

11.2.1 File Names, Directories, and the File Class

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 (or "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 its name is. A relative path name tells the computer how to locate the file, starting from the current directory.

Unfortunately, the syntax for file names and path names varies quite a bit from one type of computer to another. Here are some examples:

- `data.dat` – on any computer, this would be a file named data.dat in the current directory.
- `/home/eck/java/examples/data.dat` – This is an absolute path name in the UNIX operating system. It refers to a file named data.dat in a directory named examples, which is in turn in a directory named java,....
- `C:\eck\java\examples\data.dat` – An absolute path name on a DOS or Windows computer.
- `Hard Drive:java:examples:data.dat` – Assuming that "Hard Drive" is the name of a disk drive, this would be an absolute path name on a computer using Macintosh OS9.
- `examples/data.dat` – a relative path name under UNIX. "Examples" is the name of a directory that is contained within the current directory, and data.data is a file in that directory. The corresponding relative path names for Windows and Macintosh would be `examples\data.dat` and `examples:data.dat`.

Similarly, the rules for determining which directory is the current directory are different for different types of computers. 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.

To avoid some of the problems caused by differences 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, `new File(File,String)`, has two parameters. The first is a `File` object that refers to the directory that contains the file. The second is the name of the file. Later in this section, we'll look at a convenient way of letting the user specify a `File` in a GUI program.

`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 function returns `true` if the file named by the `File` object already exists. You could use this method if you wanted to avoid overwriting the contents of an existing file when you create a new `FileWriter`.

`file.isDirectory()` – This `boolean` -valued function 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.

`file.list()` – If the `File` object refers to a directory, this function 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:

```
import java.io.File;

public class DirectoryList {

   /* 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 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 keyboard = new Scanner(System.in);

      System.out.print("Enter a directory name: ");
      directoryName = keyboard.next();
      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 `Scanner` to read from the file, you might use:

```
Scanner data;

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

11.2.2 File Dialog Boxes

In many programs, you want the user to be able to select the file that is going to be used for input or output. If your program lets the user type in the file name, you will just have to assume that the user understands how to work with files and directories. But in a graphical user interface, the user expects to be able to select files using a file dialog box, which is a special window that a program can open when it wants the user to select a file for input or output. Swing includes a platform-independent technique for using file dialog boxes in the form of a class called `JFileChooser`. This class is part of the package `javax.swing`. We looked at some basic dialog boxes previously. File dialog boxes are similar to those, but are a little more complicated to use.

A file dialog box shows the user a list of files and sub-directories in some directory, and makes it easy for the user to specify a file in that directory. The user can also navigate easily from one directory to another. The most common constructors for `JFileChooser` specify the directory that is selected when the dialog box first appears:

```
new JFileChooser( File startDirectory )

new JFileChooser( String pathToStartDirectory )
```

There is also a constructor with no arguments that will set the user's home directory to be the starting directory in the dialog box. (The constructor call `new JFileChooser(``.'')` produces a dialog box that has the current directory as its starting directory. This is true since "." is a special path name that refers to the current directory, at least on Windows and UNIX systems.)

Constructing a `JFileChooser` object does not make the dialog box appear on the screen. You have to call a methods in the object to do that. There are two different methods that can be used because there are two types of file dialog: An open file dialog allows the user to specify an existing file to be opened for reading data into the program; a save file dialog lets the user specify a file, which might or might not already exist, to be opened for writing data from the program. File dialogs of these two types are opened using the `showOpenDialog` and `showSaveDialog` methods.

A file dialog box always has a parent, another component which is associated with the dialog box. The parent is specified as a parameter to the `showOpenDialog` or `showSaveDialog` methods. The parent is a GUI component, and can usually be specified as "`this` ". (The parameter can be `null`, in which case an invisible component is used as the parent.) Both `showOpenDialog` and `showSaveDialog` have a return value, which will be one of the constants `JFileChooser.CANCEL_OPTION`, `JFileChooser.ERROR_OPTION` or `JFileChooser.APPROVE_OPTION`. If the return value is `JFileChooser.APPROVE_OPTION`, then the user has selected a file. If the return value is something else, then the user did

not select a file. The user might have clicked a "Cancel" button, for example. You should always check the return value, to make sure that the user has, in fact, selected a file. If that is the case, then you can find out which file was selected by calling the `JFileChooser`'s `getFile()` method, which returns an object of type `File` that represents the selected file.

Putting all this together, typical code for using a `JFileChooser` to read character data from a file looks like this:

```
JFileChooser fileDialog = new JFileChooser(".");
int option = fileDialog.showOpenDialog(this);
if (option == JFileChooser.APPROVE_OPTION) {
   File selectedFile = fileDialog.getFile();
   try {
      Scanner data = new Scanner(new FileReader(selectedFile));
   }
   catch (FileNotFoundException e) {
      // Handle the error.
   }
   .
   .  // Read data from the file.
   .
}
```

The first line creates a new `JFileChooser` object in which the current directory is initially selected. The second line shows the file dialog box on the screen and waits for the user to select a file or close the dialog box in some other way. The third line tests whether the user has actually selected a file. Only in that case do we proceed to get the selected file, open it, and use it. Writing data to a file would be similar, but `showSaveDialog` would replace `showOpenDialog`.

There is nothing to stop you, by the way, from using the same `JFileChooser` object over and over. This would have the advantage that the selected directory would be remembered from one use to the next.

It's common to do some configuration of a `JFileChooser` before calling `showOpenDialog` or `showSaveDialog`. For example, the instance method `setDialogTitle(String)` can be used to specify a title to appear at the top of the window. And `setSelectedFile(File)` can be used to set the file that is selected in the dialog box when it appears. This can be used to provide a default file choice for the user.

We'll look at some more complete examples of using files and file dialogs in the next section.

11.3 Programming With Files

IN THIS SECTION, we look at several programming examples that work with files.

The first example is a program that makes a list of all the words that occur in a specified file. The user is asked to type in the name of the file. The list of words is written to another file, which the user also specifies. A word here just means a sequence of letters. The list of words will be output in alphabetical order, with no repetitions. All the words will be converted into lower case, so that, for example, "The" and "the" will count as the same word.

Since we want to output the words in alphabetical order, we can't just output the words as they are read from the input file. We can store the words in an array, but since there is

no way to tell in advance how many words will be found in the file, we need a "dynamic array" which can grow as large as necessary. The data is represented in the program by two static variables:

```
static String[] words; // An array that holds the words.
static int wordCount; // The number of words currently
                      // stored in the array.
```

The program starts with an empty array. Every time a word is read from the file, it is inserted into the array (if it is not already there). The array is kept at all times in alphabetical order, so the new word has to be inserted into its proper position in that order. The insertion is done by the following subroutine:

```
static void insertWord(String w) {

   int pos = 0;  // This will be the position in the array
                 //      where the word w belongs.

   w = w.toLowerCase();  // Convert word to lower case.

   /* Find the position in the array where w belongs, after all the
      words that precede w alphabetically.  If a copy of w already
      occupies that position, then it is not necessary to insert
      w, so return immediately. */

   while (pos < wordCount && words[pos].compareTo(w) < 0)
      pos++;
   if (pos < wordCount && words[pos].equals(w))
      return;

   /* If the array is full, make a new array that is twice as
       big, copy all the words from the old array to the new,
       and set the variable, words, to refer to the new array. */

   if (wordCount == words.length) {
      String[] newWords = new String[words.length*2];
      System.arraycopy(words,0,newWords,0,wordCount);
      words = newWords;
   }

   /* Put w into its correct position in the array.  Move any
      words that come after w up one space in the array to
      make room for w. */
   for (int i = wordCount; i > pos; i--)
      words[i] = words[i-1];
   words[pos] = w;
   wordCount++;

}  // end insertWord()
```

This subroutine is called by the `main()` routine of the program to process each word that it reads from the file. If we ignore the possibility of errors, an algorithm for the program is

```
Get the file names from the user
Create a Scanner for reading from the input file
Create a PrintWriter for writing to the output file
while there are more words in the input file:
   Read a word from the input file
   Insert the word into the words array
For i from 0 to wordCount - 1:
   Write words[i] to the output file
```

Most of these steps can generate `IOExceptions`, and so they must be done inside `try...catch` statements. In this case, we'll just print an error message and terminate the program when an error occurs.

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

   Scanner in;    // A stream for reading from the input file.
   Scanner keyboard = new Scanner(System.in);
   PrintWriter out;  // A stream for writing to the output file.

   String inputFileName;   // Input file name, specified by the user.

   String outputFileName;  // Output file name, specified by the user.

   words = new String[10];  // Start with space for 10 words.

   wordCount = 0;           // Currently, there are no words in array.


   /* Get the input file name from the user and try to create the
      input stream.  If there is a FileNotFoundException, print
      a message and terminate the program. */

   System.out.print("Input file name?  ");
   inputFileName = keyboard.next();
   try {
      in = new Scanner(new FileReader(inputFileName));
   }
   catch (FileNotFoundException e) {
       System.out.println("Can't find file \"" + inputFileName + "\".");
       return;  // Returning from main() ends the program.
   }

   /* Get the output file name from the user and try to create the
      output stream.  If there is an IOException, print a message
      and terminate the program. */

   System.out.print("Output file name? ");
```

```
        outputFileName = keyboard.next();
        try {
           out = new PrintWriter(new FileWriter(outputFileName));
        }
        catch (IOException e) {
            System.out.println("Can't open file \"" +
                                    outputFileName + "\" for output.");
            System.out.println(e.toString());
            return;
        }

        /* Read all the words from the input stream and insert them into
           the array of words.  */

        try {
           while (in.hasNext()) {
              insertWord(in.next());
           }
        }
        catch (IOException e) {
           System.out.println("An error occurred while reading from input file.");
           System.out.println(e.toString());
        }

        /* Write all the words from the list to the output stream. */

        for (int i = 0; i < wordCount; i++)
           out.println(words[i]);

        /* Finish up by checking for an error on the output stream and
           printing either a warning message or a message that the words
           have been output to the output file.  The PrintWriter class
           does not throw an exception when an error occurs, so we have
           to check for errors by calling the checkError() method. */

        if (out.checkError() == true) {
           System.out.println("Some error occurred while writing output.");
           System.out.println("Output might be incomplete or invalid.");
        }
        else {
           System.out.println(wordCount + " words from \"" + inputFileName +
                         "\" output to \"" + outputFileName + "\".");
        }

    } // end main()
```

Making a copy of a file is a pretty common operation, and most operating systems already have a command for doing so. 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 function `source.read()` can be used to read one byte. This function 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 DOS or 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()` routine. 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.*;

public class CopyFile {

   public static void main(String[] args) {

      String sourceName;   // Name of the source file,
                           //    as specified on the command line.
```

```
String copyName;     // Name of the copy,
                     //    as 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(e.toString());
        }

     }  // end main()

  } // end class CopyFile
```

Both of the previous programs use a command-line interface, but graphical user interface programs can also manipulate files. Programs typically have an "Open" command that reads the data from a file and displays it in a window and a "Save" command that writes the data from the window into a file. We can illustrate this in Java with a simple text editor program. The window for this program uses a `JTextArea` component to dis-

play some text that the user can edit. It also has a menu bar, with a "File" menu that includes "Open" and "Save" commands. To fully understand the examples in the rest of this section, you must be familiar with the material on menus and frames from Section7.7 and Section7.7. The examples also use file dialogs, which were introduced in Section2.

When the user selects the Save command from the File menu in the `TrivialEdit` program, the program pops up a file dialog box where the user specifies the file. The text from the `JTextArea` is written to the file. All this is done in the following instance method (where the variable, `text`, refers to the `TextArea`):

```
private void doSave() {
       // Carry out the Save command by letting the user specify
       // an output file and writing the text from the TextArea
       // to that file.
   File file;  // The file that the user wants to save.
   JFileChooser fd; // File dialog that lets the user specify the file.
   fd = new JFileChooser(".");
   fd.setDialogTitle("Save Text As...");
   int action = fd.showSaveDialog(this);
   if (action != JFileChooser.APPROVE_OPTION) {
          // User has canceled, or an error occurred.
      return;
   }
   file = fd.getSelectedFile();
   if (file.exists()) {
         // If file already exists, ask before replacing it.
      action = JOptionPane.showConfirmDialog(this,
                                    "Replace existing file?");
      if (action != JOptionPane.YES_OPTION)
         return;
   }
   try {
         // Create a PrintWriter for writing to the specified
         // file and write the text from the window to that stream.
      PrintWriter out = new PrintWriter(new FileWriter(file));
      String contents = text.getText();
      out.print(contents);
      if (out.checkError())
         throw new IOException("Error while writing to file.");
      out.close();
   }
   catch (IOException e) {
         // Some error has occurred while trying to write.
         // Show an error message.
      JOptionPane.showMessageDialog(this,
          "Sorry, an error has occurred:\n" + e.getMessage());
   }
}
```

When the user selects the Open command, a file dialog box allows the user to specify the file that is to be opened. It is assumed that the file is a text file. Since `JTextAreas`

are not meant for displaying large amounts of text, the number of lines read from the file is limited to one hundred at most. Before the file is read, any text currently in the `JTextArea` is removed. Then lines are read from the file and appended to the `JTextArea` one by one, with a line feed character at the end of each line. This process continues until one hundred lines have been read or until the end of the input file is reached. If any error occurs during this process, an error message is displayed to the user in a dialog box. Here is the complete method:

```
private void doOpen() {
       // Carry out the Open command by letting the user specify
       // the file to be opened and reading up to 100 lines from
       // that file.  The text from the file replaces the text
       // in the JTextArea.
   File file;  // The file that the user wants to open.
   JFileChooser fd; // File dialog that lets the user specify a file.
   fd = new JFileChooser(new File("."));
   fd.setDialogTitle("Open File...");
   int action = fd.showOpenDialog(this);
   if (action != JFileChooser.APPROVE_OPTION) {
         // User canceled the dialog, or an error occurred.
       return;
   }
   file = fd.getSelectedFile();
   try {
          // Read lines from the file until end-of-file is detected,
          // or until 100 lines have been read.  The lines are added
          // to the JTextArea, with a line feed after each line.
      Scanner in = new Scanner(new FileReader(file));
      String line;
      text.setText("");
      int lineCt = 0;
      while (lineCt < 100 && in.peek() != '\0') {
         line = in.next();
         text.append(line + '\n');
         lineCt++;
      }
      if (hasNext())
         text.append("\n\n****** Text truncated to 100 lines! ******\n");
      in.close();
   }
   catch (Exception e) {
         // Some error has occurred while trying to read the file.
         // Show an error message.
      JOptionPane.showMessageDialog(this,
            "Sorry, some error occurred:\n" + e.getMessage());
   }
}
```

The `doSave()` and `doOpen()` methods are the only part of the text editor program that deal with files. If you would like to see the entire program, you will find the source code in the file TrivialEdit.java.

For a final example of files used in a complete program, you might want to look at ShapeDrawWithFiles.java. This file defines one last version of the ShapeDraw program. This version has a "File" menu for saving and loading the patterns of shapes that are created with the program. The program also serves as an example of using `ObjectInputStream` and `ObjectOutputStream`. If you check, you'll see that the `Shape` class in this version has been declared to be `Serializable` so that objects of type `Shape` can be written to and read from object streams.

11.4 Quiz Questions

THIS PAGE CONTAINS A SAMPLE quiz on material from Chapter 10 of this on-line Java textbook. You should be able to answer these questions after studying that chapter.

Question 1: In Java, input/output is done using streams. Streams are an *abstraction*. Explain what this means and why it is important.

Question 2: Java has two types of streams: character streams and byte streams. Why? What is the difference between the two types of streams?

Question 3: What is a *file?* Why are files necessary?

Question 4: What is the point of the following statement?

```
out = new PrintWriter( new FileWriter("data.dat") );
```

Why would you need a statement that involves two different stream classes, `PrintWriter` and `FileWriter` ?

Question 5: The package `java.io` includes a class named `URL`. What does an object of type `URL` represent, and how is it used?

Question 6: Explain what is meant by the *client / server* model of network communication.

Question 7: What is a *Socket?*

Question 8: What is a *ServerSocket* and how is it used?

Question 9: Network server programs are often *multithreaded.* Explain what this means and why it is true.

Question 10: Write a complete program that will display the first ten lines from a text file. The lines should be written to standard output, `System.out`. The file name is given as the command-line argument `args[0]`. You can assume that the file contains at least ten lines. Don't bother to make the program robust.

11.5 Programming Exercises

1. The `WordList` program from Section 10.3 reads a text file and makes an alphabetical list of all the words in that file. The list of words is output to another file. Improve the program so that it also keeps track of the number of times that each word occurs in the file. Write two lists to the output file. The first list contains the words in alphabetical order. The number of times that the word occurred in the file should be listed along with the word. Then write a second list to the output file in which the words are sorted according to the number of times that they occurred in the files. The word that occurred most often should be listed first.

2. Write a program that will count the number of lines in each file that is specified on the command line. Assume that the files are text files. Note that multiple files can be

specified, as in "`java LineCounts file1.txt file2.txt file3.txt`". Write each file name, along with the number of lines in that file, to standard output. If an error occurs while trying to read from one of the files, you should print an error message for that file, but you should still process all the remaining files.

3. A `PhoneDirectory` holds a list of names and associated phone numbers. But a phone directory is pretty useless unless the data in the directory can be saved permanently – that is, in a file. Write a phone directory program that keeps its list of names and phone numbers in a file. The user of the program should be able to look up a name in the directory to find the associated phone number. The user should also be able to make changes to the data in the directory. Every time the program starts up, it should read the data from the file. Before the program terminates, if the data has been changed while the program was running, the file should be re-written with the new data. Designing a user interface for the program is part of the exercise.