What Is Programming Anyway?

Most of the machines that have been developed to improve our lives serve a single purpose. Just try to drive to the store in your washing machine or vacuum the living room with your car and this becomes quite clear. Your computer, by contrast, serves many functions. In the office or library, you may find it invaluable as a word processor. Once you get home, slip in a DVD and your computer takes on the role of a television. Start up a flight simulator and it assumes the properties of anything from a hang glider to the Concorde. Launch an mp3 player and you suddenly have a music system. This, of course, is just a short sample of the functions a typical personal computer can perform. Clearly, the computer is a very flexible device.

While the computer's ability to switch from one role to another is itself amazing, it is even more startling that these transformations occur without major physical changes in the machine. Every computer system includes both hardware, the physical circuitry of which the machine is constructed, and software, the programs that determine how the machine will behave. Everything described above can be accomplished by changing the software used without changing the machine's actual circuitry in any way. In fact, the changes that occur when you switch to a new program are often greater than those you achieve by changing a computer's hardware. If you install more memory or a faster network card, the computer will still do pretty much the same things it did before but a bit faster (hopefully!). On the other hand, by downloading a new application program through your Web browser, you can make it possible for your computer to perform completely new functions.

Software clearly plays a central role in the amazing success of computer technology. Very few computer users, however, have a clear understanding of what software really is. This book
Chapter 1 What Is Programming Anyway

provides an introduction to the design and construction of computer software in the programming language named Java. By learning to program in Java, you will acquire a useful skill that will enable you to construct software of your own or participate in the implementation or maintenance of commercial software. More importantly, you will gain a clear understanding of what a program really is and how it is possible to radically change the behavior of a computer by constructing new programs.

A program is a set of instructions that a computer follows. We can therefore learn a lot about computer programs by examining the ways in which instructions written for humans resemble and differ from computer programs. In this chapter we will consider several examples of instructions for humans in order to provide you with a rudimentary understanding of the nature of a computer program. We will then build on this understanding by presenting a very simple but complete example of a computer program written in Java. Like instructions for humans the instructions that make up a computer program must be communicated to the computer in language that it comprehends. Java is such a language. We will discuss the mechanics of actual communicating the text of a Java program to a computer so that it can follow the instructions contained in the program. Finally, you have undoubtedly already discovered that programs don’t always do what you expect them to do. When someone else’s program misbehaves, you can complain. When this happens with a program you wrote yourself, you will have to figure out how to change the instructions to eliminate the problem. To prepare you for this task, we will conclude this chapter by identifying several categories of errors that can be made when writing a program.

1.1 Without Understanding

You have certainly had the experience of following instructions of one sort or another. Electronic devices from computers to cameras come with thick manuals of instructions. Forms, whether they be tax forms or the answer sheets for an SAT exam, come with instructions explaining how the should be completed. You can easily think of many other examples of instructions you have had to follow.

If you have had to follow instructions, it is likely that you have also complained about the quality of the instructions. The most common complaint is probably that the instructions took too long to read. This, however, may have more to do with our impatience than with the quality of the instructions. A more serious complaint is that instructions are often unclear and hard to understand. It seems obvious that instructions are more likely to be followed correctly if they are easy to understand. This "obvious" fact, however, does not generalize to the types of instruction that make up computer programs. A computer is just a machine. Understanding is something humans do, but not something machines do. How can a computer understand the instructions in a computer program? The simple answer is that it cannot. As a result, the instructions that make up a computer program have to satisfy a very challenging requirement. It must be possible to follow them correctly without actually understanding them.

This may seem like a preposterous idea. How can you follow instructions if you don’t understand them? Fortunately, there are a few examples of instructions for humans that are deliberately designed so that they can be followed without understanding. Examining such instructions will give you a bit of insight into how a computer must follow the instructions in a computer program.

Section 1.1 Without Understanding

First, consider the "mathematical puzzle" described below. To appreciate this example, don’t just read the instructions. Follow them as you read them.

1. Pick a number between 1 and 40.
2. Subtract 20 from the number you picked.
4. Square the result.
5. Add up the individual digits of the result.
6. If the sum of the digits is even, divide by 2.
7. If the result is less than 5 add 5, otherwise subtract 4.
8. Multiply by 2.
10. Find the letter whose position in the alphabet is equal to the number you obtained (a = 1, b = 2, c = 3, etc.).
11. Think of a country whose name begins with this letter.
12. Think of a large mammal whose name begins with the second letter of the country’s name.

You have probably seen puzzles like this before. You are supposed to be surprised that it is possible to predict the final result produced, even though you are allowed to make random choices at some points in the process. In particular, this puzzle is designed to leave you thinking about elephants. Were you thinking about an elephant when you finished? Are you surprised we could predict this?

The underlying reason for such surprise is that the instructions are designed to be followed without being understood. The person following the instructions thinks that the choices he or she gets to make in the process (choosing a number or choosing any country whose name begins with “D”) could lead to many different results. A person who understands the instructions realizes this is an illusion.

To understand why almost everyone who follows the instructions above will end up thinking about elephants, you have to identify a number of properties of the operations performed. The steps that tell you to multiply by 3 and square the result ensure that after these steps the number you are working with will be a multiple of nine. When you add up the digits of any number that is a multiple of nine, the sum will also be a multiple of nine. Furthermore, the fact that your initial number was relatively small (less than 40) implies that the multiple of nine you end up with is also relatively small. In fact, the only possible values you can get when you sum the digits are 0, 9, and 18. The next three steps are designed to turn any of these three values into a 4, leading you to the letter “D”. The last step is the only point in these instructions where something could go wrong. The person following them actually has a choice at this point. There are four countries on Earth whose names begin with “D”: Denmark, Djibouti, Dominica, and the Dominican Republic. Luckily, for most readers of this text, Denmark is more likely to come to mind than any of the other three countries (even though the Dominican Republic is actually larger in both land mass and population).

This example should make it clear that it is possible to follow instructions without understanding how they work. It is equally clear that it is not possible to write instructions like those above without understanding how they work. This contrast provides an important insight into the relationship between a computer, a computer program, and the author of the program. A computer follows the instructions in a program the way you followed the instructions above. It can comprehend and complete each step individually but has no understanding of the overall purpose of the program,
the relationships between the steps, or the ways in which these relationships ensure that the program will accomplish its overall purpose. The author of a program, on the other hand, must understand its overall purpose and ensure that the steps specified will accomplish this purpose.

Instructions like this are important enough to deserve a name. We call a set of instructions designed to accomplish some specific purpose, even when followed by a human or computer, an algorithm.

There are situations where specifying an algorithm that accomplishes some purpose can actually be useful rather than merely amusing. To illustrate this, consider the standard procedure called long division. A sample of the application of the long-division procedure to compute the quotient 13042144 / 32 is shown below:

```
 407567
128
242
242
181
160
192
224
224
0
```

Although you may be rusty at it by now, you were taught the algorithm for long division sometime in elementary school. The person teaching you might have tried to help you understand why the procedure works, but ultimately you were probably simply taught to perform the process by rote. After doing enough practice problems, most people reach a point where they can perform long division but can't even precisely describe the rules they are following, let alone explain why they work. Again, this process was designed so that a human can perform the steps without understanding exactly why they work. Here, the motivation is not to surprise anyone. The value of the division algorithm is that it enables people to perform division without having to devote their mental energies to thinking about why the process works.

Finally, to demonstrate that algorithms don't always have to involve arithmetic, let's consider another example where the motivation for designing the instructions is to provide a pleasant surprise. Well before you learned the long-division algorithm, you were probably occasionally entertained by the process of completing a connect-the-dots drawing like the one shown in Figure 1.1. Go ahead! It's your book. Connect the dots and complete the picture.

A connect-the-dots drawing is basically a set of instructions that enable you to draw a picture without understanding what it is you are actually drawing. Just as it wasn't clear that the arithmetic you were told to perform in our first example would lead you to think of elephants, it is not obvious looking at Figure 1.1 that you are looking at instructions for drawing an elephant. Nevertheless, by following the instructions "Connect the dots" you will do just that (even if you never saw an elephant before).

This example illustrates a truth of which all potential programmers should be aware. It harder to devise an algorithm to accomplish a given goal than it is to simply accomplish the goal.
number the dots so the desired picture can be drawn without ever having to lift your pencil for
the paper can be tricky. If all you really wanted in the first place was a picture of an elephant
it would be easier to draw one yourself. Similarly, if you have a division problem to solve (as
you don’t have a calculator handy) it is easier to do the division yourself than to try to teach it
long-division algorithm to someone who doesn’t already know it, so that he or she can solve it
problem for you.

As you learn to program, you will see this pattern repeated frequently. Learning to conve
your own knowledge of how to perform a task into a set of instructions so precise that they ca
be followed by a computer can be quite challenging. Developing this ability, however, is the ke
to learning how to program. Fortunately, you will find that as you acquire the ability to turn a
informal understanding of how to solve a class of problems into a precise algorithm, you will de
developing mental skills you will find valuable in many other areas.

1.2 The Java Programming Language

An algorithm starts as an idea in one person’s mind. To become effective, it must be communicat
to other people or to a computer. Communicating an algorithm requires the use of a language
A program is in fact an algorithm expressed in a computer comprehend
The choice of the language in which an algorithm is expressed is important. The numeral
calculation puzzle that led you to think of Danish elephants was expressed in English. If on
instructions had been written in Danish, most readers of this text would not understand them.
The use of language in a connect-the-dots puzzle may not be quite as obvious. Note, however
that we could easily replace the numbers shown with numbers expressed using the Roman
numerals I, II, III, IV, . . . . LXXII. Most of you probably understand Roman numerals, so ye
would still be able to complete the puzzle. You would probably have difficulty, however, if ye
switched to something more ancient like the numeric system used by the Babylonians or ye
something more modern like the binary system that most computers use internally, in which it
first few dots would be labeled 1, 10, 11, 100, 101, and 110.

The use of language in the connect-the-dots example is interesting from our point of vie
because the language used is quite simple. Human languages like English and Japanese are ye
complex. It would be very difficult to build a computer that could understand a complete huma
language. Instead, computers are designed to interpret instructions written in simpler language
designed specifically for expressing algorithms intended for computers. Computer languages a
much more expressive than a system for writing numbers like the Roman numerals, but muc
simpler in structure than human languages.

One consequence of the relative simplicity of computer languages is that it is possible t
write a program to translate instructions written in one computer language into another comput
language. These programs are called compilers. The internal circuitry of a computer usually ca
only interpret commands written in a single language. The existence of compilers, however, mak
it possible to write programs for a single machine using many different languages. Suppose th
you have to work with a computer that can understand instructions written in language A but ye
want to write a program for the machine in language B. All you have to do is find (or write
program written in language A that can translate instructions written in language B into equivalent
instructions in language A. This program would be called a compiler for B. You can then wr

97

Section 1.3 Your First Step of Java

your programs in language B and use the compiler for B to translate the programs you write into
language A so the computer can comprehend the instructions.

Each computer language has its own advantages and disadvantages. Some are simpler th
others. This makes it easier to construct compilers that process programs written in these lan
guages. At the same time, a language that is simple can limit the way you can express yourself,
making it more difficult to describe an algorithm. Think again about the process of constructin
the elephant connect-the-dots puzzle. It is easier to draw an elephant if you let yourself use curved
lines than if you restrict yourself to straight lines. To describe an elephant in the language of
connect-the-dots puzzles, however, you have to find a way to use only straight lines. On the other
hand, a language that is too complex can be difficult to learn and use.

In this text, we will teach you how to use a language named Java to write programs. Java p
vides some sophisticated features that support an approach to programming called object-ori
ted programming that we emphasize in our presentation. While it is not a simple language, it
is one of the simpler languages that support object-oriented programming.

Java is a relatively young computer language. It was designed in the early 90s by a group at
Sun Microsystems. Despite its youth, Java is widely used. Compilers for Java are readily available
for almost all computer platforms. We will talk more about Java compilers and how you will use
them, once we have explained enough about Java itself to let you write simple programs.

Our approach to programming includes an emphasis on what is known as event-driven p
ogramming. In this approach, programs are designed to react to events generated by the user or
system. The programs that you are using to type on the user computer use the event-driven approach. You
do something—press the mouse on a button, select an item from a menu, etc.—and the computer
reacts to the "event" generated by that action. In the early days of computing, programs were
started with a collection of data all provided at once and then run to completion. Many textbooks
still teach that approach to programming. In this text we take the more intuitive event-driven ap
rroach to programming. Java is one of the first languages to make this easy to do as a standard par
te of the language.

1.3 Your First Step of Java

The task of learning any new language can be broken down into at least two parts: studying the
language’s rules of grammar and learning its vocabulary. This is true whether the language is a for
eign language, such as French or Japanese, or a computer programming language, such as Java.
In the case of a programming language, the vocabulary you must learn consists primarily of verbs
can be used to command the computer to do things like "show the number 47.2 on the screen" or
"move the image of the game piece to the center of the window." The grammatical structures of
a computer language enable you to form phrases that instruct the computer to perform several
primitive commands in sequence or to choose among several primitive commands.

When learning a new human language, one undertakes the tasks of learning vocabulary and
grammar simultaneously. One must know at least a little vocabulary before one can understand
eamples of grammatical structure. On the other hand, developing an extensive vocabulary with
out any knowledge of the grammar used to combine words would just be silly. The same applies
to learning a programming language.Java by presenting a few sample programs that illustrate fundamentals of the grammatical structure
Chapter 1 What Is Programming Anyway

of Java programs, using only enough vocabulary to enable us to produce some interesting examples.

1.3.1 Simple Responsive Programs

The typical program run on a personal computer reacts to a large collection of actions if
user can perform using the mouse and keyboard. Selecting menu items, typing in file name
pressing buttons, and dragging items across the screen all produce appropriate reactions from
such programs. The details of how a computer responds to a particular user action are determine
by the instructions that make up the program running on the computer. The examples presente
in this section are intended to illustrate how this is done in a Java program.

To start things off simply, we will restrict our attention to programs that react to simple mouse
operations. The programs we consider in this section will only specify how the computer should
respond when the user manipulates the mouse by clicking, dragging, or moving the mouse with
the boundaries of a single window. When one of these programs is run, all that will appear on it
display will be a single, blank window. The programs may draw graphics or display text messages
within this window in response to user actions, but there will be no buttons, menus, scrollbars or
the like.

As a first example, consider the structure of a program which simply draws some text on the
screen when the mouse is clicked. When this program is run, a blank window appears on the
screen. The window remains blank until the user positions the mouse cursor within the window
and presses the mouse button. Once this happens, the program displays the phrase

I'm Touched

in the window as shown in Figure 1.2. As soon as the user releases the mouse, the message
disappears from the window. That is all it does! Not exactly Microsoft® Word®, but it is sufficient
to illustrate the basic structure of many of the programs we will discuss in this text.

Such a Java program is shown in Figure 1.3. A brief examination of the text of the program
features that are certainly consistent with our description of this program’s behavior. Then is the line

new Text( "I'm Touched", 40, 50, canvas );

which specifies the message to be displayed. This line comes shortly after a line containing the
word "on mouse press" (all forced together to form the single word onMousePress) which

I'm Touched

Figure 1.2: Window displayed by a very simple program
Chapter 1 What Is Programming Anyway

This line is called a class header. The programs you write will contain a line that looks just like this except that you will replace the word TouchyWindow with a word of your own choosing; TouchyWindow is just the name we have chosen to give to our program. It is appropriate to give a program a name that reflects its behavior.

This line is called a class header because it informs the computer that the text that follows describes a new class. Why does Java call the specification that describes a program a "class"? Java uses the word class to refer to:

A set, collection, group, or configuration containing members regarded as having certain attributes or traits in common. (From the American Heritage Dictionary)

If several people were to run the program shown above at the same time but on different computer each would have an independent copy of the program described by this class. If one person clicked in the program's window, the message "I'm touched" would only appear on that person's computer. The other computers running the same program would be unaffected. Thus, the runtime copies of the program form a collection of distinct but very similar objects. Java refers to such collection of objects as a class.

Using Software Libraries

The class header of TouchyWindow indicates that it extends something called WindowController. This means that our program depends on previously written Java instructions. Programs are rarely built from scratch. The physical circuits of which a computer is constructed are only capable of performing very simple operations like changing the color of a single cell on the screen. If every program were built from scratch, every program would have to explicitly describe every one of the primitive operations required to accomplish its purpose. Instead, libraries have been written containing collections of instructions describing useful common operations like drawing a line on the screen. Programs can then be constructed using the operations described in the library in addition to the operations that can be performed by the basic hardware.

This notion of using collections of previously written Java instructions to simplify the construction of new programs explains the mysterious phrases found in the first two lines of our program. Lines that start with the word import inform Java which libraries of previously written instructions our program uses. In our example, we list two libraries, java.awt and objectdraw. The library named java.awt is a collection of instructions describing common operations for creating windows and displaying information within windows. The initials "awt" stand for "Abstract Windowing Toolkit". The prefix "java." reveals that this library is a standard component of the Java language environment used by many Java programs.

The second library mentioned in our import specifications is objectdraw. This is a library designed by the authors of this text to make the Java language more appropriate as an environment for teaching programming. Recall that the class header of our example program mentions the TouchyWindow extends WindowController. WindowController refers to a collection of Java instructions that form part of this objectdraw library. A WindowController is an object that coordinates user and program activities within the window associated with a program. If the program were nothing but a WindowController, then all that would happen when it was run would be that a window would appear on the screen. Nothing would ever appear within the window.

Our TouchyWindow class specification extends the functionality of the WindowController by telling it to display a message in the window when the mouse is pressed.

Section 1.3 Your First Sip of Java

Getting Started

The single open brace ("") that appears at the end of the class header for TouchyWindow introduces an important and widely used feature in Java's grammatical structure. Placing a pair of opening and closing brace around a portion of the text of a program is Java's way of letting the programmer draw a big box around that text. Enclosing lines of text in braces indicate that they form a single, logical unit. If you can scan quickly over the complete example, you will see that braces are used in this way in several parts of this program, even though it is quite short.

The open brace after the public class TouchyWindow... line is matched by the closing brace on the last line of the program. This indicates that everything between these two braces (i.e., everything left in the example) should be considered part of the description of the class named TouchyWindow. The text between these braces is called the body of the class.

Exercise 1.3.1

Write the class header for a program called HIMon.

1.3.3 Discourse on the Method

The first few lines in the body of the class TouchyWindow look like:

    public void onMousePress(Location point) {
      new Text( "I'm touched", 40, 50, canvas );
    }

This text is an example of another important grammatical form in Java, the method definition. A method is a named sequence of program instructions. In this case, the method being defined is named onMousePress and within its body (which is bracketed by braces just like the body of the class) it contains the single instruction:

    new Text( "I'm touched", 40, 50, canvas );

In general, the programmer is free to choose any appropriate name for a method. The method name can then be used in other parts of the program to cause the computer to obey the instructions within the method's body. Within a class that extends WindowController, however, certain method names have special significance. In particular, if such a class contains a method which is named onMousePress, then the instructions in that method's body will be followed by the computer when the mouse is depressed within the program's window. That is why this particular program reacts to a mouse press as it does.

The single line that forms the body of our onMousePress method:

    new Text( "I'm touched", 40, 50, canvas );

is an example of one of the primitive commands provided to display text and graphics on a computer's screen. It specifies that the phrase

    I'm touched

should be displayed on the canvas. The portion of the computer's screen controlled by the program, at a position determined by the x and y coordinates (40,50). The components of an instruction like this that tells the computer to display information on the screen are quite important. By changing
Writing a program isn’t enough. You also have to get the program into your computer and convince your computer to follow the instructions it contains.

A computer program like the one shown in the preceding section is just a fragment of text. You already know ways to get other forms of textual information into a computer. You use a word processor to write papers. When entering the body of an email message you use an email application like Eudora® or Outlook®. Just as there are computer applications designed to allow you to enter these forms of text, there are applications designed to enable you to enter the text of a program.

Entering the text of your program is only the first step. As explained earlier, unless you write your program in the language that the machine’s circuits were designed to interpret, you need to use a compiler to translate your instructions into a language the machine can comprehend. Finally, after this translation is complete you still need to somehow tell the computer to treat the file(s) created by the translation process as instructions and to follow them.

Typically, the support needed to accomplish all three of these steps is incorporated into a single application called an integrated development environment or IDE. It is also possible to provide separate applications to support each step. Which approach you use will likely depend on the facilities available to you and the inclination of the instructor teaching you to program. There are too many possibilities for us to attempt to cover them all in this text. To provide you with a sense of what to expect, however, we will sketch how two common integrated development environments, BlueJ and Eclipse, could be used to enter and run the TouchyWindow program. These sketches are not intended to provide you with the detailed knowledge required to actually use either of these IDEs effectively. We will merely outline the main steps that are involved.

The IDEs we will describe share several important properties:

- Implementations of both IDEs are available for a wide range of computer systems including Windows systems, MacOS, and Unix and its variants.
- Both IDEs are available for free and can be downloaded from the Web.

They also differ in major ways. BlueJ was designed by computer science faculty members with the primary goal of providing a Java development system for use when teaching students to program. Eclipse was developed to meet the needs of professional programmers.

Just as it is helpful to divide a book into chapters, it is helpful to divide large programs into separate text files describing individual components of the program. Within a Java IDE, the collection of text files that constitute a program is called a project. In fact, most Java IDEs expect all programs, even programs that are quite small, to be organized as projects. As a result, even though our TouchyWindow program is only ten lines long and will definitely only require one text file, the first step performed to enter and run this program using either Eclipse or BlueJ will be to use an entry in the application’s “File” menu to create a new project.

The IDE will then display a number of dialog boxes asking for information about the project we wish to create. Among other things, we will be asked to specify a name for the project and to select a location in our computer’s file system to store the file(s) that will hold the text of our program.

Once a project has been created, the IDE will display a window representing the state of the project. The window Eclipse would present is shown in Figure 1.5 and the window BlueJ would present is shown in Figure 1.6.
The next step is to tell the IDE that we wish to create a new file and enter the text of our program. With Bluej, we do this by pressing the “New Class…” button seen in the upper left of the window shown in Figure 1.6. With Eclipse, we select a “New Class” menu item. In either case, the IDE will then ask us to enter information about the class, including its name, in a dialog box. Then the IDE will present us with a window in which we can type the text of our program, much as we would type within a word processor. In Figures 1.7 and 1.8 we show how the windows provided by Bluej and Eclipse would look after we ask the IDE to create a new class and enter the text of the class. Eclipse incorporates the text entry window as a subwindow of the project window. Bluej displays a separate text window.

In both the Bluej project window and the Bluej window holding the text of the TouchyWindow class there is a button labeled “Compile”. Pressing this button instructs Bluej that we have completed entering our code and would like to have it translated into a form the machine can more easily interpret. Under most Java IDEs, compiling your code will produce files storing a translation of your instructions into a language called Java virtual machine code or byte code. After this is done, we can ask Java to run our program by depressing the mouse on the icon that represents the TouchyWindow class within the Bluej project window and selecting the “new TouchyWindow” item from the pop-up menu that appears. Bluej will then display a new window controlled by the instructions included in our program. When the mouse is pressed in this window, the words “I’m touched!” will appear, as shown in Figure 1.9.

With Eclipse, compiling your program and running it can be combined into a single step. You first create what Eclipse calls a run configuration. This involves specifying things like the size of the window created when your program starts running. We will not discuss the details of this process here. Once you have created a run configuration, you can compile and run your program by pressing an icon displayed at the top of the Eclipse window that is designed to look like a human runner. Like Bluej, Eclipse then displays a new window in which you can interact with your program.
Section 1.5 Drawing Primitives

1.5.1 The Graphics Coordinate System

Programs that display graphics on a computer screen have to deal extensively with a coordinate system similar to the one you have used when plotting functions in math classes. This is not evident to users of these programs. A user of a program that displays graphics can typically specify the position or size of a graphical object using the mouse to indicate screen positions without ever thinking in terms of x and y coordinates. Writing a program to draw such graphics, however, is very different from using one. When your program runs, someone else controls the mouse. Just imagine how you would describe a position on the screen to another person if you were not allowed to point with your finger. You would have to say something like “Two inches from the left edge of the screen and three inches down from the top of the screen.” Similarly, when writing programs you will specify positions on the screen using pairs of numbers to describe the coordinates of each position.

The coordinate system used for computer graphics is like the Cartesian coordinate system studied in math classes but with one big difference. The y axis in the coordinate system used in computer graphics is upside down. Thus, while your experience in algebra class might lead you to expect the point (2,3) to appear below the point (2,5), on a computer screen just the opposite is true. This difference is illustrated by Figure 1.10, which shows where these two points fall in the

![Diagram](image-url)
normal Cartesian coordinate system and in the coordinate system used to specify positions when drawing on a computer screen.

The graphics that appear on a computer screen are actually composed of tiny squares of color called pixels. For example, if you looked at the text displayed by our TouchyWindow program with a magnifying glass you would discover it is actually made up of little black squares as shown in Figure 1.11. The entire screen is organized as a grid of pixels. The coordinate system used to place graphics in a window is designed to match this grid of pixels in that the basic unit of measurement in the coordinate system is the size of a single pixel. So, the coordinates (30,50) describe the point that is 30 pixels to the right and 50 pixels down from the origin.

Another important aspect of the way in which coordinates are used to specify where graphics should appear is that there is not just a single set of coordinate axes used to describe location anywhere on the computer’s screen. Instead, there is a separate set of axes associated with each window on the screen and, in some cases, even several pairs of axes for a single window.

Rather than complicating the programmer’s job, the presence of so many coordinate systems makes it simpler. Many programs may be running on a computer at once, and each should only produce output in certain portions of the screen. If you are running Microsoft Word at the same time as Adobe Photoshop, you would not expect text from your Word document to appear in or of Photoshop’s windows. To make this as simple as possible, each program’s drawing commands must specify the window or other screen area in which the drawing should take place. Then the coordinates used in these commands are interpreted using a separate coordinate system associated with that area of the screen. The origin of each of these coordinate systems is located in the upper left corner of the area in which the drawing is taking place rather than in the corner of the machine physical display. This makes it possible for a program to produce graphical output without being aware of the location of its window relative to the screen boundaries or the locations of other windows.

In many cases, the area in which a program can draw graphics corresponds to the entire interior of a window on the computer’s display. In other cases, however, the region used by a program may be just a subsection of a window or there may be several independent drawing areas with a given window. Accordingly, we refer to a program’s drawing area as a canvas rather than as a window.

In interpreting your graphic commands, Java will assume that the origin of the coordinate system is located at the upper left corner of the canvas in which you are drawing. The location of the coordinate axes that would be used to interpret the coordinates specified in our TouchyWindow example are shown in Figure 1.12. Notice that the coordinates of the upper left hand corner of the window are (0,0). The window shown is 165 units wide and 100 units high. Thus the coordinate of the lower right corner are (165,100). The text is positioned so that it falls in a rectangle whose upper left corner has an x coordinate of 40 and a y coordinate of 50.

The computer will not consider it an error if you try to draw beyond the boundaries of your program’s canvas. It will remember everything you have drawn and show you just the portion of these drawings that falls within the boundaries of your canvas.

Exercise 1.5.1

Rewrite the line of code in onMousePress of class TouchyWindow so that it now displays the message “Hello” 60 pixels to the right and 80 pixels down from the top left corner of the window.

1.5.2 Constructing Graphic Objects

The line

new Text("I’m Touched", 40, 50, canvas);

used in our example program’s onMousePress method is called a construction. Whenever you want to display a new object on the screen, you will include a construction in your program. The general syntax for such a command includes:

- the word new, which informs the computer that a new object is being constructed;
- the name of the class of object to be constructed; and
- a list of coordinates and other items describing the details of the object. These extra pieces of information are called actual parameters or arguments. The entire list is surrounded by parentheses, and its elements are separated by commas.

After the parameter list, Java expects you to type a semicolon. This semicolon is not part of the construction itself but is a more general aspect of Java’s syntax. Java requires that each simple command we include in a method’s body be terminated by a semicolon, much as we terminate sentences with periods in English. For example, the other command which appears
in the TouchyWindow program:

```java
canvas.clear();
```
also ends with a semicolon. Most phrases that are not themselves commands, such as the header of methods, do not end with semicolons.

The parameters required in a construction will depend on the class of the item being constructed. In our example program, we construct a Text object. Text is the name for a piece of text displayed on the screen. The first parameter expected in a Text constructor is the text to be displayed. In this example, the text we want displayed is:

**I'm Touched**

We surround this text with a pair of double quote marks to tell Java that we want exactly this text to appear on the screen. The next two values specify that the text be indented 40 pixels from the left edge of the drawing area and that the text be placed immediately below an imaginary line 5 pixels from the top of the drawing area.

The last item in the list of parameters to the Text construction, the canvas, tells the computer in which area of the screen the new message should be placed. In your early programs, there will only be one area in which your program can draw, and the name canvas will refer to this region. As a result, including this bit of information will seem unnecessary (if not tedious). Eventually, however, you will want to construct programs that display information in multiple windows. I provide the flexibility to construct such programs, the primitives for displaying graphics require you to include the canvas specification even when it seems redundant.

Several other types of graphical objects can be displayed using similar constructions. For example, to display a line between the corners of a canvas whose dimensions are 200 by 300, you would write:

```java
new Line(0, 0, 200, 300, canvas);
```

The line produced would look like the line shown in the window in Figure 1.13. In this construction, the first pair of numbers, 0, 0, specifies the coordinates of the starting point of the line (the upper left corner of your window) and the pair 200, 300 specifies the coordinates of the line's end point (the lower right corner).

Similarly, to draw a line from the middle of the window, which has the coordinates (100,150) to the upper right corner, whose coordinates are (200,0), you would say:

```java
new Line(100, 150, 200, 0, canvas);
```

Such a line is shown in Figure 1.14.

Using combinations of these construction statements, we could replace the single instruction in the body of the onMousePress method shown above with one or more other instructions. Such a modified program is shown in Figure 1.15.

The only differences between this example and TouchyWindow are the name given to the classes (CrossedLines vs. TouchyWindow) and the commands included in the body of the onMousePress method. The modified program's version of onMousePress includes two commands in its body which instruct the computer to draw two intersecting, perpendicular lines. The drawing produced is also shown in the figure.
import objectdraw.*;
import java.awt.*;

public class CrossedLines extends WindowController {
    public void onMousePress(Location point) {
        new Line(40, 40, 60, 60, canvas);
        new Line(60, 40, 40, 60, canvas);
    }

    public void onMouseRelease(Location point) {
        canvas.clear();
    }
}

Figure 1.16 A program that draws two crossed lines.

There are several other forms of graphics you can display on the screen. The command:

    new FramedRect(20, 50, 80, 40, canvas);

will display the outline, or frame, of an 80-by-40 rectangular box in your canvas. The pair 20
50 specifies the coordinates of the box’s upper left corner. The pair 80, 40 specifies the width
and height of the box. If you replace the name FramedRect by FilledRect to produce its
construction

    new FilledRect(20, 50, 80, 40, canvas);

the result will instead be an 80-by-40 solid black rectangular box.

The command:

    new FilledOval(20, 50, 80, 40, canvas);

will draw an oval on the screen. The parameters are interpreted just like those to the FilledRect
construction. Instead of drawing a rectangle, however, FilledOval draws the largest ellipse
that can fit within the rectangle described by its parameters. To illustrate this, Figure 1.1

shows what the screen would contain after executing the two constructions:

    new FramedRect(20, 50, 80, 40, canvas);
    new FilledOval(20, 50, 40, canvas);

The upper left corner of the rectangle shown is at the point with coordinates (20, 50). Both shapes
are 80 pixels wide and 40 pixels high.

Other primitives allow you to draw additional shapes and to display image files in your
canvas. A full listing and description of the available graphic object types and the forms of
the commands used to construct them can be found in Appendix B. For now, the graphical object
types Text, Line, FramedRect, FilledRect, and FilledOval will provide enough flexibility for our purposes.

Exercise 1.5.2
Sketch the picture that would be produced if the following constructions were executed. You should
assume that the canvas associated with the program containing these instructions is 200 pixels
wide and 200 pixels high.

    new Line(0, 100, 100, 0, canvas);
    new Line(100, 0, 200, 100, canvas);
    new Line(200, 100, 100, 200, canvas);
    new Line(100, 200, 0, 100, canvas);
    new FramedRect(50, 50, 100, 100, canvas);

Exercise 1.5.3
Write a sequence of Line and/or FramedRect constructions that would produce each of the
drawings shown below. In both examples, assume that the drawing will appear in a 200-by-
200-pixel window. For the drawing of the three-dimensional cube, there should be a space
5 pixels wide between the cube and the edges of the window in those areas where the cube
comes closest to the edges. The rectangle drawn for the front face of the cube should be 155
pixels wide and 155 pixels high. The two visible edges of the rear of the cube should also be
155 pixels long.
1.6 Additional Event-Handling Methods

In our examples thus far, we have used the two method names `onMousePress` and `onMouseRelease` to establish a correspondence between certain user actions and instructions the computer should follow when these actions occur. In this section, we introduce several other method names that can be used to associate instructions with other user actions.

1.6.1 Mouse-Event-Handling Methods

In addition to `onMousePress` and `onMouseRelease`, there are five other method names that have special significance for handling mouse events. If you include definitions for any of these methods within a class that extends `WindowController`, then the instructions within the methods will be executed when the associated events occur. The definitions of all these methods have the same form. You have seen that the header for the `onMousePress` method looks like:

```
public void onMousePress( Location point )
```

The headers for the other methods are identical except that `onMousePress` is replaced by the appropriate method name.

All of the mouse-event-handling methods are described below:

- `onMousePress` specifies the actions the computer should perform when the mouse button is first pressed down.
- `onMouseRelease` specifies the actions the computer should perform when the mouse button is released.
- `onMouseClick` specifies the actions the computer should perform whenever the mouse is pressed and then quickly released without significant mouse movement between the two events. The action specified in this method will be performed in addition to (and after) any instructions in `onMousePress` and `onMouseRelease`.

### Exercise 1.6.1

Write the method header for the `onMouseMove` method.

### Exercise 1.6.2

Write a method that draws a filled square on the canvas when the mouse enters the canvas. The square should be 100 by 100 pixels with the upper left corner at the origin.

### Exercise 1.6.3

Write a complete program that will display "I’m inside" when the mouse is inside the program’s window and "I’m outside" when the mouse is outside the window. The screen should be blank when the program first begins to execute and should stay blank until the mouse is moved in or out of the window.

1.6.2 The `begin` Method

In addition to the seven mouse-event-handling methods, there is one other event-handling method that is independent of the mouse. This is the method named `begin`. If a `begin` method is defined in a program, then it is executed once each time the program begins to execute. The form of the definition of the `begin` method is slightly different from that of the mouse-event-handling methods. Since there is no point on the screen associated with this event, `Location` point is omitted from the method’s header. The parentheses that would have appeared around the word `Location` point are not required. Thus, a `begin` method’s definition will look like:

```
public void begin() {
    ...
}
```

The `begin` method provides a way to specify instructions the computer should follow to set things up before the user begins interacting with the program. As a simple example of this, consider how we can modify our `TouchingWindow` program to improve its rather limited user interface. When the current version of the program runs, it merely displays a blank window. Now that you are familiar with the program, you know that it expects the user to click on the window. The program, however, could make this more obvious by displaying a message asking the user to click on the window when it first starts. This can be done by including a command to construct an appropriate `Text` object in a `begin` method.
import objectdraw.*;
import java.awt.*;

public class TouchyWindow extends WindowController {

    public void begin() {
        new Text("Click in this window.", 20, 20, canvas);
    }

    public void onMousePress(Location point) {
        canvas.clear();
        new Text("I'm Touched", 40, 50, canvas);
    }

    public void onMouseRelease(Location point) {
        canvas.clear();
    }

    Figure 1.17 A simple program with interactions

    The code for this improved version of TouchyWindow is shown in Figure 1.17. In addition to adding the begin method, we have added the invocation canvas.clear(); to onMousePress so that the instructions will be removed as soon as the user follows them.

1.7 To Err is Human

We all make mistakes. Worse yet, we often make the same mistakes over and over again.

If we make a mistake while giving instructions to another person, he or she can frequently figure out what we really meant. For example, if you give someone driving directions and say "turn left" where you should have said "turn right," chances are that they will realize after driving in the wrong direction for a while that they are not seeing any of the landmarks the remaining instructions mention, go back to the last turn before things stopped making sense and try to turn in the other direction. Our ability to deal with such incorrect instructions, however, depends on our ability to understand the intent of the instructions we follow. Computers, unfortunately, nev er really understand the instructions they are given, so they are far less capable of dealing with errors.

There are several distinct types of errors you can make while writing or entering a program. The computer will respond differently, depending on the type of mistake you make. The first type of mistake is called a syntax error. The defining feature of a syntax error is that the IDE can detect that there is a problem before you try to run your program. As we have explained, a computer program must be written in a specially designed language that the computer can interpret or, least translate into a language which it can interpret. Computer languages have rules of grammar just like human languages. If you violate these rules, either because you misunderstand the

or simply because you make a typing mistake, the computer will recognize that something is wrong and tell you that it needs to be corrected.

The mechanisms used to inform the programmer of syntactic errors vary from one IDE to another. Eclipse constantly examines the text you have entered and indicates fragments of your program that it has identified as errors by underlining them and/or displaying an error icon on the offending line at the left margin. If you point the mouse at the underlined text or the error icon, Eclipse will display a message explaining the nature of the problem. For example, if we accidentally left out the closing ")" after the body of the onMousePress method while entering the program shown in Figure 1.17, Eclipse would underline the semicolon at the end of the last line of the method. Pointing the mouse at the underlined semicolon would cause Eclipse to display the message "Syntax error, insert '}' to complete MethodBody" as shown in Figure 1.18.

The bad news is that your IDE will not always provide you with a message that pinpoints your mistake so clearly. When you make a mistake, your IDE is forced to try to guess what you meant to type. As we have emphasized earlier, your computer really cannot be expected to understand what your program is intended to do or say. Given its limited understanding, it will often mistake your intention and display error messages that reveal its confusion. For example, if you type

canvas.clear;

instead of

canvas.clear();
As a simple example of a logical error, suppose that while typing the onMouseRelease method for the TouchWindow program you got confused and typed onMouseExit instead of onMouseRelease. The result would still be a perfectly legitimate program. It just wouldn’t be the program you meant to write. Your IDE would not identify your mistake as a syntax error. Instead, when you ran the program, it just would not do what you expected. When you released the mouse, the “I’m Touched” message would not disappear as expected.

This may appear to be an unlikely error, but there is a very common error which is quite similar to it. Suppose that, instead of typing the name onMouseRelease, you typed the name onMouseRelease. Look carefully. These names are not the same. onMouseRelease is not the name of one of the special event-handling methods discussed in the preceding sections.

In more advanced programs, however, we will learn that it is sometimes useful to define additional methods that do things other than handle events. The programmer is free to choose names for such methods. onMouseRelease would be a legitimate (if strange) name for such a method. That is, a program containing a method with this name has to be treated as a syntactically valid program by any Java IDE. As a result, your IDE would not recognize this as a typing mistake, but when you ran the program, Java would think you had decided not to associate any instructions with mouse-release events. As before, the text “I’m Touched” would never be removed from the canvas.

There are many other examples of logical errors a programmer can make. Even in a simple program like TouchyWindow, mistyping screen coordinates can lead to surprises. If you mistyped an x coordinate, as in

```java
new Text("I’m Touched", 400, 50, canvas);
```

the text would be positioned outside the visible region of the program window. It would seem as if it had never appeared. If the line

```java
canvas.clear();
```

had been placed in the onMousePress method after the line to construct the message, the message would disappear so quickly that it would never be seen.

Of course, in larger programs the possibilities for making such errors just increases. You will find that careful, patient, thoughtful examination of your code as you write it and after errors are detected is essential.

Programming a computer to say “I’m Touched” is obviously a rather modest achievement. In the process of discussing this simple program, however, we have explored many of the principles and practices that will be developed throughout this book. We have learned that computer programs are just collections of instructions. These instructions, however, are special in that they can be followed mechanically, without understanding their actual purpose. This is the notion of an algorithm, a set of instructions that can be followed to accomplish a task without understanding the task itself. We have seen that programs are produced by devising algorithms and then expressing
them in a language that a computer can interpret. We have learned the rudiments of it
language we will explore further throughout this text, Java. We have also explored some of
the tools used to enter computer programs, translate them into a form the machine can follow
and run them.

Despite our best efforts to explain how these tools interact, nothing can take the place of actual
writing, entering, and running a program. We strongly urge you to do so before proceeding to
read the next chapter. Throughout the process of learning to program you will discover that it
is a skill that is best learned by practice. Now is a good time to start.

### 1.9 Chapter Review Problems

#### Exercise 1.9.1

Here is a sample class header:

```java
public class Hangman extends WindowController
```

Explain what is meant by the following words from the above line:

a. class
b. Hangman
c. extends
d. WindowController

#### Exercise 1.9.2

Consider the point located at coordinates (100,100). What are the coordinates of the following
points in the computer graphics coordinate system?

a. 40 pixels down and 30 pixels to the left of (100,100)
b. 60 pixels up and 10 pixels to the right of (100,100)
c. 35 pixels up and 45 pixels to the left of (100,100)
d. 20 pixels down and 50 pixels to the left of (100,100)
e. 80 pixels down and 15 pixels to the right of (100,100)

#### Exercise 1.9.3

Sketch the picture that the following lines of code would produce. Assume the window is 200
pixels wide and 200 pixels high.

```java
new FrameRect(20, 20, 160, 160, canvas);
new Line(20, 180, 100, 20, canvas);
new Line(100, 20, 100, 180, canvas);
new FilledOval(100, 100, 80, 80, canvas);
new Line(180, 100, 100, 100, canvas);
```
CHAPTER 2

What's in a Name?

An important feature of a programming language is that the vocabulary used can be expanded by the programmer. Suppose you want to draw a line that ends at the current position of the mouse. The actual location of this point will not be determined until the program you write to being used. To talk about this position in your program you must introduce a name that will function as a placeholder for the information describing the mouse’s position. Such names are somewhat like proper names used to refer to the characters in a story. You cannot determine their meanings by simply looking them up in a standard dictionary. Instead, the information that enables you to interpret them is part of the story itself. In this chapter, we will continue our introduction to programming in Java by discussing how to introduce and use such names in Java programs. In addition, we will introduce additional details of the primitives used to display graphics.

2.1 Naming and Modifying Objects

Constructions like:

```java
no Line( 200, 200, 300, 300, canvas );
```

provide the means to place a variety of graphic images on a computer screen. Most program that display graphics, however, do more than just place graphics on the screen. Instead, as the run, they modify the appearance of the graphics they have displayed in a variety of ways. Items are moved about the screen, buttons change color when the mouse cursor is pointed at them, text is highlighted, and often items are simply removed from the display. To learn how to produce such behavior in a Java program, we must learn about operations that change the properties of objects after they have been constructed. These operations are called mutator methods, based on the etymology meaning of the word “mutate”, to change or alter.

2.1.1 Mutator Methods

Just as each class of graphical objects has a specific name which must be used in a construction, each mutator method has a specific name. The names are chosen to suggest the change associated with the method, but there are some subtleties. For example, there are two mutator methods that can be used to move a graphical object to a new position on the screen. They are named move and moveTo. The first tells an object to move a certain distance from its current position. The second is used to move an object to a position described by a pair of coordinates, regardless of its previous position.

With most mutator methods, including move and moveTo, the programmer must specify additional pieces of information that determine the details of the operation applied. For example, when you tell Java to move an object, you need to tell it how far. The syntax used to provide such information is similar to that used to provide extra information in a construction. A comma-separated list of values is placed in parentheses after the method name. Thus, to move an object 30 pixels to the right and 15 pixels down the screen one would say:

```java
move( 30, 15 )
```

While the use of a mutator method shares portions of the syntax of a construction, there are major syntactic and conceptual differences between the two. A construction produces a new object. Hence, each construction begins with the word new. A mutator method is used to modify an already existing object. Accordingly, the word new is eliminated and must be replaced with something that indicates which existing object should be modified. To make this clear, let us consider a simple example.

Many programs start by displaying an entertaining animation. With our limited knowledge of Java, we can’t yet manage an entertaining animation, but we can, with a bit of help from the program’s user, create a very simple animation. In particular, we can write a program that displays a circle near the bottom of the canvas and then moves the circle up a bit each time the mouse is clicked. With a bit of imagination, you can think of the circle as the sun rising at the dawn of a new day.

Without knowing anything about mutator methods, you should be able to imagine the rough outline of such a program. Basically, from the description it is clear that the program needs to construct a `Filledoval` in its `begin` method. It is also clear that the program will need to define an `onMouseClick` method that uses the `move` mutator method. You don’t know enough to write this method yet. The fact that `onMouseClick` will be used, however, should tell you a bit more about `begin`. If the user of our program needs to click the mouse to get it to function, then, just as we did in our improved version of `TouchyWindow`, we should include code in `begin` to display instructions telling the user to do this. So, your first draft of a `begin` method might look something like:

```java
public void begin() {
    no Filledoval( 30, 150, 100, 100, canvas );
    no Text( "Please click the mouse repeatedly", 20, 20, canvas );
}
```
Chapter 2  What's in a Name

In general, to apply a method to a particular object, Java expects us to provide a name or some other means of identifying the object, followed by a period and the name of the method to be used. So, in order to move the oval created in our begin method, we have to tell Java to associate a name with the oval.

### 2.1.2 Instance Variable Declarations

First, we have to choose a name to use. Java puts a few restrictions on the names we can pick. Names that satisfy these restrictions are called identifiers. An identifier must start with a letter. After the first letter, we can use either letters, digits, or underscores. So, we could name our oval something like sunSpot, oval2move, or ra. Case is significant. An identifier can be as long (or short) as you like, but it must be just one word (i.e., no blanks or punctuation marks are allowed in the middle of an identifier). A common convention used to make up for the inability to separate parts of a name using spaces is to start each part of a name with a capital letter. For example, we might use a name like ovalToMove. It is also a convention to use identifiers starting with lowercase letters to name variables to help distinguish them from the names of classes and constants.

We can use a sequence of letters, numbers, and underscores as an identifier in a Java program even if it has no meaning in English. Java would be perfectly happy if we named our bvs x2d, xw0. It is much better, however, to choose a name that suggests the role of an object. Such names make it much easier for you and others reading your code to understand its meaning. We suggested earlier that you could think of the display produced by the program we are trying to write as an animation of the sun rising. In this case, sun would be an excellent name for the oval. We will use this name to complete this example.

There are two steps involved in associating a name with an object. Java requires that we first introduce each name we plan to use by including what is called a declaration of the name. Then, we associate a particular meaning with the name using an assignment statement. We discuss declarations in this section and introduce assignments in the following section.

The syntax of a declaration is very simple. For each name you plan to use, you enter the word `private` followed by the name of the type of object to which the name will refer and finally the name you wish to introduce. In addition, like commands, each declaration is terminated by a semicolon. So, to declare the name sun, which we intend to use to refer to a FilledOval, we would type the declaration:

```java
private FilledOval sun;
```

With the declaration of sun added, the contents of the program file for our animation of the sunrise might begin with the code shown in Figure 2.2.

The form and placement of a declaration within a program determines where in the program the name can be used. This region is called the scope of the name. In particular, we will want to refer to the name sun in both the begin and onMouseClick methods of the program we are designing. The declaration of names that will be used in several methods should be placed within the braces that surround the body of our class, but outside any of the method bodies. Names declared in this way are called instance variables. We recommend that instance variable declarations be placed before all the method declarations. The inclusion of the word `private` in an instance variable declaration indicates that only code within the class we are defining should be allowed to refer to this name. The public qualifier that we included in method declarations, by way of contrast, indicates that the method is accessible outside of the class.
Chapter 2: What's in a Name

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

public class RisingSun extends WindowController {
    private FilledOval sun;

    public void begin() {
        ...
    }

    Figure 2.2 Declaring sun in the RisingSun program

    The declaration of an instance variable does not determine to which object the name will refer.
    Instead, it merely informs Java that the name will be used at some point in the program and tell
    Java the type of object that will eventually be associated with the name. The purpose of such
    declaration is to enable Java to give you helpful feedback if you make a mistake. Suppose the
    after deciding to use the name “sun” in our program we made a typing mistake and typed “sin”:
    one line where we meant to type “sun”. It would be nice if when Java tried to run this program
    could notice such a mistake and provide advice on how to fix the error similar to that provided
    a spelling checker. To do this, however, Java needs the equivalent of a dictionary against which
    it can check the names used in the program. The declarations a programmer must include for it
    names used provide this dictionary. If Java encounters a name that was not declared, it reports
    the equivalent of a spelling mistake.

    2.1.3 Assigning Meanings to Variable Names

    Before a name can be used in a command like:
    ```java
    sun.move(0, -5);
    ```
    we must associate the name with a particular object using a command Java calls an assigner
    statement. An assignment statement consists of a name followed by an equals sign and a phrase
    that describes the object we would like to associate with that name. As an example, the assignmernt
    statement needed to associate a name with the oval that represents the sun in our program is:
    ```java
    sun = new FilledOval(50, 150, 100, 100, canvas);
    ```

    In this assignment statement, we use the construction that creates the oval as a subphrase
    describe the object we want associated with the name sun. When we use a construction as an
    independent command, as we have in all our earlier examples, the only effect of executing the
    command is to create the specified object. When a construction is used as a subphrase of a
    assignment, on the other hand, execution of the command both creates the object and associates
    a name with it.

    Ordering is critical in an assignment. The name being defined must be placed on the left side
    the equal sign while the phrase that describes the object to which the name refers belongs on the
    right side. This may be a bit nonintuitive, since Java must obviously first perform the construct
    described on the right before it can associate the new object with the name on the left, and we as
    used in processing information left to right. Java, however, will reject the command as nonsense
    if we interchage the order of the name and the construction.

    Java will also reject an assignment statement that attempts to associate a name with an object
    that is not of the type included in the name’s declaration. The declaration included in Figure 2.2
    states that sun will be used to refer to a FilledOval. If we included the assignment
    ```java
    sun = new FilledRect(50, 150, 100, 100, canvas);
    ```
    in our program, it would be identified as an error, because it attempts to associate the name with
    an object that is a FilledRect rather than with a FilledOval.

    Given this introduction to associating names with objects, we can now show the complete code
    for a “rising sun” program. It appears in Figure 2.3. It includes examples of all three of the basic
    constructs involved in using names: declarations, assignments, and references.

    - The declaration:
      ```java
      private FilledOval sun;
      ```
      appears at the beginning of the class body.

    - An assignment of a meaning to a name appears in the begin method:
      ```java
      sun = new FilledOval(50, 150, 100, 100, canvas);
      ```

    - A reference to an object through a name appears in onMouseClick:
      ```java
      sun.move(0, -5);
      ```

    2.1.4 Comments

    In the complete version of the program, we introduce one additional and very important feature
    of Java, the comment. As programs become complex, it can be difficult to understand their operation
    by just reading the Java code. It is often useful to annotate this code with English text that explains
    more about its purpose and organization. In Java, you can include such comments in the program
    text itself as long as you follow conventions designed to enable the computer to distinguish the
    actual instructions it is to follow from the comments. This is done by preceding such comments
    with a pair of slashes (“//”). Any text that appears on a line after a pair of slashes is treated as a
    comment by Java.

    The program we are writing seems a good example in which to introduce comments. Although
    the program is short and simple, it is not clear that someone reading the code would have the
    imagination to realize that the black circle created by the program was actually intended to
    reproduce the beauty of a sunrise. The Java language isn’t rich enough to allow one to express
    non-technical ideas in code, but we can include them in comments. We include some general
    guidance on using comments to make your programs easier to read and understand in Appendix A.

    The class declaration in Figure 2.3 is preceded by three lines of comments. If we have multiple
    lines of comments, we can write them a bit more simply by starting the comments with a “//”
    and ending them with “//” as follows:

    ```java
    // A program that produces an animation of the sun rising.
    The animation is driven by clicking the mouse button.
    The faster the mouse is clicked, the faster the sun will rise. */
    ```
import objectdraw.*;
import java.awt.*;

// A program that produces an animation of the sun rising.
// The animation is driven by clicking the mouse button.
// The faster the mouse is clicked, the faster the sun will rise.
public class RisingSun extends WindowController {

    private Filledoval sun; // Circle that represents the sun
    private Text instructions;

    // Place the sun and some brief instructions on the screen
    public void begin() {
        sun = new Filledoval(50, 150, 100, 100, canvas);
        instructions = new Text(“Please click the mouse repeatedly”, 20, 20, canvas);

        // Move the sun up a bit each time the mouse is clicked
        public void onMouseClick(Location point) {
            sun.move(-5, 0);
        }
    }

    public void onMouseExit(Location point) {
        sun.move(50, 150);
    }
}

Figure 8.3: Code for rising sun example

Many programmers prefer to format multiline comments as follows:

```java
// A program that produces an animation of the sun rising.
// The animation is driven by clicking the mouse button.
// The faster the mouse is clicked, the faster the sun will rise.
```

While Java only considers the initial “/**” and final “*/”, the “*/”s at the beginning of new lines make it easier for the reader to see that it is part of a comment.

2.1.6 Additional Mouse-Button Methods

There are several other operations that can be applied to graphical objects, once we have the ability to associate names with the objects. For example, as we mentioned earlier, there is a mouse button method named moveTo which moves an object by a specific amount on the screen. There are also two methods named hide and show which can be used to temporarily remove a graphical item from the screen. We can use these three methods to extend the behavior of our RisingSun program.

First, the version of the program shown above becomes totally uninteresting after the mouse has been clicked enough to push the filled oval off the top of the canvas. Once this happens additional mouse clicks have no visible effect. It would be nice if there were a way to tell the program to restart by placing the sun back at the bottom of the canvas. Second, as soon as the user starts to click the mouse, the instructions asking the user to click become superfluous. Worse yet, at some point, the rising sun will bump into the instructions. It would be nice to remove them from the display temporarily and then restore them when the program is reset.

Section 2.1 Naming and Modifying Objects

All we need to do to permit the resetting of the sun is to add the following definition of the onMouseExit method to our RisingSun class.

```java
public void onMouseExit(Location point) {
    sun.moveTo(50, 150);
}
```

With this addition, the user can reset the program by simply moving the mouse out of the program’s canvas. When this happens, the body of the onMouseExit method will tell Java to move the sun oval back to its initial position.

Making the instructions disappear and then reappear is a bit more work. In order to apply mutually exclusive instructions to the instructions, we will have to tell Java to associate a name with the Text created to display the instructions. As we did to define the name sun, we will have to both declare the name we wish to use and then incorporate the creation of the Text into an assignment statement. As an obvious name for this object is “instructions”. If this is our choice, then we need to tell Java that we plan to use this name by adding a declaration of the form:

```java
private Text instructions;
```

to the beginning of our class. We also need to add an assignment of the form:

```java
instructions = new Text(“Please click the mouse repeatedly”, 20, 20, canvas);
```

to the begin method to actually associate the name with the text of the instructions.

It is worth noting that it is sometimes helpful to split a long command into several lines, as we have done in presenting this assignment statement. It can make your code much easier to read and understand. Using multiple lines for one command like this is perfectly acceptable in Java. It is semantically indiscernible rather than the end of a line that tells Java where a command ends. You can split an instruction between two lines at any point where you could type a space except within quoted text. You will also find that the use of indentation and blank lines can make groups of related commands stand out to the reader. These issues are discussed further in Appendix A.

The pair of mutator methods named hide and show provide the means to temporarily remove a bit of graphics from the display. To make the text disappear when the mouse is clicked, we include an instruction of the form:

```java
instructions.hide();
```

in the onMouseClick method. Each time the mouse is clicked, the instructions will be told to hide. Of course, once they are hidden, telling them to hide again has no effect. Note that even though hide expects no parameters, Java still expects us to include the parentheses that would surround the parameters.

When the program is reset, the instructions should reappear. To do this, we would include an instruction of the form:

```java
instructions.show();
```

in the onMouseExit method. The complete text of this revised program is shown in Figure 2.4.

The hide method should only be used when an object is being removed from the canvas temporarily. If an object is being removed permanently—that is, you know that your program will never use the show method to make the object reappear—another method
*import* `objectdraw.*`;  
*import* `java.awt.*`;  

// A program that produces an animation of the sun rising.  
// The animation is driven by clicking the mouse button.  
// The faster the mouse is clicked, the faster the sun will rise.  
public class RisingSun extends WindowController {  
  private FilledOval sun;  // Circle that represents the sun  
  private Text instructions;  // Display of instructions  
  
  // Place the sun and some brief instructions on the screen  
  public void begin() {  
    sun = new FilledOval(50, 150, 100, 100, canvas);  
    instructions = new Text("Please click the mouse repeatedly",  
                           20, 20, canvas);  
  }  
  
  // Move the sun up a bit each time the mouse is clicked  
  public void onMouseClicked(Location point) {  
    sun.move(0, -5);  
    instructions.hide();  
  }  
  
  // Move the sun back to its starting position and redisplay  
  // the instructions  
  public void onMouseExit(Location point) {  
    sun.moveTo(50, 150);  
    instructions.show();  
  }  

Figure 2.4 Rising sun program with neat frame  

named `removeFromCanvas` is more appropriate. The `removeFromCanvas` method irreversibly removes a graphical object from the display. There is no way to put an object that has been removed in this way back on the display. When `hide` is used, the system must save information needed to display the object in case the `show` method is invoked. This consumes space in the computer memory. Using `removeFromCanvas` instead allows the system to totally remove information about the object from the computer’s memory.

2.1.6 Exercises  
In the exercises below, we ask you to consider how to write a program displaying a diamond that appeared to grow a bit each time the mouse was clicked. The following constructions will produce a drawing of the initial diamond shape desired if executed in a program whose canvas is 200 by 200 pixels.

```
new Line(0, 100, 100, 0, canvas);
new Line(100, 0, 200, 100, canvas);
new Line(200, 100, 100, 0, canvas);
new Line(100, 200, 0, 100, canvas);
```

The complete program we have in mind won’t actually make the diamond grow. Instead, all it will do is move each of the four lines drawn by the constructions a bit closer to the corner closest to the line each time the mouse is clicked. For example, each time the mouse is clicked, the line in the upper left corner of the window will be moved one pixel to the left and one pixel up, so that it ends up closer to the upper left corner of the window. The line that starts in the upper right quarter of the window, on the other hand, will be moved one pixel to the right and one pixel up with each click of the mouse. If this is done, after a number of clicks, the display will look like the picture shown below. The four lines won’t be any longer than they were at the start, but they will appear to be part of a bigger diamond than was originally drawn, a diamond that is too big to fit in the window.
2.1.1 Exercise

Before we can move one of these lines or any other graphical object, we must first associate name with the object. A list of possible names that might be used to refer to the line of the diamond shape drawn by the construction:

```
new Line( 200, 100, 100, 200, canvas );
```

is shown below. For each name, indicate whether it would be:

- **invalid** according to Java's rules for forming names,
- **inappropriate** as a name for this particular line, because it would not help a person working with the program remember the purpose of the name or if it does not conform to Java's naming conventions, or
- **appropriate** as a name for this particular line.

In each case, briefly explain your answer.

```
3rdline  thirdline  line3
leftlower s.e. lowerright lowerleft south-east
```

2.1.2 Exercise

Suppose that you selected the instance variable names leftToTop, topToRight, rightToBottom, and bottomToLeft to describe the four lines that form the diamond. Show the declarations required to introduce these names in a Java program.

2.1.3 Exercise

Assuming that the names leftToTop, topToRight, rightToBottom, and bottomToLeft had been chosen to describe the four lines that form the diamond and that they had been declared appropriately, show how the constructions shown above would be turned into assignment statements that both created the Lines and associated the names listed with them. In what methods should these commands be placed if you want the lines to appear in their initial position as soon as the program is started?

2.1.4 Exercise

Assuming that the names leftToTop, topToRight, rightToBottom, and bottomToLeft had been declared appropriately and associated with the lines of the diamond using assignment statements, show the statements required to invoke the move method on each line to move it line closer to the nearest corner of the program's window. Each line should be moved one pixel horizontally and one pixel vertically. In which method should these commands be placed if you want the lines to move each time the mouse is clicked?
Assuming that the declaration of initialPosition shown above is added to the RisingSun class and that the assignment:

```
initialPosition = new Location(50, 150);
```

is included in the begin method, then the construction for the sun could be replaced by:

```
sun = new Filled Oval(initialPosition, 100, 100, canvas);
```

Similarly, the instruction used to reposition the sun in the onMouseExit method:

```
sun.moveTo(50, 150);
```

could be revised to read:

```
sun.moveTo(initialPosition);
```

These changes would not alter the behavior of the program, but they would make it easier to read. A human looking at your program is more likely to understand the purpose of the move instruction using initialPosition than the one that uses “50, 150”. In addition, this approach makes the program easier to change. If you wanted to run the program using a larger screen area, the coordinates for the starting position would need to be changed. In the version of the program written using the name initialPosition, only one line would have to be altered to make this change.

Like the graphical objects we introduced earlier, Location objects can be altered using mutator methods. There is a mutator method for Locations named translate that is very similar to the move method associated with graphical objects. Both translate and move expect two numbers specifying how far to travel in the x and y dimensions of the graphical coordinate system. The difference is that when you tell a graphical object to move, you cause it move on the screen. When you tell a Location to translate, nothing on the screen changes. A Location object describes a position on the screen, but it does not appear on the screen itself. Accordingly, translating Location changes the position described by the Location, but it does not change anything already on the screen. The effect of the translate only becomes apparent if the Location later used to position some graphical object.

To clarify how translate works, consider the sample program shown in Figure 2.5. Each time the mouse is clicked, this program will draw a pair of thick, perpendicular lines. The first lines drawn will intersect at the upper left corner of the window. With each click, the lines drawn will be placed to the right and below the preceding lines, so that the window is eventually filled with a grid pattern.

The program includes two Location variables named verticalCorner and horizontalCorner. If you examine the declaration of these variables, you will see that it allows us to define several variables of the same type in a single declaration by listing all of the names to be declared separated by commas. It is appropriate to use this form of declaration only when the names declared together have related functions. In this case they do. Each of these name describes the Location at which one of a pair of FilledRects will be drawn when the user clicks the mouse.

In its begin method, this program creates two Location objects that both describe the point at the origin of the coordinate system, the upper left corner of the canvas. One of these objects associated with the name verticalCorner and the other with the name horizontalCorner.
This process will be repeated each time the mouse is clicked. After about 15 additional clicks, the window will be nearly filled with a grid of lines, as shown in Figure 2.8. Just a few more clicks will complete the process of filling the screen.

2.3 Layering on the Canvas

Now that we can set the color of a graphical object, we could actually make a much prettier picture for our rising sun program. We could add a blue sky, some white clouds and even a bit of green grass. Better yet, given that the figures in this text are not printed in color, instead of the sun we can draw the moon with a light gray cloud passing in front of it and a dark gray sky behind it. The picture we have in mind is shown in Figure 2.9.

To construct this picture we must declare variables to refer to the sky, moon, and cloud:

```java
private FilledOval moon, cloud;
private FilledRect sky;
```

Then we must construct the desired objects:

```java
sky = new FilledRect( 0, 0, 300, 300, canvas );
moon = new FilledOval( 50, 50, 100, 100, canvas );
cloud = new FilledOval( 70, 110, 160, 40, canvas );
```

and finally set their colors appropriately:

```java
sky.setColor( new Color( 60, 60, 60 ) );
moon.setColor( Color.WHITE );
cloud.setColor( Color.GRAY );
```
The parameters "60, 60, 60" provided to the Color constructor used to set the color of sky describe a dark shade of gray.

Suppose we changed this code by reordering the assignment statements in which the object are created so that the moon is created after the cloud, as in:

```java
sky = new FilledRect(0, 0, 100, 100, canvas);
cloud = new FilledOval(70, 110, 160, 40, canvas);
moon = new FilledOval(50, 50, 100, 100, canvas);
```

This will change the picture drawn so that it resembles the one shown in Figure 2.10. The moon and the cloud still overlap, but now part of the cloud is hidden behind the moon rather than the other way around.

Section 2.4 Accessing the Location of the Mouse

The canvas views the collection of objects it has been asked to display as a series of layers. When a new object is created, it is placed in a layer above all the older objects on the canvas. When two objects overlap, the object on the lower level will be partially or completely hidden by the object on the upper level. In the code to produce Figure 2.9, the moon was created before the cloud and therefore was drawn as if it were underneath the cloud. In the revised code, the moon was created last and therefore was drawn as if it were above the cloud.

None of the methods we have considered so far changes this layering. Changing an object’s color or moving it will not change the way in which it is drawn when it overlaps with other objects. If we included code to move the moon shown in Figure 2.9 as we could move the sun in our RisingSun program, the moon would move vertically on the screen, but it would still remain on a layer below the cloud. It would therefore appear to slide upward while remaining behind the cloud. Even the hide and show methods preserve this ordering. If we show an object that has been hidden, it will reappear underneath the same objects that were above it before it was hidden. In particular, it does not appear at the top level as if it had just been created.

There are, however, several methods that enable a programmer to rearrange the layering of objects on the screen. The methods sendForward and sendBackward move an object up or down one layer. The methods sendToFront and sendToBack move an object to the top or bottom of all the layers drawn. Thus, if we had created the picture shown in Figure 2.9 and then executed either the command

```java
cloud.sendBackward();
```

or the command

```java
moon.sendToFront();
```

the picture displayed would change to look like Figure 2.10.

2.4 Accessing the Location of the Mouse

In the header of every mouse-event-handling method we have included the phrase Location point in parentheses after the name of the method. Now that we have explained what a Location point is, we can explain the purpose of this phrase. It provides a means by which we can refer to the point at which the mouse cursor was located when the event handled by a method occurred. Basically, within the body of a mouse-event-handling method that includes the phrase Location point we can use the name point to refer to a Location object that describes where in the canvas the mouse was positioned.

As a simple example, we can write a variant of our very first example program TouchyWindow. This new program will display a bit of text on the screen when the mouse is pressed, just like TouchyWindow, but:

1. it will display the word “Pressed” instead of the phrase “I’m Touched”.
2. it will place the word where the mouse was clicked instead of in the center of the canvas, and finally
3. it will not erase the canvas each time the mouse is released
The code for this new example is shown below.

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

// A program that displays the word "Pressed" wherever
// the mouse is pressed
public class Pressed extends WindowController {
    public void onMousePress( Location point ) {
        new Text( "Pressed", point, canvas );
    }
}
```

Note that the name point is used in the construction that places the text “Pressed” on the screen. It appears in place of an explicit pair of x and y coordinates. Because point is included in the method’s header, Java knows that we want the computer to make this name refer to the location which the mouse was clicked. Therefore Java will place the word “Pressed” wherever the mouse is pressed.

You may have noticed that the phrase location point is syntactically very similar to a instance variable declaration. It is composed of a name we want to use preceded by the name of the class of things to which the name will refer. All that is missing is the word private. In fact this phrase is another form of declaration known as a formal parameter declaration, and a name such as point that is included in such a declaration is called a formal parameter name or simply a formal parameter.

As in instance variable declarations, we are free to use any name we want when we declare a formal parameter. There is nothing special about the name point (except that we have used it in all our examples so far). We can choose any name we want for the mouse location as long as we place the name after the word Location in the header of the method. For example, the program shown in Figure 2.11 uses the name mousePosition as a formal parameter instead of point, will behave exactly like the version that used the name point.

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

// A program that displays the word "Pressed" wherever
// the mouse is pressed
public class Pressed extends WindowController {
    public void onMousePress( Location mousePosition ) {
        new Text( "Pressed", mousePosition, canvas );
    }
}
```

Figure 2.11 Using a different parameter name

Another important aspect of the behavior of formal parameter names like mousePosition and point is that each parameter name is meaningful only within the method whose header contains its declaration. To illustrate this, consider the program shown in Figure 2.12. This program displays the word “Pressed” at the current mouse position each time the mouse button is pushed, and it displays the word “Released” at the current mouse position each time the mouse is released.

The onMousePress method in this example is identical to the corresponding method from the Pressed example except for the word it displays and the name chosen for its formal parameter.

The onMouseRelease method is also quite similar to the earlier program’s onMousePress.

Suppose now that we wanted to write another program that would display the words “Pressed” and “Released” just as in the UpsAndDowns example, but would also draw a line connecting the point where the mouse button was pressed to the point where the mouse button was released. A snapshot of what the window of such a program would look like right after the mouse was pressed, dragged across the screen, and then released is shown in Figure 2.13.

Given that the UpsAndDowns program has names that refer to both the point where the mouse button was pressed and the point where the mouse button was released, it might seem quite easy to modify this program to add the desired line-drawing feature. In particular, it probably seems that we could simply add a construction of the form:

```java
new Line( pressPoint, releasePoint, canvas );
```

to the program’s onMouseRelease method.

THIS WILL NOT WORK!

Because pressPoint is declared as a formal parameter in onMousePress, Java will not allow the programmer to refer to it in any other method. Java will treat the use of this name

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

// A program that displays the words "Pressed" and
// "Released" where the mouse button is pressed and
// released.
public class UpsAndDowns extends WindowController {
    public void onMousePress( Location pressPoint ) {
        new Text( "Pressed", pressPoint, canvas );
    }

    public void onMouseRelease( Location releasePoint ) {
        new Text( "Released", releasePoint, canvas );
    }
}
```

Figure 2.12 A program to record mouse-button changes
in onMouseRelease as an error and refuse to run the program. In general, formal parameter names cannot be used to share information between two different methods.

If we want to share information about the mouse location between two event-handling methods we must use formal parameters and instance variables together. We have already seen that instance variables can be used to share information between methods. In the RisingSun example, it began method created the oval that represented the sun and the onMousePress later moved it oval. This was arranged by associating the name sun with the oval.

In order to write a program to draw a line between the points where the mouse was pressed and released, we will have to associate an instance variable name with the location where the mouse was pressed. This variable will then make it possible for onMousePress to share the needed information with onMouseRelease. We will choose firstPoint as the name for the variable.

As always, we will have to add both a declaration and an assignment involving this new instance variable. Accordingly, the completed program will look like the code shown in Figure 2.14.

When the mouse is pressed, the assignment

```
firstPoint = pressPoint;
```

associates the name firstPoint with the Location of the mouse. This assignment is interesting in several ways. It is the first assignment we have encountered in which the text on the right side of the equal sign is something other than the construction of a new object. In the general form of the assignment statement, the text on the right side of the equal sign can be any phrase that describes the object we wish to associate with the name on the left. So, in this case, rather than creating a new object, we take the existing Location object that is named pressPoint and give it a second name, firstPoint. Immediately after this assignment, the Location that describes the mouse position has two names. This may seem unusual, but it isn’t. Most of us can also be identified with multiple names (e.g., your first name, a nickname, or Mr. or Ms. followed by your last name).

This assignment also illustrates the fact that a name in a Java program may refer to different things at different times. Suppose when the program starts you click the mouse at the point where

coordinates (5.5). As soon as you do this, onMousePress is invoked and the name firstPoint is associated with a Location that represents the point (5.5). If you then drag the mouse across the screen, release the mouse button, and then press it again at the point (150, 140), onMousePress is invoked again and the assignment statement in its body tells Java to associate firstPoint with a Location that describes the point (150, 140). At this point, Java forgets that firstPoint ever referred to (5.5). A name in Java may refer to different objects at different times, but at any given time it refers to exactly one thing (or to nothing if no value has yet been assigned to the name).

In fact, the values associated with most instance variables are changed frequently rather than remaining fixed like the variable in our RisingSun example. To illustrate the usefulness of such changes, we can write a simple drawing program.

Complex drawing programs provide many tools for drawing shapes, lines, and curves on the screen. We will write a program to implement the behavior of just one of these tools, the one that allows the user to scribble on the screen with the mouse as if it were a pencil. A sample of the kind of scribbling we have in mind is shown in Figure 2.15. The program should allow the user to trace a line on the screen by depressing the mouse button and then dragging the mouse around the screen with the button depressed. The program should not draw anything if the mouse is moved without depressing the button.
The trick to writing this program is to realize that what appears to be a curved line on a computer screen is really just a lot of straight lines hooked together. In particular, to write this program, we want to do is notice each time the mouse is moved (with the button depressed) and draw a line from the place where the mouse started to its new position. Each time the mouse is moved with its button depressed, Java will follow the instruction in the onMouseDrag method. In this method, we will include an instruction like:

```java
new Line( previousPosition, currentPosition, canvas );
```

where `previousPosition` and `currentPosition` are names that refer to the previous and current positions of the mouse. The trick is to also include statements that will ensure that Java associates these names with the correct locations.

Associating the correct location with the name `currentPosition` is easy. When `onMouseDrag` is invoked, the computer will automatically associate the current mouse location with whatever name we choose to use as the method's formal parameter name. So, if the head we use when declaring `onMouseDrag` looks like:

```java
public void onMouseDrag( Location currentPosition )
```

we can assume that the name `currentPosition` will refer to the location of the mouse when the method is invoked.

Getting the correct location associated with `previousPosition` is a bit trickier. This carefully for a moment about the beginning of the process of drawing with this program. To get the user will position the mouse wherever the first line is to be drawn. Then the user will depress the mouse button and begin to drag the mouse. The first line drawn should start at the position when the mouse button was depressed and extend to the position to which the mouse was first dragged. This situation is similar to the problem we faced when we wanted to draw a line between two points where the mouse was depressed and the point where the mouse was released. The position at which the mouse button is first depressed will be available through the formal parameter in the `onMousePress` method, but we need to access it in the `onMouseDrag` method because

Unfortunately, if we actually use this code, the program will not behave as we want. For example, if we were to start near the upper left corner of the screen and then drag the mouse in an anticlockwise, hoping to draw the picture shown below on the left, the program would actually draw the picture shown on the right.

The problem is that we are not changing the point associated with `previousPosition` as often as we should. In `onMousePress`, we tell the computer to make this name refer to the point where the mouse is first pressed, and it continues to refer to this point until the mouse is released and pressed again. As a result, as we drag the mouse, all the lines created start at the point where the mouse button was first pressed. Instead, after the first line has been drawn, we always want `previousPosition` to refer to the mouse's position when the last line was drawn. To do this, we must add the assignment statement:

```java
previousPosition = currentPosition;
```

at the end of the `onMouseDrag` method, yielding the complete program shown in Figure 2.16.
58

Chapter 2. What's in a Name

\import{objectdraw.*};
\import{java.awt.*};

// This program allows its user to draw simple lines on the screen
// using the mouse as if it were a pencil

class Scribble extends WindowController {

private Location previousPosition; // Last known position of mouse

// When the mouse button is depressed, note its location
public void onMousePress(Location pressPoint) {
    previousPosition = pressPoint;
}

// Connect current and previous mouse positions with a line
public void onMouseDrag(Location currntPosition) {
    new Line(previousPosition, currntPosition, canvas);
    previousPosition = currntPosition;
}

Figure 2.15 A simple-drawing program

Exercise 2.5.1

Explain why the assignment statement is still needed in onMousePress as shown below; even though previousPosition is always updated in onMouseDrag:

public void onMousePress(Location pressPoint) {
    previousPosition = pressPoint;
}

2.6 Summary

In this chapter we explored the importance of the use of names to refer to the objects our program manipulates. Instance variable names were used to share information between methods, and form parameter names provided a means to pass information from outside the program into a method body. We saw that these names had to be declared before we could use them in a program, and in the case of instance variable names, that we had to use an assignment statement to associate each name with a particular object.

We learned more about displaying simple graphical objects on our program's canvas and learned how to modify the properties of these objects using mutator methods. In addition, we learned the notions of "objects", constructions, and mutator methods in Java extend to types of object such as Colors and Locations that are not themselves visible on our screen.

Section 2.7 Chapter Review Problems

In case you did not notice, the last example program we discussed, the Scribble program, was different from most of the other examples in one important regard. It actually is (at least part of) a useful program. This reflects the fact that the features we have explored are fundamental to the construction of all Java programs. With this background, we are now well prepared to expand our knowledge of the facilities Java provides.

Exercise 2.7.1

Revise TouchyWindow so that the Text object has a name. Create the Text object in the begin method, but don't show it unless the mouse is pressed. Change the onMouseRelease method so that it does not use the canvas.clear() command, but hides the text when the mouse is released.

Exercise 2.7.2

Suppose that in the Scribble class you want to clear the contents of the canvas if the mouse exits the window. Write the appropriate method to do this.

Exercise 2.7.3

What is wrong with the following line of code?

message = Text( "Welcome to Hangman v1.0", 60, 60, canvas );

Exercise 2.7.4

Write a program that draws a filled 20-by-20 square at the mouse location each time the mouse is pressed. When the mouse is released, the frame of the square should remain. The user should then be able to press again to create a new filled square at a new location that will leave a frame when the mouse is released.

Exercise 2.7.5

What is an instance variable and what are the two important steps needed to create it before being able to use it?

Exercise 2.7.6

Modify Scribble so that everything drawn in the window is red instead of black.

Exercise 2.7.7

Look at the two pieces of code. Do they produce the same or different outcomes? If the outcomes are different, explain the difference.
Chapter 2  What's in a Name

2.8 Programming Problems

Exercise 2.8.1

Write a simple program that does the following:

• When the program begins, a red square with a black frame is drawn. Each side of the square is 60 pixels, and the square is located 70 pixels from the right and 60 pixels down from the top left corner.
• When the mouse is clicked, the square turns blue.
• When the mouse exits the window, the square disappears.
• When the mouse re-enters the window, the square reappears and is once again red.

Exercise 2.8.2

Write a program called DrawRect. The program should display the frame of a rectangle where the mouse is pressed. The rectangle should be 100 by 100 pixels with the upper left corner at the point where the mouse was clicked. When the mouse is dragged, the frame should move with the mouse, with the upper left corner of the rectangle always following the cursor. When the mouse is released, the rectangle should be filled in so it is no longer just a frame.

Exercise 2.8.3

a. Write a simple program called Measles with these features:

• When the program begins there should be a message telling the user "Measles: Click to catch the disease!"
CHAPTER 3

Working with Numbers

In the preceding chapter, we used numbers extensively to manipulate graphical objects. They were used to specify coordinates, dimensions, and even colors. While we use numbers to describe what we want to do to various graphical objects, we did not do much of interest with the numbers themselves. Numbers are of critical importance to Java. Just as Java provides operations to let us work with graphical objects, it provides operations to let us work with numbers. In this chapter, we will explore some of these operations. We will see how to obtain numerical values describing properties of existing objects, how to perform basic arithmetic computations, how to work with numeric variables, and how to display numeric values.

3.1 Introduction to Accessor Methods

When we perform a construction of the form

```java
new Location( 50, 150 )
```

we combine two numbers to form a single `Location` object. There are many situations where this is useful to do the opposite. That is, we have a `Location` object and want to access the numeric values of the x and y coordinates associated with that location. Suppose, for example, that we decided to change the `RisingSun` program so that, rather than having to click the mouse to move the sun, the user could simply drag the mouse up and down and the sun would follow it. The desired behavior is similar to the way in which the scrollbar found in many programs reacts to mouse movement. If you grab the scroll box displayed in

vertical scrollbar you can move the scroll box up and down by moving the mouse, but you cannot move the scroll box to the left or right. Similarly, in the program we have in mind, even if the mouse is dragged about in spirals, the circle that represents the sun should only move straight up and down so that its y coordinate is always the same as the y coordinate of the mouse’s current position.

In this new version of `RisingSun`, which we will name `ScrollingSun`, we need to replace the `onMouseClicked` method from the previous version with an `onMouseDrag` method of the form:

```java
public void onMouseDrag( Location mousePosition ) {
    sun.moveTo( ... );
}
```

The question is: What should we provide as parameter information to the `moveTo` method?

We want the sun to move to the position in the canvas whose x coordinate is the same as the sun’s initial x coordinate, 50, and whose y coordinate is equal to the y coordinate of the mouse. We can handle the x coordinate by simply typing 50 as the first parameter to the `moveTo` method. The hard part is providing the y coordinate of the mouse’s position.

The `Location` object named `mousePosition` certainly contains enough information to determine the y coordinate of the mouse. Java lets us ask the `Location` object to provide this information through a mechanism called an accessor method. Like the mutator methods discussed in the preceding chapter, a small collection of accessor methods is associated with each class of objects. Objects of the `Location` class support two accessor methods, `getX` and `getY`.

To use an accessor method we write a name that refers to the object that is the target of the request followed by a period, the name of the method to be applied, and a parenthesized list of parameter values. So, to position the sun appropriately in the `onMouseDrag` method, we should say:

```java
sun.moveTo( 50, mousePosition.getY() );
```

You should observe that this notation for using accessor methods is identical to the notation used for mutator methods. In particular, in the case that no parameters are provided, you still need to include a set of empty parentheses after the name of the method.

The complete text of the revised program is shown in Figure 3.1. With the exception of the substitution of the `onMouseDrag` method for the `onMouseClicked` method, the only difference between this program and the one shown in Figure 2.3 is that the instructions displayed by the `begin` method have been modified.

Accessor methods are also associated with objects of the graphics classes introduced in the last chapter. The location and dimensions of a graphical object can be accessed using methods named `getX`, `getY`, `getWidth`, and `getHeight`. Like the methods discussed above, these accessor methods provide numeric information about an object. In addition, there are accessor methods associated with graphical objects that provide other forms of information. There is a method named `getColo` that returns the color of a graphical object. Similarly, `getLocation` returns a `Location` object describing an object’s current position.
// A program that produces an animation of the sun rising and setting.
// The animation is driven by dragging the mouse.
//
public class ScrollinSun extends WindowController {

private FilledOval sun; // Circle that represents the sun

// Place the sun and some brief instructions on the screen
public void begin() {
    sun = new FilledOval(50, 150, 100, 100, canvas);
    new Text("Drag the mouse up or down", 20, 20, canvas);
}

// Move the sun to follow the mouse's vertical motion
public void onMouseDrag(Location mousePosition) {
    sun.moveTo(50, mousePosition.getY());
}

Figure 3.1 Program to make the sun scroll with the mouse

Exercise 3.1.1

Is the translate method of the Location class an accessor method? Why or why not?

Exercise 3.1.2

What is the difference between an accessor method and a mutator method? What can an accessor method do that a mutator method cannot?

3.2 Accessing Numerical Attributes of the Canvas

In the last chapter, to keep things simple, we assumed that we knew the size of the window in which our programs would run. For example, the DrawGrid program presented in Section 2.2, draws a grid like the one shown in Figure 3.2. The bars in this grid are created by construction of the form:

new FilledRect(verticalCorner, 5, 200, canvas);
new FilledRect(horizontalCorner, 200, 5, canvas);

placed within the program's onMouseClick method. The vertical rectangles created by these constructions are 200 pixels tall and the horizontal rectangles are 200 pixels wide. The result is that drawing looks fine if the window is exactly 200 by 200, which is the window size we showed in the figures that illustrated the drawings the program would produce, but it would not look right if the window was larger. If the window was wider, we would want the program to draw with horizontal rectangles. If the window was taller, we would want taller vertical rectangles.

When we write a program, we cannot be certain of the size of the window in which it will run. The size of the window is not determined by our Java code. For the types of programs discussed in this text, the window size is determined by specifications written in a different language, HTML, or Hypertext Markup Language, the language used to describe the content of Web pages.

Given that the canvas size is unpredictable, it would be best to write programs that determine the actual size of the canvas while running and adjust the objects they draw accordingly. We have seen that the canvas provides a mutator method named clear. It also supports two accessor methods named getWidth and getHeight which allow a program to determine the dimensions of the canvas.
3.3 Expressions and Statements

It is important to observe that accessor methods serve a very different function than do mutator methods. A mutator method instructs an object to change in some way. An accessor method requests some information about the object's current state.

Simply asking an object for information is rarely worthwhile by itself. We also need to tell Java what to do with the information requested. We would never write an instruction of the form

```java
mousePosition.getY();
```

Such an instruction would tell Java to ask the Location named mousePosition for its coordinate but make no use of the information. Instead, we use accessor methods in places where Java expects the programmer to describe an object or value. For example, a accessor method can be used to describe a parameter to a construction as in

```java
FilledRect( 10, mousePosition.getY(), 2, 2, canvas );
```

or to describe the parameters for another method, like moveTo.

This notion of a phrase that describes an object or value is important enough to deserve a name. Such phrases are called expressions. We have already seen several different sorts of phrases that Java recognizes as examples of expressions.

---

**Exercise 3.2.1**

Write the `begin` method for a program that draws an X through the canvas using the `getHeigh` and `getWidth` accessor methods of the canvas.

---

**Section 3.3 Expressions and Statements**

Where numeric information is needed, we have often explicitly included the numbers to use by typing numeric literals like "50" and "150" as in the invocation

```java
box.moveTo( 50, 150 );
```

In other situations, accessor method invocations have been used to describe numeric values as in

```java
box.moveTo( 50, point.getY() );
```

Where non-numeric information was needed, we have either used a `construction` to create the needed information as in

```java
sun.setColor( new Color( 200, 100, 0 ) );
```

or provided a name that was associated with the desired object as in

```java
sun.setColor( purple );
```

Thus, numeric literals, instance variable names, constructions, and invocations of accessor methods can all be used as expressions.

In any context where it is necessary to describe an object or value, Java will accept any of the forms of expressions we have introduced. In a context where Java expects the programmer to describe a Color, we can equally well use a name associated with a Color, a Color construction, or an invocation of the `getColor` accessor method. Java is, however, picky about the type of value described by an expression. In a context where Java expects us to provide a Color, we can't provide an expression that describes a number or a Location instead.

Not all phrases found in a Java program are expressions. The invocation of a mutator method such as

```java
sun.move( 0, -5 );
```

is an example of a phrase that is not an expression. This phrase contains subparts that are expressions, the numeric literals 0 and -5, but is not an expression itself because it does not describe a value. Instead, this phrase instructs Java to perform an action. Such phrases are called instructions or statements. Statements instruct Java to perform actions that either produce output visible to the user or alter the internal state of the computer in a way that will affect the future behavior of the program. The body of each method we define in a Java program must be a sequence of statements.

We have seen three types of statements at this point. The invocation of a mutator method, such as

```java
sun.move( 0, -5 );
```

is one type of statement. The second type is the assignment statement. It instructs the computer to perform the action of associating a name with an object or value. Note that the phrase on the right side of an assignment must be an expression.

The third type of statement we have encountered is the construction. We have used instructions like:

```java
new Text( "Pressed", mousePosition, canvas );
```

to place graphics on the canvas. We have already stated, however, that a construction is an expression. Which is it? It is both. A construction like the one shown above describes an object.
Chapter 3: Working with Numbers

Section 3.4 Arithmetic Expressions

It can be used in contexts where expressions are required. At the same time, the construction of a graphical object involves the action of changing the contents of the display. Accordingly, the construction by itself can be viewed as a statement.

There are constructions that merely describe an object without having an associated action that affects any aspect of the state of the program. For example,

```java
new Location( 10, 20 );
```

It does not make much sense to use such a construction as a command, because a program that contained such a command would behave the same if the command were removed. Java, however, does not prevent the programmer from writing such nonsense. In fact, Java will allow the programmer to use many kinds of expressions as if they were commands by simply placing semicolons after the expressions. In a sensible program, however, the only expressions we have introduced so far that make sense as commands are constructions of graphical objects.

**Exercise 3.3.1**

Which of the following statements could actually be useful in a program and which could not?

- a. `sun.getColor();`
- b. `new Text( "Hello", point, canvas );`
- c. `new Color( 60, 60, 60 );`
- d. `myLocation = new Location( 50, 50 );`

3.4 Arithmetic Expressions

Sometimes it is very useful to describe a numeric value to Java by providing a formula to compute the number. For example, to describe the x coordinate of a point slightly to the left of the current mouse position we might say something like:

```java
mousePosition.getX() - 10
```

Java allows the programmer to use such formulae and calls them *arithmetic expressions*. As an example of the use of arithmetic expressions, we can make some additional improvements to the ScrollingSun program.

Using arithmetic expressions involving the `getWidth` and `getHeight` methods of the canvas we can revise the ScrollingSun program so that it adjusts the size and position of the circle that represents the sun based on the size of the canvas. To maintain the proportions used in the original program as shown in Figure 3.3:

- the diameter of the circle should be half the width of the canvas,
- the left edge of the circle should fall one-quarter of the width of the canvas from the edge of the canvas,
- initially, the top of the circle should be placed so that half the circle is visible above the horizon.

To do this, the top of the circle should be half of its diameter above the bottom of the canvas.

It would also be appropriate to center the text of the instructions horizontally on the canvas. The indentation of the text from the left edge of the canvas should be equal to that on the right side. So, it should be half of the difference between the width of the text and the width of the canvas.

Each of these verbal descriptions can be turned into a formula, which can then be used in the program. The diameter of the circle, which is the value that should be specified as the width and height in the `FillOval` construction, would be described as:

```java
canvas.getWidth() / 2
```

The y coordinate value for the left edge of the circle would be given by the formula:

```java
canvas.getHeight() / 4
```

The y coordinate for the top of the circle would be described as:

```java
canvas.getHeight() - canvas.getWidth() / 4
```

Finally, the x coordinate for the left edge of the instructions should be:

```java
(canvas.getWidth() - instructions.getWidth() ) / 2
```

The complete `ScrollingSun` program using such formulae is shown in Figure 3.4. In most cases, we have simply replaced a number used as an expression by the appropriate formula. The only slight complication is the code to center the text. We cannot use the `getPos` method associated with the `Text` object until it has been constructed. So, when we construct the `Text`, we just use 0 as its x coordinate value. Then, once it exists, we use the `getPos` method to figure out how big it is. Finally, we use `moveTo` to place the `Text` where it belongs.

The arithmetic expressions shown in the preceding examples use only two of the arithmetic operators, subtraction and division. It is also possible to use multiplication and addition. The symbols used to indicate addition, subtraction and division are the standard symbols from mathematics: +, -, and / respectively. Thus, to say...
A program that produces an animation of the sun rising and setting.

```java
public class ScrollingSun extends WindowController {
    private Filled Oval sun; // Circle that represents the sun
    private Text instructions; // Display of instructions

    // Place the sun and some brief instructions on the screen
    public void begin() {
        sun = new Filled Oval( canvas.getWidth()/4, canvas.getHeight()/4 - canvas.getWidth()/4, canvas.getWidth()/2, canvas.getHeight()/2, canvas );

        instructions = new Text( "Drag the mouse up or down", 0, 0, canvas );
        instructions.moveTo( ( canvas.getWidth() - instructions.getWidth() ) /2, 20 );
    }

    // Move the sun to follow the mouse's vertical motion
    public void onMouseDrag( Location mousePosition ) {
        sun.moveTo( canvas.getWidth()/4, mousePosition.getY() );
        instructions.hide();
    }

    // Move the sun back to its starting position and redisplay
    public void onMouseExit( Location point ) {
        sun.moveTo( canvas.getWidth()/4, canvas.getHeight() - canvas.getWidth()/4 );
        instructions.show();
    }
}
```

Figure 3.4: Program to make the sun scroll with the mouse

"2 times the width of the canvas" one would write

```
2 * canvas.getWidth()
```

The values being operated upon are called **operands**. In the example above of multiplication, the operands are 2 and `canvas.getWidth()`.

The behavior of the division operator frequently surprises beginners. When both operands of a division are integers, Java returns the integer value of the quotient and simply discards any remainder. Thus, when asked to compute 9/5, Java determines that the answer is 1 with a remainder of 4, discards the remainder, and returns 1 as the answer rather than 1.8. We will examine the logic behind this behavior in Chapter 5.

---

**Section 3.4 Arithmetic Expressions**

The following table summarizes the most commonly used arithmetic operators in Java.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>Addition</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiplication</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division</td>
</tr>
</tbody>
</table>

---

**Exercise 3.4.1**

Write the `begin` method for a program that draws two lines that form a cross (consisting of a horizontal line and a vertical line as shown in Figure 3.5) on the canvas with the intersection of the two lines in the center of the canvas.

---

**3.4.1 Ordering of Arithmetic Operations**

Two of the arithmetic expressions used in the ScrollingSun program shown in Figure 3.4 illustrate an issue a programmer must be aware of when writing such expressions: the rules used to determine the order in which operations are performed. The first of these is the expression

```
( canvas.getHeight() - instructions.getWidth() ) /2
```

which is used in the `begin` method to position the instructions. The second determines the initial y-coordinate for the top of the sun:

```
canvas.getHeight() - canvas.getWidth()/4
```

Both involve a subtraction and a division. The first, however, uses parentheses to make it clear that the subtraction should be performed first and that the result of the subtraction should be divided by 2. The correct interpretation of the second expression is not as clear. In fact, Java will first divide the width of the canvas by 4 and then subtract the result of this division from the height of the canvas. In the absence of parentheses that dictate otherwise, Java always performs divisions in an expression before subtractions. Thus, the second formula is equivalent to the formula:

```
canvas.getHeight() - ( canvas.getWidth()/4 )
```
The rule that division is performed before subtraction is an example of a precedence rule. When evaluating simple arithmetic expressions, Java follows two basic precedence rules:

- Perform divisions and multiplications before additions and subtractions. We therefore say that division has higher precedence than addition but that division and multiplication are of equal precedence.
- When performing operations of equal precedence (i.e., additions and subtractions or division and multiplications), perform the operations in order from left to right as written.

Parentheses can be used to override these precedence rules, as seen in the first example above. Any part of a formula enclosed in parentheses will be evaluated before its result can be used in an expression outside the parentheses.

**Exercise 3.4.2**

What are the values of the following expressions?

a. \(4 + 3 \times 8 / 2 - 3\)

b. \((4 + 3) \times 8 / 2 - 3\)

c. \(4 + (3 \times 8) / 2 - 3\)

d. \(4 + 3 \times (8 / 2) - 3\)

e. \((4 + 3) \times 8 / (2 - 3)\)

f. \((4 + 3) / 8 / 2 - 3\)

### 3.5 Numeric Instance Variables

In the previous chapter, we saw that it is sometimes necessary to associate instance variable names with `Location`s or other objects to enable one method to communicate information to another method. Unsurprisingly, it is often useful to associate names with numeric values in similar ways. We can illustrate this by adding yet another feature to our `RisingSun` program.

As the real sun rises, the sky becomes brighter and brighter. Suppose we wanted to try simulating this in our program. For this example we will return to the original interface where the user clicks repeatedly to make the sun rise. Now, when the sun is near the bottom of the screen, it would look like the background to be filled with a dark shade of gray. We can do this by constructing a `FillerRectangle` as big as the canvas and setting its color to an appropriate shade of gray. As the user clicks, we can make the background become lighter by using `setColor` to replace the origin shade of gray with lighter and lighter shades until it is eventually white.

We have seen that each color is described by a triple of numbers giving the amounts of red, green, and blue in the color. Shades of gray correspond to triples in which all three values are the same. The bigger the number used, the brighter the shade. So,

```
new Color( 0, 0, 0 )
```

describes black,

```
new Color( 50, 50, 50 )
```

describes a dark shade of gray.

```
new Color( 200, 200, 200 )
```

describes white.

To control the brightness of the background, we would like to associate an instance variable with the number to be used to generate the shade of gray currently desired. We will use the name `brightness` for this variable. This name can then be used to construct shades of gray for the background by using the construction:

```
new Color( brightness, brightness, brightness )
```

To use such a name, of course, we must first declare the name and then add assignment statements to ensure that it is associated with the correct number at each point in time as the program executes. In each instance variable declaration, the declared name must be preceded by the name of the type of information with which the declared name will be associated. Java distinguishes between numbers that include fractional components like 3.14 and 93, and integers like 17 and -45. It uses the name `double` to describe numbers with fractional components and the name `int` to describe integers. In Chapter 5, we will discuss why Java distinguishes between integers and nonintegers in this way. For now, we merely observe that the values associated with `brightness` in our program will always be integers. Accordingly, to declare the name `brightness` we say

```
private int brightness;
```

In the `begin` method, we will associate the name `brightness` with a number corresponding to a dark shade of gray by including an assignment statement of the form:

```
brightness = 50;
```

Each time the user clicks the mouse, we want to associate a larger number with `brightness`. We can do this by including the assignment statement:

```
brightness = brightness + 1;
```

in `onMouseClicked`. This statement tells the computer to take the current value associated with the name `brightness`, add one to it, and then associate the name `brightness` with the result.

The first time the mouse is clicked, `brightness` will be associated with the value 50 specified in the `begin` method. The result of adding 1 to 50 is 51. So, after the assignment statement is executed, `brightness` will be associated with the value 51. The next time the mouse is clicked, Java will add 1 to the new value of `brightness`, 51, and set it equal to 52. Thus, each time the mouse is clicked, the value of `brightness` will become 1 greater and the color generated by the construction

```
new Color( brightness, brightness, brightness )
```

will become a bit brighter.

The action of increasing the value associated with a numerical variable by 1 as described by the assignment

```
brightness = brightness + 1;
```

would be a fairly light shade of gray, and

```
new Color( 255, 255, 255 )
```

is white.
A program to simulate the brightening of the sky at sunrise:

```java
public class LightenUp extends WindowController {

    private FilledRect sky; // Background rectangle
    private int brightness; // Brightness of sky’s color
    private FilledOval sun; // Circle that represents the sun

    public void begin() {
        // Create the sky and make it a dark gray
        brightness = 50;
        sky = new FilledRect(0, 0, canvas.getWidth(), canvas.getHeight(), canvas);
        sky.setColor(new Color(brightness, brightness, brightness, brightness));

        // Place the sun and some brief instructions on the screen
        sun = new FilledOval(50, 150, 100, 100, canvas);
        sun.setColor(new Color(brightness, brightness, brightness, brightness));
        text = Text("Please click the mouse repeatedly", 20, 20, canvas);
    }

    // Brighten the sky and move the sun with each click
    public void onMouseClick(int locationPoint) {
        brightness += 1;
        sky.setColor(new Color(brightness, brightness, brightness, brightness));
        sun.move(0, -5);
    }

    Figure 3.8 Using numeric instance variables:

    is so common that Java provides a special shorthand notation. We can instruct Java to increase
    the value associated with a name by 1 simply following the name by a pair of adjacent plus
    signs, as in:

    brightness++;

    The notation

    brightness--;

    can also be used to tell Java to reduce the value associated with a numeric variable by 1.
    With these details we can complete the program. The code is shown in Figure 3.6.

    Exercise 3.5.1

    What would you expect to happen in LightenUp if you changed the assignment of brightness
    to brightness = 200 and then changed the first line in the onMouseClick method to if
    following?

    brightness = brightness - 1;
```

Section 3.6 Initializers

We have seen that we must complete two steps before using a variable. We must include a
declaration to tell Java that we plan to use the name and to specify the type of information
with which the name will be associated. We must also include an assignment statement associating
a particular meaning with the name before using it.

Although declaration and assignment are logically separate actions, it is often useful to combine
them. When we declare a variable, we often want to know what value we will assign to it first. Putting
a variable’s declaration and the assignment of its initial value together (and topping it off with a
comment describing the purpose of the variable being declared) can often improve the readability
of a program.

To make this possible, Java allows the programmer to include an initial value for a variable in
the variable’s declaration. The result looks like an assignment statement preceded by private
and the name of the type of the variable. It is, however, interpreted as a declaration by Java.

In the example considered in the preceding section, the variable brightness is introduced
by the instance variable declaration:

```java
private int brightness;
```

Then, the initial value of this variable is set to 50 by the assignment statement

```java
brightness = 50;
```

in the begin method. Using Java’s notation for initialized variable declarations, we could remove
this assignment from the begin method and rewrite the variable’s declaration as

```java
private int brightness = 50; // Brightness of sky’s color
```

When an initial value is included in an instance variable declaration in this way, the variable’s
value is set before the begin method or any other method is invoked.

Although initialized declarations are most often used with numeric variables, Java will allow
you to include an initializer in the declaration of a variable of any type. For example, if we wanted
to keep a name associated with the current color of the sky, we might declare this new variable as

```java
private Color skyShade = new Color(brightness, brightness, brightness);
```

As this example suggests, Java will allow you to use expressions of many forms to describe
the initial value of a variable. The main restriction is that any names used in the expression
must already be declared and associated with a value. For example, the declaration shown for skyShade
above would only be valid if preceded by a declaration of brightness. Furthermore, it will only
function as desired if brightness is assigned its initial value in its declaration (as shown above)
rather than in the begin method (as in the original version of the program).
3.7 Naming Numeric Constants

In Chapter 1, we introduced the concept. Comments are a rather interesting construct precisely because they have no effect on how the programs that contain them actually behave. As far as the computer is concerned, comments are useless. Nevertheless, a special notation is included in Java (and in almost every other programming language) to enable us to include these "useless" comments in our program. This reflects the fact that it is very important that your programs be made as easy as possible to understand.

Comments are just one mechanism Java provides to help you improve the readability of your programs. Another similar feature is the fact that Java ignores white space (i.e., blanks, tabs, and new lines) added to a program. This makes it possible to use blank lines and indentation structures to reflect its logical organization. Such formatting can be an important aid to an individual trying to understand the code.

The appropriate use of comments and good program layout are aspects of good programming style. To the beginner, the importance of good style may be difficult to appreciate. Short example programs can generally be read and understood even if they are not designed to be as readable as possible. As a programmer becomes more experienced and becomes involved in the construction of larger programs, the practice of good programming style becomes more critical. It is very easy to produce a large program that is impossible for any human reader (including its author) to understand. Accordingly, it is best to develop the habit of always considering how to make the code you write as clear as possible from the very beginning.

Unfortunately, there is one rule of good style that we have been violating in almost all of our example programs. In this section, we will introduce a Java mechanism designed to support the rule of good style, and then we will begin following the rules ourselves.

In nearly all the examples we have presented, we have specified coordinates and dimensions of objects using numbers. We have also used numeric values to specify object colors and to determine how far certain items should move in reaction to a user action.

In most of these examples, we have simply typed numeric literals that specify the desire information into the constructions and method invocations where they were needed. While this approach certainly works, it is considered poor style. To appreciate why, just consider the instruction

```java
new FilledOval(50, 150, 100, 100, canvas);
```

By now, you have seen this instruction often enough that you may recognize it and know what it is for. It is the construction that creates the circle representing the sun in our rising sun example. Suppose, however, that you encountered this construction while reading through a complex Java program composed of thousands of lines of code. How would you guess the purpose of it?

The preferred alternative to using numbers explicitly in program instructions is to associate variable names with the numeric values and then to use the names in place of the numbers. For example, the above construction might be rewritten as

```java
new FilledOval( sunCornerX, sunCornerY, sunSize, sunSize, canvas );
```

Of course, if we want to use names like this instead of typing the numbers themselves, we will need to declare the names and initialize them. For example, we might say

```java
// Values that determine position and size of the sun
private int sunCornerX = 50;
private int sunCornerY = 150;
private int sunSize = 100;
```

There is one flaw with this alternative. If you were reading a large program and found a construction like the one shown above, finding the declarations of the three names used in the construction would not be enough to assure that you knew what the actual values employed by the construction would be. The problem is that there might be some other point in the program where values other than 50, 150, and 100 were assigned to the variables, thus changing their initial values. If the program you were reading was large, it could be time consuming to search the program to make certain such assignments did not occur.

To avoid this problem, Java provides a mechanism through which the programmer can assure the reader that the initial value assigned to a variable in its declaration will not be changed anywhere else in the program. To do this, the programmer simply adds the word "final" to the declaration after the word "private". Using this feature, the declarations above would be rewritten as

```java
// Constants that determine position and size of the sun
private final int sunCornerX = 50;
private final int sunCornerY = 150;
private final int sunSize = 100;
```

Including the word "final" in a declaration tells Java not to allow any assignment statement that would change the value of the variable being declared. That is, if the assignment

```java
sunSize = 200;
```

was included at some point in a program containing the final declaration shown above, the assignment would be reported as an error to the programmer, and Java would refuse to run the program.

There are two conventions followed by most Java programmers when using final in declarations. First, so that it is easy to identify names with fixed values when reading a program, such names are usually composed of all uppercase letters, with underscores being used to separate the parts of a multisword name. Second, because doing so may in some cases improve program efficiency, it is customary to add the modifier static to declarations that contain the modifier
Chapter 3 Working with Numbers

3.8 Displaying Numeric Information

We have seen how we can use the computer's ability to work with numbers to produce both drawings on the computer's screen. Sometimes, however, it is the numbers themselves rather than any drawing that we really want to see. The main purpose of many computer programs is to perform numerical calculations. Examples include programs that determine your taxes, determine your GPA, and estimate the time required to travel from one point to another. Such programs often simply display the numbers they compute rather than drawing a graph of some sort on the screen. Even programs that are not primarily focused on numerical computations often need to display numerical information. For example, a word processor might need to display the current page number. In this section we will describe two mechanisms in Java that can be used to display numerical information.

3.8.1 Displaying Numbers as Text

As a very simple first example, let's make the computer count. You probably don't remember if at some point in your early childhood you most likely had some adult by demonstrating your remarkable ability to count to 10 or 20 or maybe even higher. To enable the computer to produce an equally impressive demonstration of its counting abilities, we will construct a program that will count. It will start at 1 and move on to the next number each time the mouse is clicked. The current value will be displayed on the computer's screen.

We have already introduced one mechanism that can be used to display numbers on the screen. We just didn't mention at the time that it could be used with numbers. In the very first program...
essentially told the computer to count upward starting at 50. The instruction that we used to
progress through the different values of brightness was

```
brightness = brightness + 1;
```

A very similar assignment statement involving the variable theCount:

```
theCount = theCount + 1;
```

is what we need to complete the counting program described above.

Such a counting program is shown in Figure 3.8. The body of the onMoseClick method use
the assignment statement shown above to associate the next counting number with theCount each
time the mouse is clicked. Note that changing the value associated with the variable theCount
does not cause the value displayed by the Text object named countDisplay to change. If
change the Text displayed, we use a method named setText. This is a mutator method that ca
be used to change the information displayed by a Text object. It expects a single parameter, a
new information to be displayed. Like the first parameter expected in a Text construction, th
information can be a quoted sequence of characters, or a numeric value, or just about any oth
form of information we might want to display in textual form. The statement

```
countDisplay.setText( theCount );
```

in the onMoseClick method tells Java to change the information displayed by the Text obj
named countDisplay that was created in the begin method.

An alternative to using setText would be to clear the canvas and then construct a ne
Text object displaying the new value of theCount. Construction of a new object, however, is
fairly time-consuming process for the computer. When possible, it is better to reuse an existi
object rather than create a new one. Accordingly, in an example like this it is preferable to u
setText.

### 3.8.2 Using System.out.println

In a program that mixes graphical output with numerical or other textual information, Text objec
t and theText method are the most appropriate tools for displaying textual information. I
programs that only display textual information, there is another tool that is often simpler and m
appropriate, System.out.println.

All the output displayed by the Java programs we have considered so far appears in the windo
associated with the name canvas. These programs can, however, display output in another windo

There are several other mutator methods that can be applied to Text objects. In Chapter 2 we already showed th
show and hide could be used with Text objects. In addition, the move and moveTo methods can be used to reposition
Text objects, just as they can be used with rectangles and ovals. Finally, there are several special mutator methods
for use with Text objects. For example, the setTextSize method can be used to change the size of displayed Text.
It expects a single integer as a parameter, which it interprets as a font size.

```
countDisplay.setFontSize( 24 );
```

could be added to the end of our program’s begin method if we wanted to increase the size of the numbers
displayed.
### Section 3.9 Displaying Numeric Information

When a number displayed all by itself has little meaning. The difference between just displaying "3" and displaying "Strike 3" or "3 P.M." or "Line 3" can be quite significant. Accordingly, in many programs, rather than just displaying a number on the screen it is desirable to display a number combined with additional text that clarifies its meaning. Luckily, this can be done easily in Java with both Text objects and System.out.println.

When specifying the information to be displayed in a Text object or on the Java console, we can use the + operator to combine quoted text with numeric information. Suppose, for example, that we want our counting programs to display a message like "You have clicked 3 times" instead of just displaying 3. For the version that uses Text objects to place the information on the screen, we could accomplish this by replacing the command

```
countDisplay.setText( theCount );
```

shown in Figure 3.8 with the command

```
countDisplay.setText( "You have clicked " + theCount + " times" );
```

Similarly, for the println version shown in Figure 3.9 we could replace the command

```
System.out.println( theCount );
```

with the command

```
System.out.println( "You have clicked " + theCount + " times" );
```

You have to be a bit careful when using this feature. Basically, Java has two different ways of interpreting the + operator. When the operands to + are both numbers, Java performs normal, arithmetic addition. If, however, either of the operands to a + is textual rather than numeric, Java instead just sticks together the textual representations of both operands. This operation of sticking text together is called concatenation.

When Java sticks together bits of text, it doesn’t think about things like words. It just sticks the letters and digits it is given together. This means you have to include all the characters you want displayed, including any spaces. If you look carefully at the setText command shown above, you will notice that there is a space after the word clicked and before the quotation symbol that follows it and another space between the word times and the quote that precedes it. If these were not included, Java would display the text

```
You have clicked times
```

instead of displaying

```
You have clicked 3 times
```

as desired.

It is also important to be aware of how Java decides when a + means addition and when it simply means to stick pieces of text together. For example, if the value of theCount is 0,
then the command

```java
countDisplay.setText("You have clicked "+(theCount + 1)+" times");
```

will display the message

You have clicked 11 times

on the screen, while the command

```java
countDisplay.setText("You have clicked "+theCount+1+" times");
```

will produce the message

You have clicked 101 times

This is because, in the version with parentheses around "theCount + 1", Java has to do the operation within these parentheses first. Both operands of this + operator are numbers, so Java does addition yielding the number 11. Without the parentheses, Java processes the + operators in order from left to right. The first operand to the first + is a quoted string, so Java performs concatenation sticking the textual representation of the value of theCount, "0", together with the quoted text The result of this first operation is then treated as the first operand of the next + operation. Since this first operand is text, Java now interprets the second + as another concatenation operator as just sticks a "1" on the end of the text rather than performing a numeric addition.

**Exercise 3.8.1**

What output would you expect from the following statements? Assume count is 34.

- a. System.out.println("The count is: "+count-3);
- b. System.out.println("The count is: "+(count+2));
- c. System.out.println("The count is: "+count+4);

**Exercise 3.8.2**

Suppose you are trying to write a simple program to help someone practice the multiplication tables. The program repeatedly displays a message of the form:

```
What is 5 x 97?
```

in a `Text` object named `question` that is created in the program’s `begin` method. The program uses two variables named `factor1` and `factor2` that are assigned randomly generated numbers between 0 and 9 to determine which question to display. For example, the question shown above would be displayed if `factor1` equaled 5 and `factor2` equaled 9.

Write the statement needed to update the `question` displayed by `question` after new value for `factor1` and `factor2` are chosen. (Don’t worry about how the numbers are chosen or how the user tells the program the correct answer.)

---

**Section 3.9 Random Numbers**

"Pick a number. Any number."

You might expect to hear this phrase from the hawker at a carnival game table. You might not expect it to be a useful instruction to give a computer within a Java program, but just the opposite is true. There are many programming contexts in which it is handy to be able to ask the computer to pick a random number for you. Obvious examples are game programs. Programs that deal with cards, simulate the tossing of dice, or simulate the spinning of a roulette wheel all need ways of picking items randomly. Many programs that simulate the behavior of real systems for practical purposes need ways to incorporate the randomness of the real world in their calculations. With this in mind, Java and most other programming systems include what are called random number generators.

In our library, we have incorporated two classes designed to make it quite easy to obtain a sequence of random values in a program. One of our classes is designed for situations where you need random integers and the other for random real numbers.

Suppose that you wanted to write a program to simulate a board game in which at each turn the player rolls two dice. Our class for generating random integers can be used to create a Java object that behaves just like a single die. To illustrate the use of this class, we will construct a simple program that simulates the rolling of a pair of dice each time the mouse is clicked.

In our library, the class of random integer generators is named `RandomIntGenerator`. Like other objects, the first step in using one of our `RandomIntGenerators` is to define a variable name that will refer to the object. We might define a variable like

```java
private RandomIntGenerator die;
```

to refer to the random number generator in a program that simulated dice.

When we construct a new `RandomIntGenerator` we must provide two int values as parameters. These values determine the range of values that might be produced by the `RandomIntGenerator` created. Since a single die must show a number between 1 and 6 and we want our random number generator to simulate a single die, we would say

```java
new RandomIntGenerator(1, 6);
```

In general, the first parameter value determines the smallest value that should ever be produced by the random number generator, while the second number specifies the largest value. We could include this construction in our program’s `begin` method or as an initializer in the variable declaration. As a matter of good style, we would define a constant

```java
private static final int NUM_SIDES = 6;
```

so that we could replace the literal 6 with a name that suggests its significance. The declaration that creates our random number generator would therefore look like

```java
private RandomIntGenerator die = new RandomIntGenerator(1, NUM_SIDES);
```

The English word for the little cubes you roll while playing many board games has an irregular plural form. If you have several of these cubes, you call them dice. If you have just one, then it is a die.
Chapter 3 Working with Numbers

### 3.10 Summary

A program to simulate the rolling of a pair of dice.

```java
public class RollAnotherOne extends WindowController {
    // Coordinates to determine positions of text displayed
    private static final int TEXT_X = 30;
    private static final int PROMPT_Y = 30;
    private static final int RESULT_Y = 100;

    // How many sides our dice have
    private static final int NUM_SIDES = 6;

    // The object that represents a single die
    private RandomIntGenerator die = new RandomIntGenerator(1, NUM_SIDES);

    // A Text message updated to describe each simulated roll
    private Text result;

    // Value of each die on a given roll
    private int roll1;
    private int roll2;

    // Display a prompt and create the Text used to display the results
    public void begin() {
        new Text("Click to make me roll the dice", TEXT_X, PROMPT_Y, canvas);
        result = new Text("", TEXT_X, RESULT_Y, canvas);
    }

    // Roll the dice with each click
    public void onMouseClick(Location point) {
        roll1 = die.nextIntValue();
        roll2 = die.nextIntValue();

        result.setText("You rolled a " + roll1 + " and a " + roll2 + " for a total of " + (roll1 + roll2));
    }
}
```

![Figure 3.11](image)

Figure 3.11 Sample message drawn by dice simulation program

Now, when the user clicks the mouse, we need to tell the object named die to pick a random number for us. In fact, if we want to simulate the rolling of a pair of dice we will have to do this twice. We can ask a RandomIntGenerator to pick a number by invoking its `nextIntValue` method. That is, an expression like

die.nextIntValue()

will produce a (possibly different) random number each time it is evaluated.

The complete code of a simple program to simulate rolling two dice is shown in Figure 3.11. A sample of the program's output is shown in Figure 3.11. Note that even though the program simulates the rolling of two dice, it only uses a single random number generator named `die`. As long as we have created a single random number generator that generates values in the desired range, we can (and should) use it over and over again whenever we need a random number select from that range. In the example, we therefore use the `nextIntValue` method of `die` to determine the values seen on the first die (`roll1`) and the second die (`roll2`).

#### Exercise 3.9.1

Write a program that draws a rectangle named `box` at the location (50, 50). The box should be 50 pixels wide and its height should be determined by a `RandomIntGenerator` that generates values for the height between 10 and 100 pixels. The height of box should change each time the mouse is clicked.

Early applications of computers involved scientific and engineering problems that required large amounts of numerical computation. The first computers clearly earned the name "computer". When using a word processor, reading your email or browsing the Web, it is easy to forget that computers perform arithmetic computations. Even in programs that do not appear to involve numbers, however, numerical computations continue to play a role. For example, a word processor has to do arithmetic just to determine how many words will fit on a line.

In this chapter, we have explored a few of the mechanisms Java provides to perform numerical computations. At this point, we have only seen how to work with integer values in a program. We will introduce the use of Java's version of real numbers, doubles, in Chapter 5.
Chapter 3 Working with Numbers

We explored the differences between statements and expressions in Java programs. Statements are phrases in a program that instruct the computer to perform an action that will change the state of the computer or change the value associated with some variable name. Expressions are phrases that describe a value or object to be used in the program. We have seen that Java recognizes several forms of expressions: literals like "12", variable names, constructions, invocations of accessor methods, and arithmetic formulas involving the operations of addition, subtraction, multiplication and division.

We showed how to instruct Java to produce textual output that included numerical information using both Text objects and the System.out.println method. Text objects are used when such information is to be included in the display of a program that also produces graphic output. System.out.println provides a simpler mechanism for producing text output that is appropriate for programs that only produce textual output.

3.11 Chapter Review Problems

Exercise 3.11.1
What is the difference between an accessor method and a mutator method? Give an example of each from the Location class.

Exercise 3.11.2
a. Write the instructions for creating and positioning a 50-by-50-pixel FramedRect so that it is centered in any window.
b. Write the instructions for creating a Text object, that says "I am centered" and positions it so that it is centered in any window.

Exercise 3.11.3
Why is the following a bad instance variable declaration?

```java
private Line edge = new Line (0, 0, 100, 100, canvas);
```

Exercise 3.11.4
Write a program called RisingMoon which is similar to RisingSun except that we have crescent moon instead of a sun and the sky is now black. Each time the user clicks, the crescent moon rises slightly.

Hint: a crescent can be created by displaying an oval with the same color as the background but another oval.

3.12 Programming Problems

Exercise 3.12.1
Take the ICanCount program one step further by writing a program called ICanCountALot. It will keep track of the number of mouse clicks, mouse exits and button presses. The display of the program should look similar to the one in Figure 3.13. Don’t forget to use constants!
Exercise 3.12.2

Write a program called GrowMan which initially draws a little man as shown in Figure 3.14. Each time the user clicks, the man grows by a set amount. After ten more clicks, the display should look like that in Figure 3.15. The following constants and instance variable declarations are given to you.

```java
public class GrowMan extends WindowController {

    // Amount each body part grows by (should be even)
    private static final int GROW = 2;

    // Initial size of head
    private static final int HEAD_SIZE = 6;

    private static final int LIMB_SIZE = 5; // Initial displacement
    // of ends of limbs from
    // body, both horizontally
    // and vertically

    private static final int HEAD_START = 50; // x and y coordinate of
    // initial starting point

    // Coordinates of body parts
    private static final int BODY_X = HEAD_START + HEAD_SIZE / 2;
    private static final int NECK_Y = HEAD_START + HEAD_SIZE;
    private static final int ARMPIT_Y = HEAD_START + 2 * HEAD_SIZE;
    private static final int BODY_END = HEAD_START + 3 * HEAD_SIZE;
    private static final int FEET_Y = BODY_END + LIMB_SIZE;

    // Instance variables
    private FrameEoval head;
    private Line body,
        leftArm,
        rightArm,
        leftLeg,
        rightLeg;
```
Chapter 4

Making Choices

To write interesting programs we must have a way to make choices in Java. For example, we might need to perform different instructions depending on whether a user has clicked inside a particular rectangle. In this chapter we show how to make choices in Java using the if statement.

Conditional statements like the if statement are programming constructs capable of choosing between blocks of code to execute. These statements provide enormous expressive power for programmers, and yet are very easy to use and understand because they mimic the way we think.

For instance, the following forms a conditional statement in English:

"If it's sunny outside then we will play frisbee, otherwise we will play cards."

An example of a conditional statement a bit more relevant to our concerns with programme might be:

"If the mouse location is contained in the rectangle then display message "success". Otherwise display message "missed"."

In Java, the if statement is the most commonly used conditional statement. It comes in a variety of forms that we will explore, each useful for certain situations. With the help of conditionals we will write programs to determine a win or loss in the game of craps and to figure out what to do on a weekend based on the weather and your finances.

After presenting a brief example illustrating the use of the if statement, we formalise several of its variations that will allow us to handle more complex situations. We also introduce the boolean data type for expressions that can be either true or false. Finally we provide some advice on how to use conditionals clearly and effectively so that your program can be understood correctly by the Java compiler and, more importantly, by other programmers.
public class Voting extends WindowController {

// Coordinates of canvas, including x-coord of middle
private static final int MID_X = 300;
private static final int TOP = 0;
private static final int BOTTOM = 400;

// x coordinates of A and B text messages
private static final int TEXT_A_X = 20;
private static final int TEXT_B_X = MID_X + 20;

// y coordinates of instructions and vote count info
private static final int INSTRUCTION_Y = 380;
private static final int DISPLAY_Y = 220;

private int countA = 0; // Number of votes for A
private int countB = 0; // Number of votes for B
private Text infoA; // Display of votes for A
private Text infoB; // Display of votes for B

// Create displays with instructions on how to vote
public void begin() {
    new Text("Click on the left side to vote for candidate A.",
     TEXT_A_X, INSTRUCTION_Y, canvas);
    new Text("Click on the right side to vote for candidate B.",
     TEXT_B_X, INSTRUCTION_Y, canvas);
    infoA = new Text("So far there are " + countA +
     " vote(s) for A.", TEXT_A_X, DISPLAY_Y, canvas);
    infoB = new Text("So far there are " + countB +
     " vote(s) for B.", TEXT_B_X, DISPLAY_Y, canvas);
    new Line(MID_X, TOP, MID_X, BOTTOM, canvas);
}

// Update votes and display vote counts
public void onMouseClick(Location point) {
    if (point.getX() < MID_X) {
        countA++;
        infoA.setText("So far there are " + countA +
         " vote(s) for A.");
    }
    else {
        countB++;
        infoB.setText("So far there are " + countB +
         " vote(s) for B.");
    }
}

Figure 4.2 Code Voting class

Section 4.2 The if Statement

The if statement allows the programmer to make choices about which statements are executed in a program based on a condition, an expression whose value is either true or false. In the sample program, when the mouse is clicked the program must determine whether to give a vote to candidate A or B. The condition is whether the x coordinate of the mouse click, obtained by evaluating point.getX(), is less than MID_X. The condition is written in Java as point.getX() < MID_X. If the condition is not satisfied (i.e., if point.getX() is greater than or equal to MID_X) then the code after the else keyword is executed.

The two lines of code following the line containing the if are grouped together by a pair of matching curly braces. A sequence of statements surrounded by curly braces in this manner is called a block. Similarly, the two statements immediately following the else also form a block. If the condition of an if statement is true, the block of statements immediately after the condition is executed. Otherwise, the block immediately after the else is executed. Recall that the statement ++ indicates that the value of variable x should be increased by 1.

Exercise 4.1.1

What are the conditions in the following statements? For example, in the statement, "When it is cold outside, I wear a hat," the condition is "it is cold outside".

a. When it is Tuesday, I have piano lessons.
b. Since today is Halloween, it must be October.
c. If Bobby says yes, we will go to the prom.
d. Yesterday's game determined the wild card, so if the Red Sox won, they made it to the playoffs.

4.2 The if Statement

Now that we have seen the if statement in action, let's carefully examine its syntax and meaning. The code in the Voting class example given in Figure 4.2 contains a form of conditional statement called the if-else statement. Its syntax is:

if (condition) {
    if-part // Statements to be executed when condition is true
} else {
    else-part // Statements to be executed when condition is false
}

The text condition in the syntax template represents an expression whose value is true or false. The phrases if-part and else-part represent sequences of Java statements. We've included comments to make it clear when each of these sequences of statements is executed, even though these comments are not part of the formal syntax.

When an if-else statement is executed, the computer first determines whether condition is true. If so, it executes the statements in the block of code surrounded by the first pair of curly braces.
Chapter 4 Making Choices

Section 4.2 The if Statement

There are many situations in which we don’t need an else-part. Fortunately, there is a simple variant of the if-else statement, the if statement, which does not have the else keyword or the else-part.

```java
if (condition) {
    if-part // Statements executed when condition is true
}
else {
    else-part // Statements executed when condition is false
}
```

If condition is true, then the if-part is executed as before. If it is false, however, the program simply moves on to the next statement after the if-part because there is no else-part to execute. The execution sequence is illustrated in Figure 4.5.

Exercise 4.2.1
Write the following statements as if-else or if statements. For example, the statement, “On Mondays I ride my bike to class. On other days, I walk.” can be rewritten as “If it is Monday, I ride my bike to class. Else, I walk.”

a. On Sunday I eat pancakes. On other days, I eat cereal.
b. I go to class on weekdays, but on weekends, I watch movies.
c. In the summer I always wear sandals, but during the other seasons, I wear sneakers.
d. If it is raining, I bring an umbrella.

4.2.1 Examples: Using the if Statement with 2-D Objects

Suppose we want to write a program that begins by displaying a square on the canvas. If the user clicks inside the square, then the computer moves the square 50 pixels to the right. If the click is not inside the square, nothing happens.

Figure 4.3 Semantics of the if-else statement

Figure 4.4 Code to display individual and total vote counts

braces, "([" and "]"), called the if-part, and then skips over the rest of the statement. Otherwise, i.e., if condition is false, it skips over the if-part and will instead execute the block of statements after the else keyword, called the else-part. Exactly one of the two blocks of code is present when the if-else statement is executed. When that block is completed, execution resumes immediately after the if-else statement. This execution sequence is illustrated in Figure 4.3.

The following example shows how execution resumes with the statements that follow the if-else statement. Suppose we want to modify our voting program so that it always displays the total number of votes. Let infoTotal be a variable of type Text that has been initialized in the begin method. Figure 4.4 shows a revised version of onMouseClick that displays the current vote total. Each time the user clicks on the canvas, the line sending the setText needs to infoTotal will be executed regardless of which half of the screen the mouse was clicked on.
Chapter 4  Making Choices

How can we determine if a point is inside a square? We could compare the coordinates of the point to the locations of the left, right, top, and bottom edges of the square. However, such tests are so common that all of the two-dimensional geometric objects (e.g., FramedRect, FilledRect, Framed Oval, FilledOval) provide the accessor method contains that does this for us.

For example, let square be a variable of type FramedRect, and let point be a variable of type Location. Then the expression square.contains( point ) evaluates to true if the object held in square contains the point, and false otherwise.

For the sake of brevity we will only write out the onMouseClick method of this program. We assume that square has been declared elsewhere to be a variable of type FramedRect and that it has been initialized in the begin method.

```java
public void onMouseClick( Location point ) {
    if ( square.contains( point ) ) {
        square.move( X.OFFSET, 0 );
    }
}
```

An English translation of the above method would read: "If the square contains the point when the mouse was clicked then tell the square to move to the right by OFFSET pixels. Otherwise do nothing." (Of course, the computer code doesn't say, "Do nothing:" Instead it simply returns.)

We will present further variations of if statements later in this chapter, but first we will explore the kinds of expressions that can be used to form the condition part of these statements.

**Exercise 4.2.2**

a. What would happen if you clicked the mouse inside square given the onMouseClick method shown below? Assume X.OFFSET is 60.

```java
public void onMouseClick( Location point ) {
    if ( square.contains( point ) ) {
        square.move( X.OFFSET, 0 );
    } else {
        square.move( -X.OFFSET, 0 );
    }
}
```

b. What would happen if you now clicked the mouse outside of square and onMouseClick was changed to the following?

```java
public void onMouseClick( Location point ) {
    if ( square.contains( point ) ) {
        square.move( X.OFFSET, 0 );
    } else {
        square.move( -X.OFFSET/2, 0 );
    }
}
```

Section 4.3  Understanding Conditions

### 4.3.1  The boolean Data Type

Java contains a data type called boolean. Unlike the int type, which has a large number of elements, the boolean type has only two values: true and false. Just as one can write down integer values directly as 37, -158, or 47, we can write down boolean values directly in Java as true or false. We can also declare variables of type boolean.

There are a large number of expressions in Java that return values of type boolean. As we have just seen, combining two integer-valued expressions with one of the comparison operators, <, >, <=, >=, ==, and !=, results in a value of type boolean. We have also seen the method contains that returns a value of type boolean.

If ok is a variable of type boolean and x has type int, then the following are valid statements:

```java
ok = true;
ok = ( x > 3 );
```
public class WhatADrag extends WindowController {
    ...
    // Constant declarations omitted

    private FilledRect box; // Box to be cragged
    private Location lastPoint; // Point where mouse was last seen

    // Whether the box has been grabbed by the mouse
    private boolean boxGrabbed;

    // Make the box
    public void begin() {
        box = new FilledRect(START_LEFT, START_TOP, 
                             BOX_WIDTH, BOX_HEIGHT, canvas);
    }

    // Save starting point and whether point was in box
    public void onMousePress(Location point) {
        lastPoint = point;
        boxGrabbed = box.contains(point);
    }

    // If mouse is in box, then drag the box
    public void onMouseDrag(Location point) {
        if (boxGrabbed) {
            box.move(point.getX() - lastPoint.getX(), 
                      point.getY() - lastPoint.getY());
            lastPoint = point;
        }
    }
}

Figure 4.6: Code for dragging a box

In each case the expression of the right-hand side evaluates to a boolean value and hence can be assigned to a variable of type boolean. We will use both the contains method and boolean variables in the implementation of new class WhatADrag. The complete code listing for this program is given in Figure 4.6. The program begins by displaying a box on the screen. If the user presses the mouse down while it is pointing inside of the box, then drags the mouse, the box will follow the mouse on the canvas. If the mouse is not pointing in the box when the mouse button is pressed, then dragging the mouse should have no effect, even if the mouse happens to cross the box at some point during the drag.

Let's look at the code in the mouse-handling methods to see how we can program this behavior. As soon as the mouse button is pressed, the onMousePress method is executed. The first assignment in onMousePress,

    lastPoint = point;

results in saving the current location of the mouse (as held in parameter point) as the value of variable lastPoint. The location is saved so that it can be used when onMouseDrag is executed later. The second assignment in onMousePress,

    boxGrabbed = box.contains(point);

determines and then remembers whether the box contained the point where the mouse was initially pressed. Because the result of evaluating box.contains(point) is a value of type boolean, it can be stored in boxGrabbed, a variable of type boolean. The value of boxGrabbed, which reflects whether the user actually pressed the mouse down inside box, will be used in onMouseDrag to determine whether the box should be moved. As usual, the onMouseDrag method will be executed when the user drags the mouse. At that time the values of the variables boxGrabbed and lastPoint become relevant. If the value of boxGrabbed is true, box will be moved by the distance between the last location of the mouse, saved in lastPoint, and the current position, held in point. The distance in each of the horizontal and vertical directions is needed to actually perform the move. The horizontal and vertical distances are obtained by evaluating point.getX() - lastPoint.getX() and point.getY() - lastPoint.getY(), respectively.

Figure 4.7 illustrates three stages of dragging a box in class WhatADrag. In the leftmost picture, the mouse button has been pressed with the mouse inside the rectangle. After the execution of onMousePress, the location of the mouse is stored in the instance variable lastPoint. In the middle picture, the mouse has been dragged down and to the right, and onMouseDrag has just begun execution. The current location of the mouse is held in the parameter point, but the if statement has not yet been executed. The rightmost picture shows what has happened immediately after the move message has been sent to box in the if statement during the execution of onMouseDrag. The rectangle has been dragged to the right and down by the difference between the x coordinates and by the difference between the y coordinates of point and lastPoint. The update of the value of lastPoint to the location held in point is not shown.
As a simple example, consider how to assign letter grades based on numeric scores on an examination. Scores greater than or equal to 90 are assigned an "A," scores from 80 to 89 are assigned a "B," scores from 70 to 79 are assigned a "C," and those below 70 are assigned "no credit." Suppose that a variable score of type int contains the numeric score of a particular examination. We would like to display the appropriate letter grade in a Text item, gradeDisplay.

In this situation we have not just two, but four different possibilities to worry about, so it is clear that a simple if-else statement is insufficient.

Java allows us to extend the if-else statement for more than two possibilities by including one or more else if clauses in an if statement. Thus we can display the appropriate grade using the following statement:

```java
if ( score >= 90 ) {
    gradeDisplay.setText( "The grade is A" );
} else if ( score >= 80 ) {
    gradeDisplay.setText( "The grade is B" );
} else if ( score >= 70 ) {
    gradeDisplay.setText( "The grade is C" );
} else {
    gradeDisplay.setText( "No credit is given" );
}
```

When we execute this if statement, the computer first evaluates the boolean expression score >= 90. If that is true, an "A" will be displayed and execution will continue after the last else-part of the statement. However, if it is false, the program will evaluate the next boolean expression, score >= 80. If that is true, a grade of "B" will be displayed and execution will continue after the last else-part. If not, the expression score >= 70 will be evaluated. If that is true, then a grade of "C" will be displayed and execution will continue after the else-part. Otherwise, the statement in the else-part will be executed and a grade of "no credit" will be displayed.

When checking to see if the student should be given a "B," why didn't we also have to check whether score < 90? The reason is that to get to the condition score >= 80, the previous test, score >= 90, must have already failed. That is, we only execute the else clauses if score is less than 90. The same reasoning shows that we do not need to check if score < 80 when we determine whether to give a "C" as the grade. We can always count on the fact that the conditions within the if-else statements are evaluated sequentially, and therefore that all previous tests in the if statement have failed, before determining whether to execute the next block.

We can summarize the execution of an if statement including else if's as follows:

1. Evaluate the conditions after the if's in order until one is found to be true.
2. Execute the statements in the block following that if and then resume execution with the first statement after the entire if statement.
If none of the conditions is true and there is an else-part, then execute the statements
in the else-part. If there is no else-part, don't execute any of the statements in the
if statement.

Finally, continue execution with the first statement after the if statement.

If there is an else clause in an if statement, then it must be the very last part of the if.
There is no further else if's are allowed after a plain else clause.

We will see later that this more complex if statement is actually a special case of a more
general way of nesting if statements. However, it is convenient for the moment to consider it
by itself.

Exercise 4.4.1

What is the output when the following pieces of code are executed? Why does the output of a
two pieces of code differ? Assume hamburgerPrice is 7.

```java
a. if ( hamburgerPrice < 2 )
    System.out.println( "This hamburger is super cheap!" );
else if ( hamburgerPrice < 4 )
    System.out.println( "This hamburger is cheap." );
else if ( hamburgerPrice < 6 )
    System.out.println( "This hamburger is fairly cheap." );
else if ( hamburgerPrice < 10 )
    System.out.println( "This hamburger is pricey!" );
else
    System.out.println( "This hamburger is super expensive!" );

b. if ( hamburgerPrice >= 10 )
    System.out.println( "This hamburger is super expensive!" );
else if ( hamburgerPrice < 10 )
    System.out.println( "This hamburger is pricey!" );
else if ( hamburgerPrice < 6 )
    System.out.println( "This hamburger is fairly cheap." );
else if ( hamburgerPrice < 8 )
    System.out.println( "This hamburger is moderate." );
else
    System.out.println( "This hamburger is super cheap!" );
```

4.5 More on Boolean Expressions

The if-else statement in the Voting class at the beginning of this chapter was sufficient because
there were only two candidates to consider when assigning a new vote. However, for more than
two candidates the else if clause introduced in the last section becomes necessary.

In the next example our program must choose among three candidates, A, B, and C, of whom
each has been allocated a vertical third of the canvas. We will not rewrite the entire program here,
but will instead focus on the onMouseClick method.

Let LEFT_SEPARATOR and RIGHT_SEPARATOR be constants representing the x coordinates of
the vertical lines that divide the canvas into the three pieces. The int variables countA, countB,
and countC will keep track of the number of votes for the three candidates. Here is the code:

```java
// Update votes and display vote counts for three candidates
public void onMouseClick( Location point ) { // Clicked on left section
    if ( point.getX() < LEFT_SEPARATOR ) { // Clicked in left section
        countA++;
        infoA.setText( "So far there are " + countA + " votes for A."");
    }
    else if ( point.getX() < RIGHT_SEPARATOR ) { // Clicked in center
        countB++;
        infoB.setText( "So far there are " + countB + " votes for B."");
    }
    else { // Clicked in right section
        countC++;
        infoC.setText( "So far there are " + countC + " votes for C." );
    }
```
When determining whether the click was in the center section, why didn’t we have to check that `point.getX()` was to the right of LEFT_SEPARATOR? As with our earlier example, the reason is that to even get to the second condition we know that the test `point.getX() < LEFT_SEPARATOR` must have failed (that is, it must have evaluated to false).

This same style solution works for determining whether clicks are in four or more vertical strips. However, once we get to four candidates, it might make sense to divide the screen both vertically and horizontally rather than into four narrow, vertical strips, as shown in Figure 4.6. We will count clicks in the upper left-hand corner as votes for A, in the upper right-hand corner as votes for B, lower left for C, and lower right for D. Thus to be a vote for A, the location where the user clicked must be both above the horizontal line and to the left of the vertical line. Here we can write this as a condition:

In Java we use the `&&` operator, between two boolean expressions to indicate that both must be true for the entire expression to be true. For example, we can ensure that `x` is positive and `y` is negative by writing `x > 0 && y < 0`. While it would be convenient, computer programmatic languages do not usually allow us to combine two inequalities, as in the expression `1 <= x < 10`. Instead we must write it out as `1 <= x && x < 10`.

Just as Java uses `&` to represent the logical and operator, it uses `||` to represent the logical or operator. Thus `x < 5 || y > 20` will be true if `x < 5` or `y > 20`. In general, if `b1` and `b2` are boolean expressions, then `b1 || b2` is true if one or both of `b1` and `b2` are true. English contains both an inclusive and an exclusive "or". The inclusive "or" evaluates to true if either or both operands are true. The exclusive "or" evaluates to true only if exactly one of the operands is true. The `|` operator represents the inclusive "or". Using the "not" operator, `!`, defined below, you can get the effect of the exclusive "or" on boolean expressions `b1` and `b2` by writing `(b1 || b2) && (!b1 && !b2)`, which states that one of `b1` and `b2` is true, but not both.
A good example illustrating the use of the or operation is determining whether someone playing the game of craps has won or lost after his or her first roll of the dice. The "shooter" in craps throws two dice. If the numbers on the faces of the dice add up to 7 or 11, then the shooter wins a sum of 2, 3, or 12 results in an immediate loss. With any other result, play continues in a way that will be described later.

The if statement below has three branches that encode the relevant outcomes of the first throw of the dice. The value of status is simply a Text object that, as usual, has been created in a method of the program.

```java
if ( roll == 7 || roll == 11 ) { // 7 or 11 wins on first throw
    status.setText( "You win!" );
} else if ( roll == 2 || roll == 3 || roll == 12 ) { // 2, 3 or 12 loses
    status.setText( "You lose!" );
} else { // Play must continue
    status.setText( "The game continues" );
}
```

The `if` portion determines whether the player has won by checking if the roll was 7 or 11. The `else if` portion determines whether the player has lost by checking if the roll was 2, 3, or 12. Finally, the `else` portion is executed if further rolls of the dice will be required to determine whether the player wins or loses.

The last boolean operator to be introduced here is 1, which stands for "not." For example, if `expression box.contains(point)` will be true exactly when `box.contains(point)` is false, i.e., when `box.contains(point)` is not true.

Although we can use 1 with equations and inequalities, it is usually clearer to rewrite the statement using a different operator. For instance, `!(x < y)` is more simply written as `x >= y`. Similarly, `!(x <= y)` is equivalent to `x > y`, which is much easier to read.

Figure 4.9 summarizes the most common operators in Java that give a boolean result.

<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>not</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
</tbody>
</table>

Section 4.6 Nested Conditionals

Exercises 4.5.1

Suppose that the current value of `x` is 6, `y` is –2, and `z` is 13. For each of the following conditions, determine whether `expression` evaluates to `true` or `false`.

| a. `x - 6 < y && z == 2 * x + 1` |
| b. `!(x - 6 < y && z == 2 * x + 1)` |
| c. `x - 6 < y || z == 2 * x + 1` |
| d. `!(x - 6 < y || z == 2 * x + 1)` |

Before we move on, we should note a few last points about && and ||. The first is to be sure and use the double version of each of the symbols && and ||, because the single versions represent slightly different operators.

Second, both && and || in Java are implemented as "short-circuit" operations. What this means is that Java will cease evaluating an expression involving one of these operators as soon as it can determine whether the entire expression is true or false.

For example, suppose a program includes a declaration of an int variable, `x`, and that it contains the expression `(x > 10) && (x <= 20)`. If the computer evaluates that expression when `x` has value 3, it will only evaluate `x > 10` without even considering `x <= 20`. Because `x > 10` is false, the expression will first evaluate to `false && (x <= 20)` As a result, Java can determine that the final value of the && expression must be false no matter what the value of `x` is. However, if `x > 10` had been true, the rest of the expression would have to be evaluated to determine whether the entire && expression evaluates to true or false.

While you are unlikely to care much in this case whether the second argument is evaluated, there are other cases in which evaluating the second argument may result in an error. For example, if the boolean expression `x != 0 && 3/x > 17` were not evaluated in this short-circuit fashion, then if `x` were 0 at run time, a run-time error would result when `3/x` was evaluated.

Expressions involving the `||` operator are also evaluated in a short-circuit fashion. If the left side evaluates to true, then Java knows that the entire `||` expression must evaluate to true, so it does not bother to evaluate the right side. Conversely, if the left side is false, then the right side must be evaluated in order to determine the final value of the entire `||` expression.

4.6 Nested Conditionals

Occasionally we run into problems that would require complex boolean conditions if they were handled using if or else if statements as we have seen them used so far. Rather than constructing these complex conditions, we will introduce alternative structures for supporting the program logic. Happily, we don’t need to introduce any more syntax in order to handle them; we just need to combine if statements in different ways.
Suppose it is a summer weekend and you are trying to figure out what to do. Your choice of recreation will depend on the weather and how much money you have to spend. The following table lists the various options and choices, where the row headings represent your possible financial situation and the column headings represent the weather possibilities.

<table>
<thead>
<tr>
<th></th>
<th>sunny</th>
<th>not sunny</th>
</tr>
</thead>
<tbody>
<tr>
<td>rich</td>
<td>outdoor concert</td>
<td>ultimate frisbee</td>
</tr>
<tr>
<td>not rich</td>
<td>indoor concert</td>
<td>watch TV</td>
</tr>
</tbody>
</table>

The table entries represent the suggested recreational activity, given the financial situation represented by the row and the weather as represented by the column. Thus if you are feeling rich and it is not sunny, you might want to go to an outdoor concert. If you are not feeling rich and it is sunny, you might play some ultimate frisbee.

How can we represent these choices with an if statement? Let rich and sunny be variables of type boolean, and let activityDisplay be a variable of type Text that will display the selected activity. The if statement below uses else if clauses to represent the four choices.

```java
if (sunny && rich) {
    activityDisplay.setText("outdoor concert");
} else if (!sunny && rich) {
    activityDisplay.setText("indoor concert");
} else if (sunny && !rich) {
    activityDisplay.setText("ultimate frisbee");
} else {
    // Not sunny and not rich
    activityDisplay.setText("watch TV");
}
```

As we will discuss in more detail in the next chapter, ! has higher precedence than &&. This means that the not operator, !, will always be applied before the && operator. Thus the condition !sunny && !rich is evaluated by first evaluating !sunny and then using the && operator to determine whether both !sunny and !rich are true. Similarly, the || operator has higher precedence than !. This code correctly represents all four options, but is rather verbose and loses the nice structure of the table. A related problem is that by the time we arrive at the last case the program has evaluated three fairly complex boolean expressions.

We can write this so that only two evaluations of boolean variables are ever made, and furthermore they are made without the added complication of negation or and operators. This

Section 4.6 Nested Conditionals

accomplished by nesting if statements. A nested if statement involves including one or more if statements inside another, as in the example below.

```java
if (sunny) {
    if (rich) {
        activityDisplay.setText("outdoor concert");
    } else if (!rich) {
        activityDisplay.setText("ultimate frisbee");
    } else if (!sunny) {
        activityDisplay.setText("indoor concert");
    } else {
        activityDisplay.setText("watch TV");
    }
}
```

The advantage to using these nested if statements is that the organization is quite similar to that of the table. There is an outer if-else statement that determines whether sunny is true. This corresponds to choosing either the first or second column of the table. Inside the outer if-else there is an if-else statement that determines whether rich is true. This corresponds to figuring out which row of the table applies. For example, if rich is false, the outcome should correspond to the first column and second row of the table, and hence the activity should be "play ultimate". The else-part corresponding to it not being sunny is handled in a similar fashion.

**Exercise 4.6.1**

Use nested conditionals to rewrite the onMouseClick method for tabulating votes of four candidates from Section 4.5.

**Style Note:** The nested if-else statements are indented from the outer ones in order to make the code easier to read and understand. Although Java compilers ignore the layout of code, human readers appreciate the cues of formatting in unindented code like this. Many Java development environments include an option to "indented" the code. Selecting this option usually results in more readable code.

Aside from the indenting, another thing that makes this code easy to understand is the inclusion of comments. In particular, notice how each else clause includes comments indicating under which conditions the else part is executed. This has the advantage of making it absolutely clear to the reader under what circumstances this code is executed. The more complete the conditional, the more important these comments become. We strongly urge all programmers to include such comments.

While the use of nested if-else statements yields code that is not quite as compact and simple to understand as the table, it is much easier to see its correspondence to the table than the version involving only else if clauses. Notice in particular that no matter what the values of sunny and rich are, only two boolean variables are ever evaluated during the execution of this code, so it is not only cleaner, but it is faster as well.
Exercise 4.6.2

Using the information provided in the table and nested conditionals, write an if statement that displays which course a student should take based on their interests in math and writing. Let likesMath and likesWriting be variables of type boolean, and let course be a variable of type Text that will display the recommended course.

<table>
<thead>
<tr>
<th>likes math</th>
<th>likes writing</th>
<th>doesn't like math</th>
<th>doesn't like writing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Economics</td>
<td>Calculus</td>
<td>English</td>
<td>Psychology</td>
</tr>
</tbody>
</table>

In Figure 4.10 we provide another example of complex choices being represented by nested if-else statements. This class provides the code to simulate a complete game of craps. 10 rules of craps are as follows:

The shooter rolls a pair of dice. If the shooter rolls a 7 or 11, it is a win. If the shooter rolls a 2, 3, or 12, it is a loss. If the shooter rolls any other number, that number becomes the "point". To win, the shooter then must roll the "point" value again before rolling a 7. Otherwise it is a loss.

The program simulates a roll of the dice by using a random number generator every time it is called. In order to implement the rules given above, we must organize the game logic in a way that can be represented using if statements. Notice that the rules for winning are quite different depending on whether this is the player's first throw. For instance, if it is the first throw, then rolling a 7 results in a win, but if it is a second or subsequent throw, then results in a loss. Therefore we will organize the first level of conditional to determine whether it is the first throw.

In order to make such a choice, the Craps class in Figure 4.10 declares a boolean variable newGame, to remember whether this is the first throw of a new game.

The outer if statement in the method onMouseClick has the following structure:

```java
if (newGame) {
    // Starting a new game
    ...
} else {
    // Continuing trying to make the point
    ...
}
```

The if-part of this code is itself a nested if statement with three branches, each of which encodes the relevant actions to be taken based on the first roll of the dice. Recall that we saw...
simplified version of this example earlier in the chapter. The new code is reproduced below:

```java
if (roll == 7 || roll == 11) { // 7 or 11 wins on first throw
    status.setText("You win!");
} else if (roll == 2 || roll == 3 || roll == 12) {
    // 2, 3, or 12 loses
    status.setText("You lose!");
} else {
    // Set the roll to be the new point to be made
    status.setText("Try for your point!");
    point = roll;
    newGame = false; // No longer a new game
}
```

Rather than having a separate branch for each possible value of the roll of the dice, there are only three. These branches correspond to winning, losing, and establishing a point to be made on subsequent rolls. The variable newGame remains true in the first two branches, so it need not be updated. Only the third branch requires setting newGame to false.

Let us now examine the else-part of the outer if statement. Like the if-part, this else statement also has three branches, though the second and third conditions are quite different for those that handle the first roll:

```java
if (roll == 7) { // 7 loses when trying for point
    status.setText("You lose!");
    newGame = true; // Set to start new game
} else if (roll == point) { // Making the point wins!
    status.setText("You win!");
    newGame = true;
} else {
    // Keep trying
    status.setText("Keep trying for " + point + "...");
}
```

In this statement, both of the first two choices result in setting newGame back to true because they represent the end of a game with either a win or a loss. The third statement merely asks if player to continue rolling, so newGame remains false.

---

**Exercise 4.8.1**

Define the following Java terms:

a. negation
b. block
c. if
  d. condition

**Exercise 4.8.2**

How would you express the following as operators in Java?

a. greater than
b. or
c. equal
d. less than or equal
e. not
f. and
g. not less than

**Exercise 4.8.3**

The database at a doctor’s office stores information about each patient. For each piece of information listed below, decide whether or not it would be appropriate to store the information as a boolean.
Exercise 4.8.4

Fix the problem(s) in the following code:

```java
if (x = 5) {
    message.setText("You win!");
}
else if (x < 5) {
    message.setText("You lose!");
}
else if (x > 5) {
    message.setText("Try again!");
}
```

Exercise 4.8.5

What are the values of the following expressions if \( x = 8 \), \( y = -5 \) and \( z = 2 \)?

a. \( x + y = z \)
   \( 1 \) (\( 2 + x + 3 + y > z \))
   \( (x - z) + z < z - 2 * y \)
   \( y - z + x >= 0 \)
   \( x - 4 + z + 1 = y \)

Exercise 4.8.6

Why is the following code more complex than it needs to be? Simplify it using else if.

```java
if (score <= 100 && score >= 90) {
    gradeDisplay.setText("You got an A");
}
if (score < 90 && score >= 80) {
    gradeDisplay.setText("You got a B");
}
if (score < 80 && score >= 70) {
    gradeDisplay.setText("You got a C");
}
if (score < 70) {
    gradeDisplay.setText("You don't get a grade");
}
```

Exercise 4.8.7

What are the values of the following expressions if \( x = -2 \), \( y = -1 \) and \( z = 4 \)?

a. \( x + y < z \) \( \text{||} \) \( (4 + y + z > x \&\& x > y) \)
   b. \( (x + y < z) \text{||} (4 + y + z > x) \&\& x > y \)
   c. \( (x + y < z) \text{||} (4 + y + z > x \&\& x > y) \)
   d. \( (x + y < z) \text{||} (4 + y + z > x) \&\& !(x > y) \)
   e. \( (x + y < z) \text{||} (4 + y + z > x \&\& !(x > y)) \)

Exercise 4.8.8

Assume that \( x = 2 \), \( y = -3 \) and \( z = 5 \). What are the values of \( x \), \( y \) and \( z \) after the following code has been executed?

a. \( x = y + z \)
   \( y = z - y \)
   \( z = x - 2 * y \)

b. \( x = y + z \)
   \( y = z - y \)
   \( z = x - 2 * y \)

Exercise 4.8.9

Another variation of the game craps is to play what is called the "No Pass Line". Now rolling a 7 or 11 on the first roll loses and a 2, 3, or 12 wins. After the point is set, rolling a 7 wins, while
**Exercise 4.9.1**

Create the game invisibleGame to master your use of conditionals. The game will have the following features:

- When the game begins, it will create three “invisible” boxes. The boxes will all be square but of different sizes. One should be 30 pixels wide, the second one 45 pixels wide, and the third 80 pixels wide. These boxes should be created randomly anywhere on the screen.
- The user will click the mouse and try to hit the boxes when doing so.
- When the mouse exits the window, the user will be notified of his/her success. The output should look like Figure 4.11.
- When the mouse re-enters the window, all variables should be reset and the boxes moved to new random locations.

*Points are assigned as follows:
- 200 points: Hitting all three boxes
- 150 points: Hitting the small box and the medium box
- 125 points: Hitting the small box and the large box
- 110 points: Hitting the medium box and the large box
- 100 points: Hitting only the small box
- 75 points: Hitting only the medium box
- 50 points: Hitting only the large box
- −1 point: For each mouse click

If possible, modify the applet settings so that the canvas is 300 pixels wide and 300 pixels high.

**Exercise 4.9.2**

Write a simple game with three dice called Dicey, which may remind you of a more popular game called dice. When the user clicks the mouse, the three dice are “rolled” and the results are displayed on the screen. The display will also include whether the user rolled three of a kind, a pair, or nothing of particular interest. See Figure 4.12.