CHAPTER 1

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 to 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 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 a new program.

A program is a set of instructions that a computer follows. We can therefore learn a good bit 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 a language that it comprehends. Java is such a language. We will discuss the mechanics of actually 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 do to. 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 they 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 take too long to read. This, however, may have more to do with our impatience than 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 instructions 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.

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.
3. Multiply by 3.
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.
9. Subtract 6.
10. Find the letter whose position in the alphabet is equal to the number you have 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 that has no understanding of their purpose, 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
32 )13042144
    128
    ---
      242
      224
      ---
       181
       160
       ---
        214
        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 is harder to devise an algorithm to accomplish a given goal than it is to simply accomplish the goal.

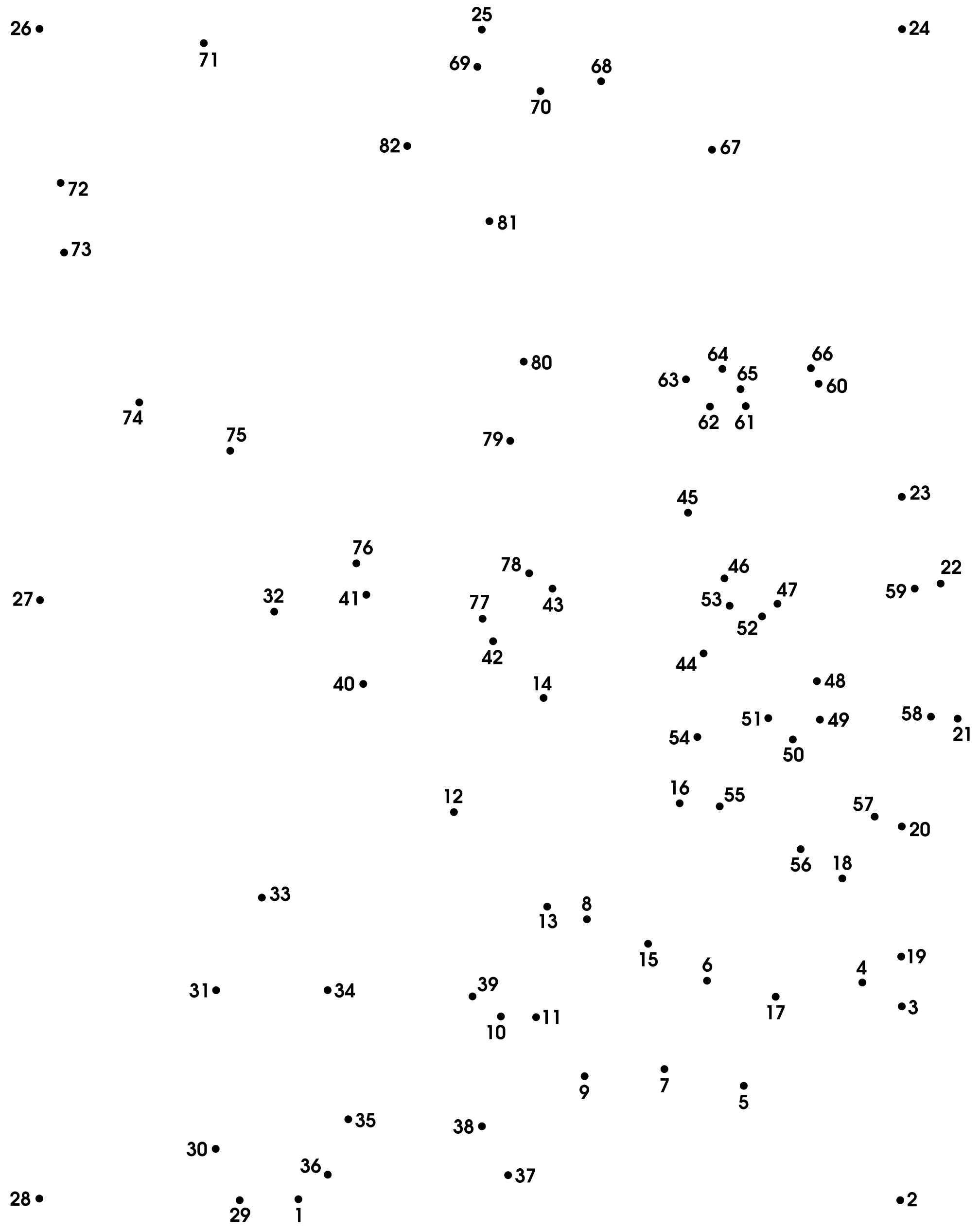


Figure 1.1 Connect dots 1 through 82 (©2000 MonkeyingAround.com)

The goal of the connect-the-dots puzzle shown in Figure 1.1 is to draw an elephant. In order to construct this puzzle, you first have to learn to draw an elephant without the help of the dots. Only after you have figured out how to draw an elephant in the first place will you be able to figure out where to place the dots and how to number them. Worse yet, figuring out how to place and number the dots so the desired picture can be drawn without ever having to lift your pencil from 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 (and you don't have a calculator handy) it is easier to do the division yourself than to try to teach the long-division algorithm to someone who doesn't already know it, so that he or she can solve the problem for you.

As you learn to program, you will see this pattern repeated frequently. Learning to convert your own knowledge of how to perform a task into a set of instructions so precise that they can be followed by a computer can be quite challenging. Developing this ability, however, is the key to learning how to program. Fortunately, you will find that as you acquire the ability to turn an informal understanding of how to solve a class of problems into a precise algorithm, you will be 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 communicated to other people or to a computer. Communicating an algorithm requires the use of a language. A program is just an algorithm expressed in a language that a computer can comprehend.

The choice of the language in which an algorithm is expressed is important. The numeric calculation puzzle that led you to think of Danish elephants was expressed in English. If our 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, . . . , LXXXII. Most of you probably understand Roman numerals, so you would still be able to complete the puzzle. You would probably have difficulty, however, if we switched to something more ancient like the numeric system used by the Babylonians or to something more modern like the binary system that most computers use internally, in which the 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 view because the language used is quite simple. Human languages like English and Japanese are very complex. It would be very difficult to build a computer that could understand a complete human language. Instead, computers are designed to interpret instructions written in simpler languages designed specifically for expressing algorithms intended for computers. Computer languages are much more expressive than a system for writing numbers like the Roman numerals, but much simpler in structure than human languages.

One consequence of the relative simplicity of computer languages is that it is possible to write a program to translate instructions written in one computer language into another computer language. These programs are called *compilers*. The internal circuitry of a computer usually can only interpret commands written in a single language. The existence of compilers, however, makes it possible to write programs for a single machine using many different languages. Suppose that you have to work with a computer that can understand instructions written in language A but you want to write a program for the machine in language B. All you have to do is find (or write) a 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 write 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 than others. This makes it easier to construct compilers that process programs written in these languages. 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 constructing 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 provides some sophisticated features that support an approach to programming called object-oriented 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 programming*. In this approach, programs are designed to react to *events* generated by the user or system. The programs that you are used to using on computers 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 approach to programming. Java is one of the first languages to make this easy to do as a standard part of the language.

1.3 Your First Sip 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 foreign 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 that 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 examples of grammatical structure. On the other hand, developing an extensive vocabulary without any knowledge of the grammar used to combine words would just be silly. The same applies to learning a programming language. Accordingly, we will begin your introduction to Java by presenting a few sample programs that illustrate fundamentals of the grammatical structure 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 the user can perform using the mouse and keyboard. Selecting menu items, typing in file names, 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 determined by the instructions that make up the program running on the computer. The examples presented 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 within the boundaries of a single window. When one of these programs is run, all that will appear on the 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 reveals features that are certainly consistent with our description of this program’s behavior. There 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 words “on mouse press” (all forced together to form the single word `onMousePress`) which

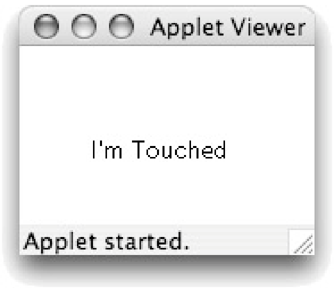


Figure 1.2 Window displayed by a very simple program

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

public class TouchyWindow extends WindowController {

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

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

}
```

Figure 1.3 Our first Java program

suggest when the new message will appear. Similarly, a little bit later, a line containing the word `onMouseRelease` is followed by a line containing the word `clear`, which is what happens to the window once the mouse is released. These suggestive tidbits are unfortunately obscured by a considerable amount of text that is probably indecipherable to the novice. Our goal is to guide you through the details of this program in a way that will enable you to understand its basic structure.

1.3.2 "Class" and Other Magic Words

Our brief example program contains many words that have special meaning to Java. Unfortunately, it is relatively hard to give a precise explanation of many of the terms used in Java to someone who is just beginning to program. For example, to fully appreciate the roles of the terms `import`, `public`, and `extends` one needs to appreciate the issues that arise when constructing programs that are orders of magnitude larger than we will discuss in the early chapters of this text. We will attempt here to give you some intuition regarding the purpose of these words. However, you may not be able to understand them completely until you learn more about Java. Until then, we can assure you that you will do fine if you are willing to regard just a few of these words and phrases as magical incantations that must be recited appropriately at certain points in your program. For example, the first two lines of nearly every program you read or write while studying this book will be identical to the first two lines in this example:

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

In fact, these two lines are so standard that we won't even show them in the examples beyond the first two chapters of this text.

The Head of the Class

Most of your programs will also contain a line very similar to the third line shown in our example:

```
public class TouchyWindow extends WindowController
```

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 computers, 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 running copies of the program form a collection of distinct but very similar objects. Java refers to such a collection of objects as a *class*.

Using Software Libraries

The class header of `TouchyWindow` indicates that it `extends` something called `Window-Controller`. 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 dot 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 by 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 words `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 that `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 a 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.

Getting Braces

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 consisting of an open 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 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 HiMom. ❖

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

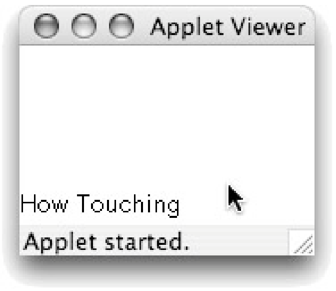


Figure 1.4 Changing the information displayed in the window

them you can display a different message or make the message appear in a different position on the screen. For example, if we replaced the body of our `onMousePress` method with the line:

```
new Text( "How Touching", 0, 80, canvas );
```

the program would display a different message in a different location as shown in Figure 1.4. Accordingly, you cannot simply view these components of a Java program as a magical incantation. Instead, we must carefully consider each component so that you understand its purpose. We will begin this process in Section 1.5.

The remainder of the body of the `TouchyWindow` class contains the specification of a second method named `onMouseRelease`:

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

As with `onMousePress`, the body of this method contains the instructions to be followed when the user performs a particular action with the mouse—releasing the mouse button. The instruction included in this case tells the computer to clear all graphics that have been displayed in the program's drawing area, named `canvas`.

Other special method names (`onMouseMove`, for example) can be used to specify how to react to other simple mouse events. We will provide a complete list of such methods in Sections 1.6.1 and 1.6.2.

There are a number of additional syntactic features visible in these method definitions that it is best not to explain in detail at this point. First, the method names are preceded by the words `public void`. For now, think of this as another magical incantation that you simply must include in the first line of almost every method definition you write. After the method names, the words `Location point` appear in parentheses. Like `public void`, you should be sure to include this text in the header of each method you define for a while. To make this part of the method header a bit less mysterious, however, we can give you a clue about its meaning. You might imagine that in many programs the instructions that respond to a mouse press would need to know where the mouse was pointing. In the next chapter, we will see that the `Location point` portion of such a method definition provides the means to access this information.

1.4 Programming Tools

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:

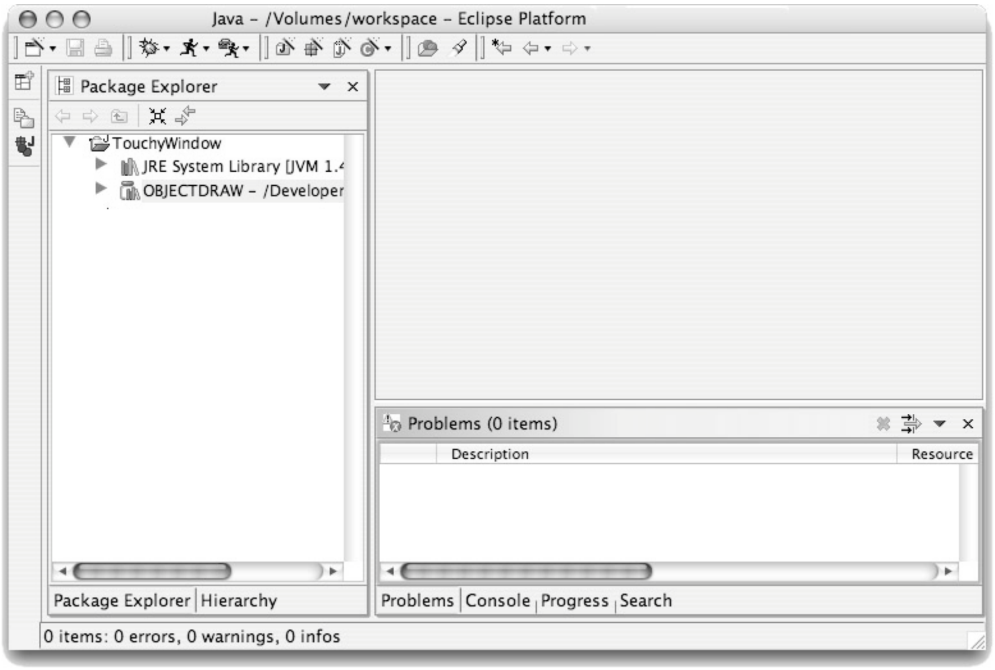
- Implementations of both IDEs are available for a wide range of computer systems including Windows systems, MacOS, and Unix and its variants.
- Both IDEs are available for free and can be downloaded from the Web.

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.



0 items: 0 errors, 0 warnings, 0 infos

Figure 1.5 An Eclipse project window

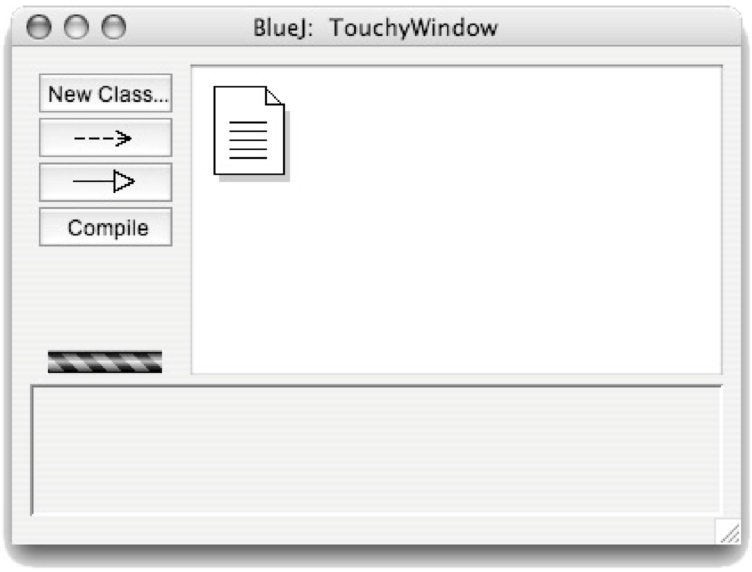
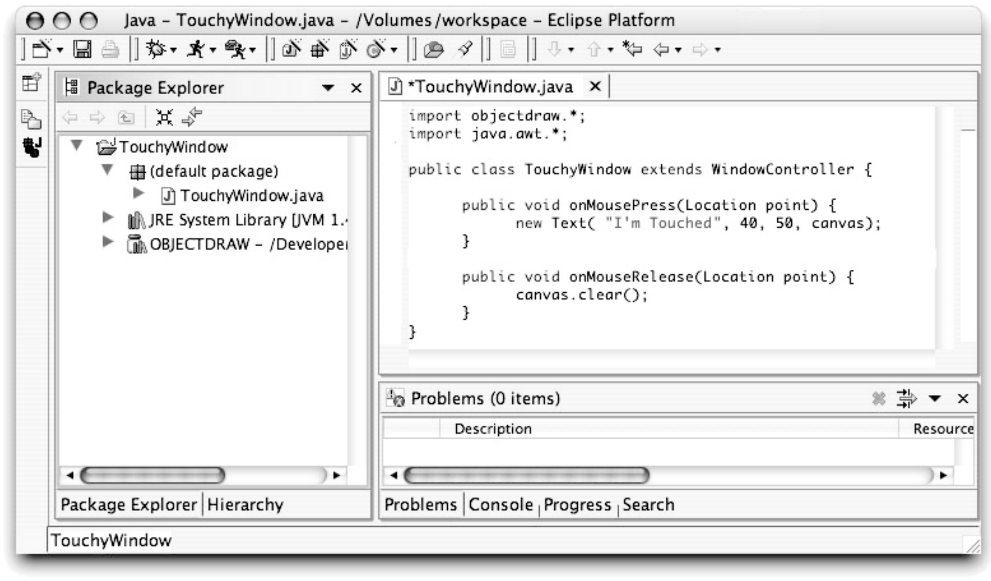


Figure 1.6 A BlueJ project window



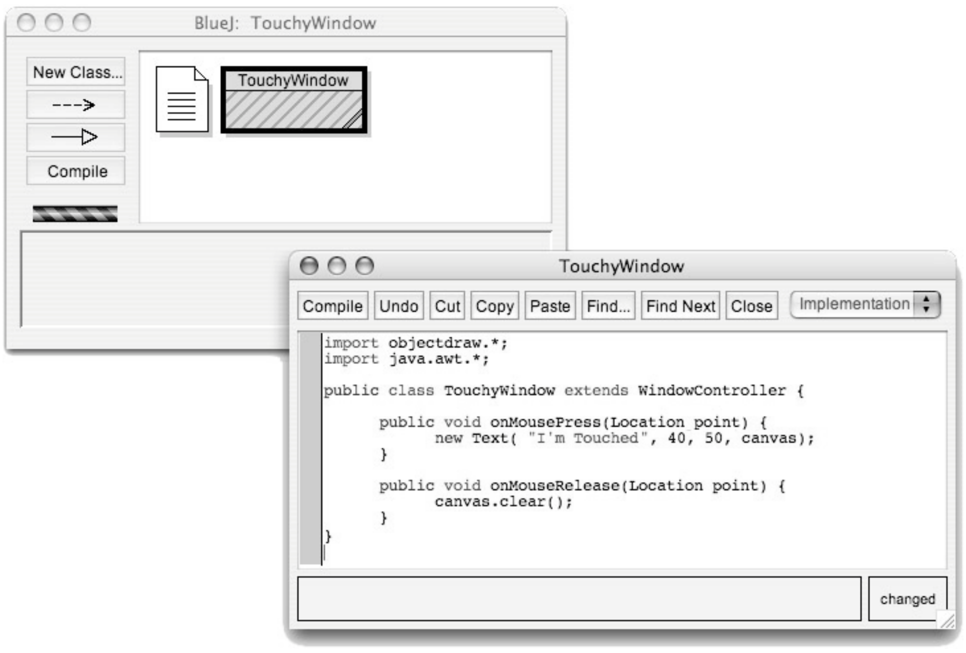
```
import objectdraw.*;
import java.awt.*;

public class TouchyWindow extends WindowController {

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

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

Figure 1.7 Entering the text of `TouchyWindow` under Eclipse

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.

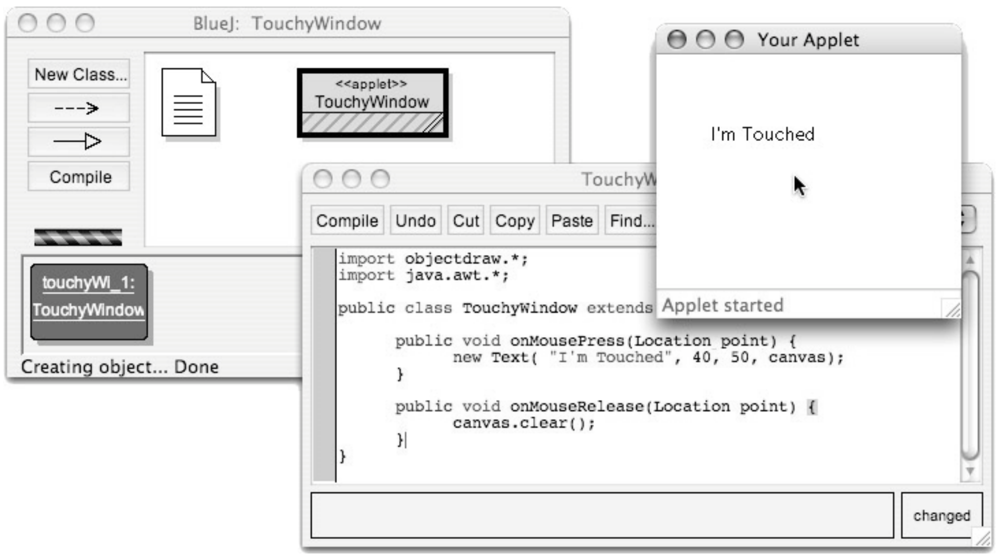
In both the BlueJ project window and the BlueJ window holding the text of the `TouchyWindow` class there is a button labeled "Compile". Pressing this button instructs BlueJ that we have completed entering our code and would like to have it translated into a form the machine can more easily interpret. 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.



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

public class TouchyWindow extends WindowController {

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

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

Figure 1.8 Entering the text of `TouchyWindow` under BlueJ



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

public class TouchyWindow extends

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

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

Figure 1.9 Running a program under BlueJ

In this discussion of how to enter and run a program we have overlooked one important fact. It is quite likely that you will make a mistake at some point in the process, leaving the IDE confused about how to proceed. As a result, in order to work effectively with an IDE you need some sense of what kinds of errors you are most likely to make and how the IDE will react to them. We will return to this issue after we teach you a bit more about how to actually write Java programs.

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

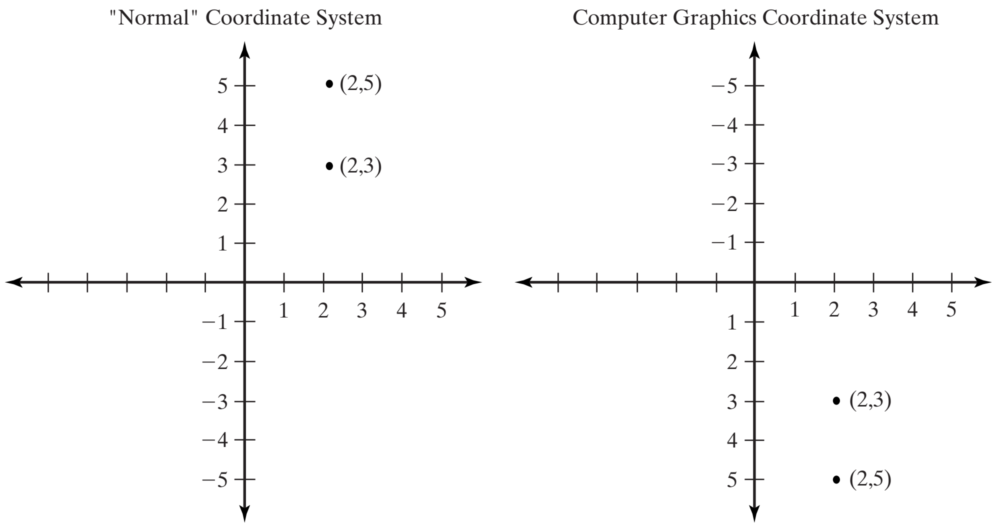

"Normal" Coordinate System
Computer Graphics Coordinate System
(2,5)
(2,3)
5
4
3
2
1
−1
−2
−3
−4
−5
1 2 3 4 5
−5
−4
−3
−2
−1
1
2
3
4
5
(2,3)
(2,5)

Figure 1.10 Comparison of computer and Cartesian coordinate systems

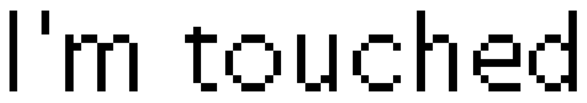


Figure 1.11 Text enlarged to make pixels visible

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 locations 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 one 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's 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 within 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 this window are (0,0). The window shown is 165 units wide and 100 units high. Thus the coordinates 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.

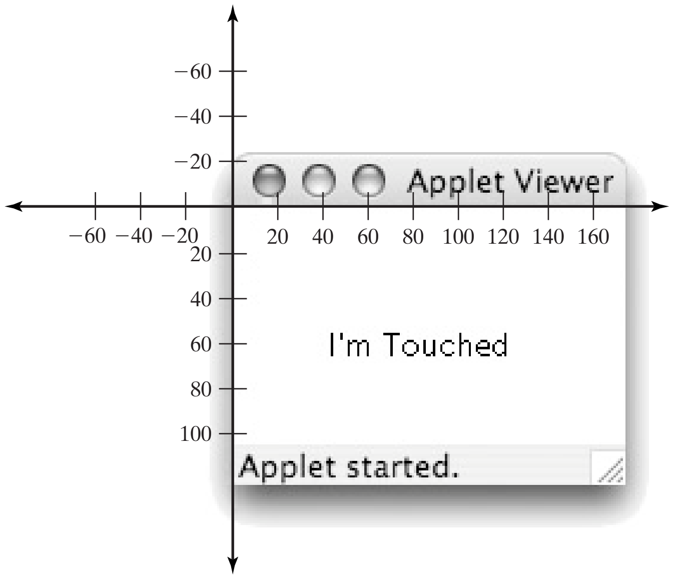


Figure 1.12 A program window and its drawing coordinate axes

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:

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

also ends with a semicolon. Most phrases that are not themselves commands, such as the headers 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 construction 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 50 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. To 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:

```
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:

```
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.

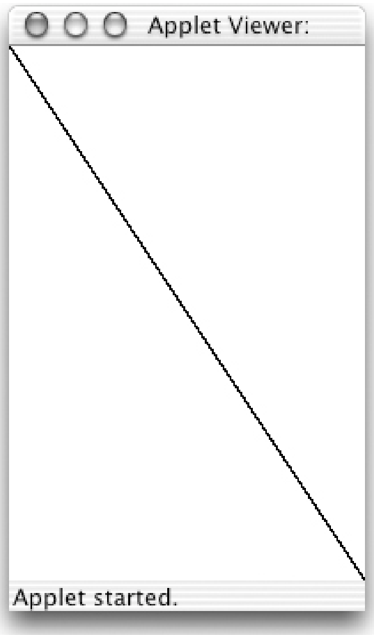


Figure 1.13 Drawing of a single line

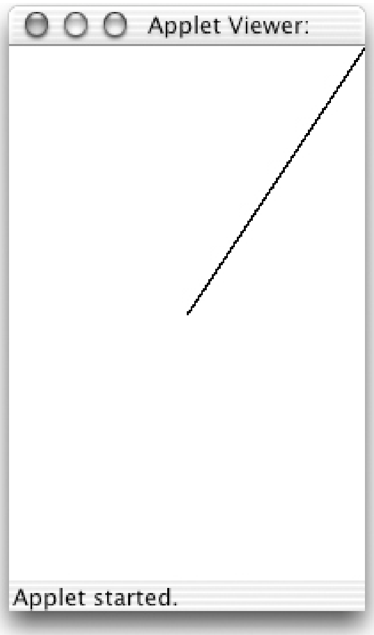
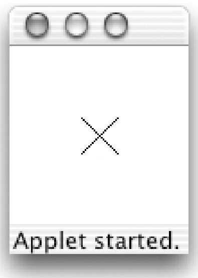


Figure 1.14 A line from (100,150) to (200,0)



```
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 1.15 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 the 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 it can fit within the rectangle described by its parameters. To illustrate this, Figure 1.16

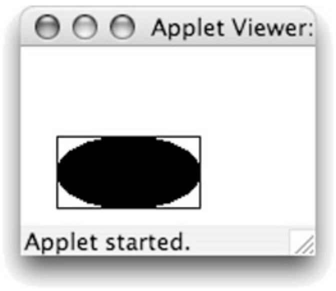


Figure 1.16 A `FilledOval` nested within a `FramedRect`

shows what the screen would contain after executing the two constructions

```
new FramedRect( 20, 50, 80, 40, canvas );
new FillOval( 20, 50, 80, 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`, `FramedOval`, and `FilledOval` will provide enough flexibility for our purposes.

Exercise 1.5.2

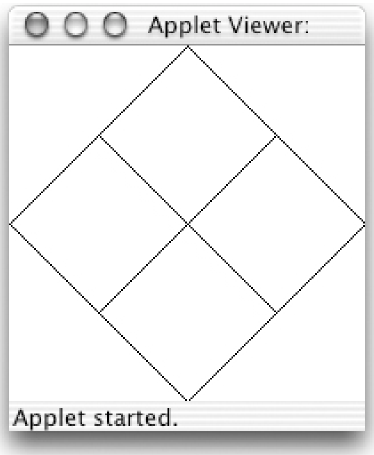
Sketch the picture that would be produced if the following constructions were executed. You should assume that the canvas associated with the program containing these instructions is 200 pixels wide and 200 pixels high.

```
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 );
```

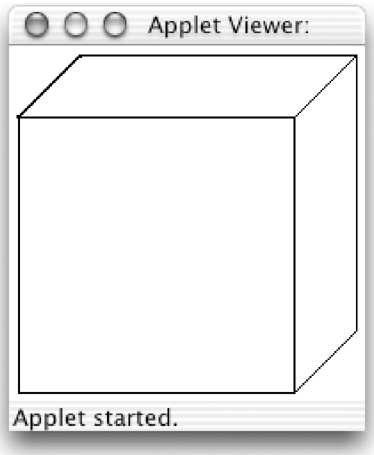
Exercise 1.5.3

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.*

a.



b.



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 we would like the computer to 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 you include 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 depressed.

`onMouseRelease` specifies the actions the computer should perform when the mouse button is released.

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

onMouseEnter specifies the actions the computer should perform when the mouse enters the program's canvas.

onMouseExit specifies the actions the computer should perform when the mouse leaves the program's canvas.

onMouseMove specifies the actions the computer should perform periodically while the mouse is being moved about without its button depressed.

onMouseDrag specifies the actions the computer should perform periodically while the mouse is being moved about with its button depressed.

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 words `Location point` are still 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 `TouchyWindow` 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 its 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 instructions

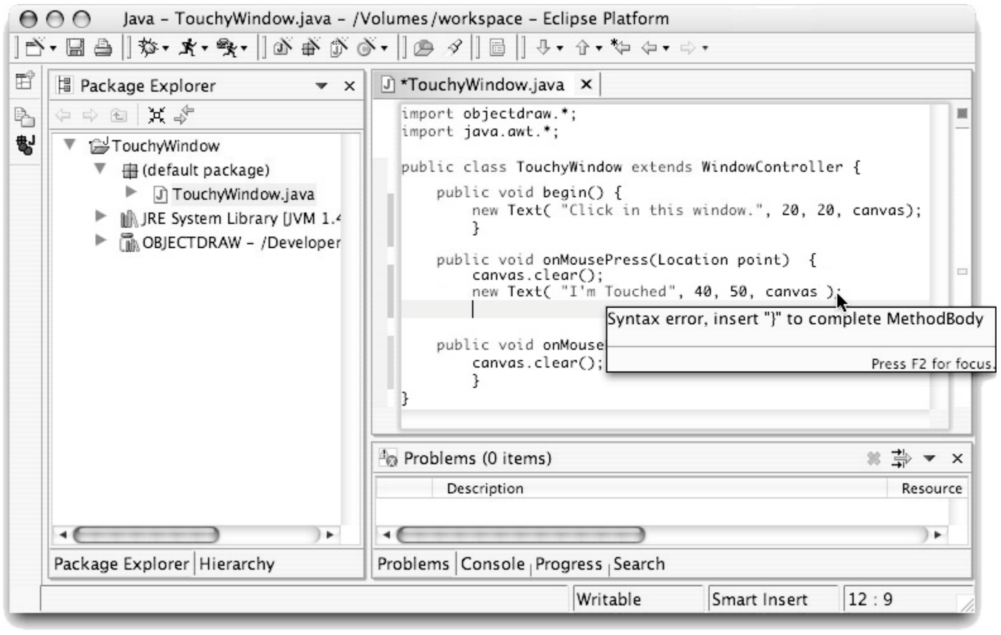
The code for this improved version of `TouchyWindow` is shown in Figure 1.17. In addition to adding the `begin` method, we have added the invocation `canvas.clear();` to `onMousePress` so that the instructions will be removed as soon as the user follows them.

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, the other person 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 turning 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, never 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 at 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 them


Java – TouchyWindow.java – /Volumes/workspace – Eclipse Platform

```
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 onMouse
        canvas.clear();
        }
}
```

Syntax error, insert "}" to complete MethodBody

Figure 1.18 Eclipse displaying a syntax error message

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.

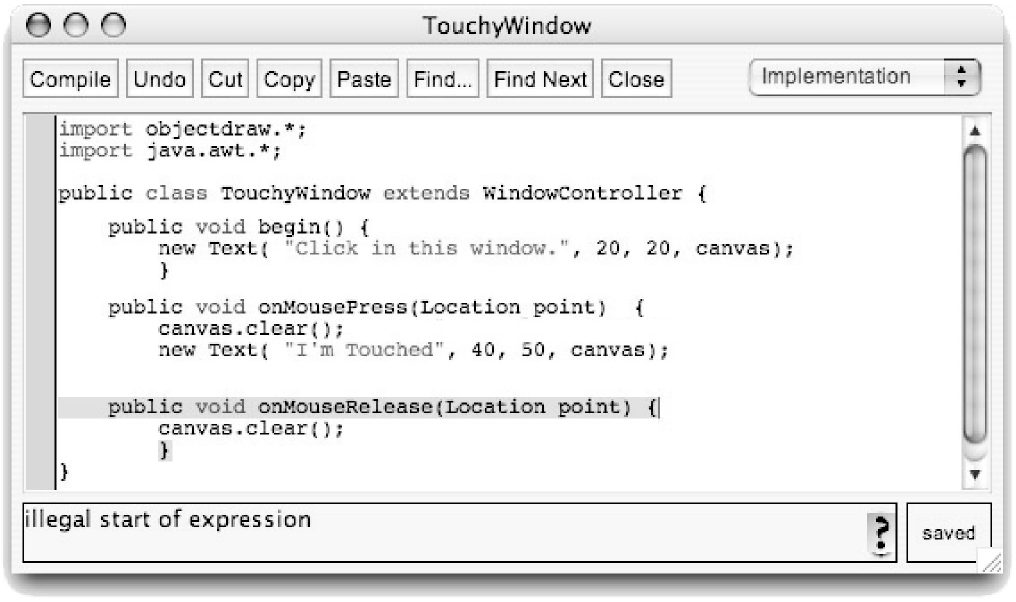
The mechanisms used to inform the programmer of syntactic errors vary from one IDE to another. Eclipse constantly examines the text you have entered and indicates fragments of your program that it has identified as errors by underlining them and/or displaying an error icon on the offending line at the 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();
```



```
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();
        }
}
```

```
illegal start of expression
```

Figure 1.19 BlueJ displaying a syntax error message

in the body of the `onMouseRelease` method of our example program, Eclipse will underline the word "clear" and display the message "Syntax error, insert 'AssignmentOperator ArrayInitializer' to complete Expression." In such cases, the error message is more likely to be a hindrance than a help. You will just have to examine what you typed before and after the position where the IDE identified the error and use your knowledge of the Java language to identify your mistake.

BlueJ is more patient about syntax errors. It ignores them until you press the Compile button. Then, before attempting to compile your code, it checks it for syntactic errors. If an error is found, BlueJ highlights the line containing the error and displays a message explaining the problem at the bottom of the window containing the program's text. Figure 1.19 shows how BlueJ would react to the mistake of leaving out the closing brace at the end of `onMousePress`. Note that different IDEs may display different messages and, particularly in the case where the error is an omission, associate different points in the code with the error. In this case, Eclipse flags the line before the missing brace, while BlueJ highlights the line after the error.

A program that is free from syntax errors is not necessarily a correct program. Think back to our instructions for performing calculations that was designed to leave you thinking about Danish elephants. If, while typing these instructions, we completely omitted a line, the instructions would still be grammatically correct. Following them, however, would no longer lead you to think of Danish elephants. The same is true for the example of saying "left" when you meant to say "right" while giving driving directions. Mistakes like these are not syntactic errors; they are instead called *logic errors*. They result in an algorithm that doesn't achieve the result that its author intended. Unfortunately, to realize such a mistake has been made, you often have to understand the intended purpose of the algorithm. This is exactly what computers don't understand. As a result, your IDE will give you relatively little help correcting logic errors.

As a simple example of a logical error, suppose that while typing the onMouseRelease method for the TouchyWindow 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 onMooseRelease. Look carefully. These names are not the same. onMooseRelease 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. onMooseRelease 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

```
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

```
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.

1.8 Summary

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 the 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 actually 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:

```
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.

```
new FramedRect( 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 );
```

❖

1.10 Programming Problems

Exercise 1.10.1

Learn how to enter and run a program using the tools available to you by entering the `TouchyWindow` *program discussed in this chapter. Once you are able to enter the program:*

- *Enter smaller or larger values where we had used the numbers 40 and 50 and see how the behavior of the program changes when run.*
- *Interchange the bodies of the two methods so that the construction appears in* `onMouseRelease` *and the* `canvas.clear` *is in* `onMousePress`. *How does the modified program behave?*

❖