Chapter 1

Your First Sip of Java

The task of learning any new language can be broken down into at least two parts: studying the new language's rules of grammar and learning vocabulary. This is true whether the language being learned 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 lesson involves learning the set of primitive commands you can issue to the computer. The vocabulary you must learn consists primarily of verbs that can be used to tell 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 based on a user input.

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. 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.

An important feature of a programming language is that the vocabulary used can be expanded by the programmer. As an example, 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 is being used. So, when you write the program, to talk about this position you must introduce a name that functions as a place holder for the information describing the mouse's position which will eventually become available when the program is run. Such names are somewhat like proper names used to refer to the character 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. Our brief introduction to programming in Java will conclude with a discussion of how to introduce and use such names in Java programs.

1.1 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 instruc-

tions that make up the program running on the computer. Each program must provide detailed instructions that the computer can follow in response to user actions. The examples presented in this section are intended to illustrate how this is done in a Java program.

When a typical application program is run, any number of windows, tool boxes, menu items or other graphical controls may appear on the computer's display. The instructions that constitute such programs must specify how the computer should react when the user manipulates any of these objects on the screen. To start things off simply, we will begin by restricting 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, etc. 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 soon as the user releases the mouse, the message disappears from the window. That is all it does! Not exactly Office 2001, but it is sufficient to illustrate the basic structure of many of the programs we will discuss in this text. In the on-line version of this document, a running version of the program is included so that you can try it and see for yourself.

The text required to create such a program in Java is shown below:

```
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();
      }


}
```

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") suggesting 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.

1.1.1 "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 one tries to construct programs 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 this chapter.

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*. Your programs 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*.

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 dot on the screen. If every program was built from scratch, every program would have to explicitly describe every one of these 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 two of the mysterious phrases found in the first lines of our program. The two 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`. Objectdraw 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 was 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.

The single open brace ("{") that appears at the end of the line that starts with the phrase `public class 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. In this case, 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.

1.1.2 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 them you can display a different message or make the message appear in a different position on the screen. Accordingly, you can not 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 the next subsection.

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 event 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`.

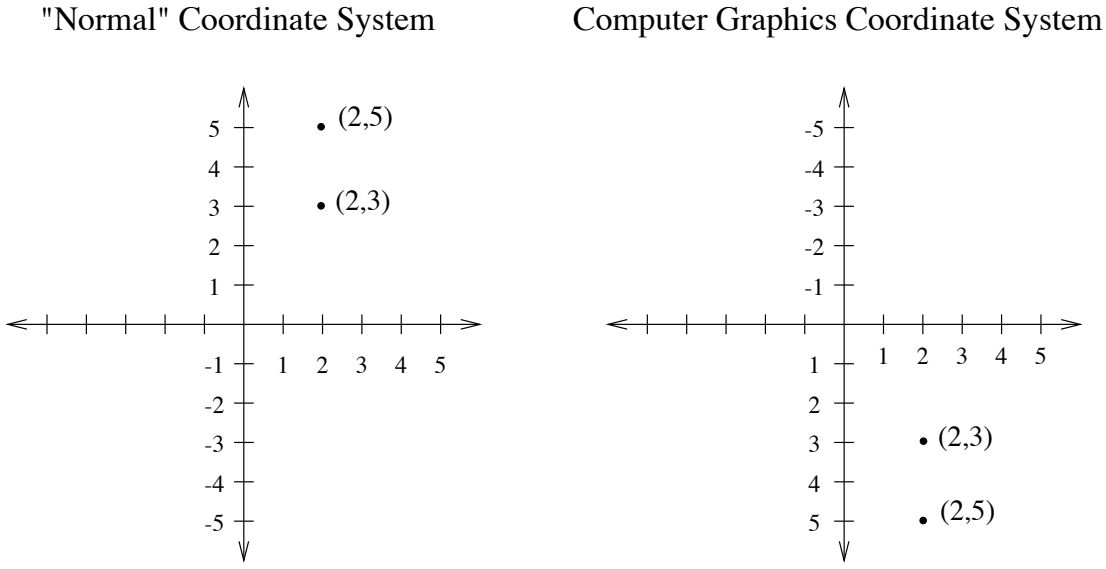
Other special method names (`onMouseMove`, etc.) can be used to specify how to react to other simple mouse events. We will provide a complete list of such methods in Sections 1.1.5 and 1.1.6

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 function header a bit less mysterious, however, we can give you a clue about its meaning. You might imagine that in real programs the instructions that respond to the mouse being pressed would need to know where the mouse was pointing. Shortly, we will see that the `Location point` portion of such a method definition provides the means to access this information.

1.1.3 The Graphics Coordinate System

Programs that display graphics on a computer screen have to deal extensively with a coordinate system similar to that 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 systems 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 the figure that shows where these two points fall in the 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. 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.

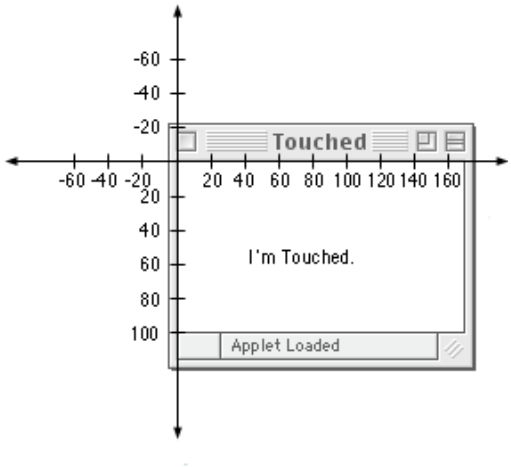
Another important aspect of the way in which coordinates are used to specify where graphics should appear is that there is not just a single set of coordinate axes used to describe locations anywhere on the computer's screen. Instead, there is a separate set of axes associated with each window on the 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 private 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 below.



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 a y coordinate of 50 and an x coordinate of 40.

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 fall within the boundaries of your canvas.

1.1.4 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 created;
- the name of the class of object to be created; 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*.

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. For example, the other command which appears in the `TouchyWindow` program:

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

also ends with a semicolon. Most phrases which 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`, 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 request 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 1.1. 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);
```

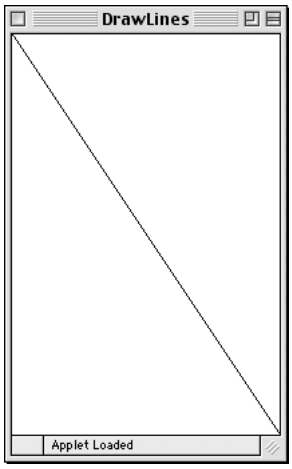


Figure 1.1: Drawing of a single line

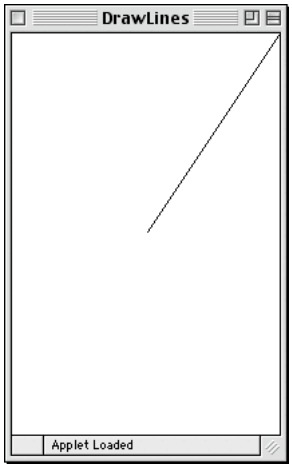
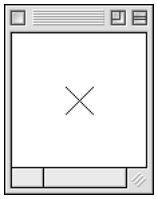


Figure 1.2:

Such a line is shown in Figure 1.2.

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 1.3.



```
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 1.3: A Program that draws two crossed lines.

The only differences between this example and `TouchyWindow` are the name given to the class defined (`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.

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

```
new FilledRect(90, 140, 20, 30, canvas);
```

will display a 20 by 30 solid black rectangular box in your canvas. The pair `90, 140` specifies the coordinates of the box's upper left corner. The pair `20, 30` specifies the width and height of the box. The command:

```
new FilledOval(90, 140, 20, 30, canvas);
```

will draw an oval on the screen. The parameters are interpreted just like those to the `FilledRect` construction. Instead of filling the rectangle, however, `FilledOval` draws the largest ellipse that it can fit within the rectangle described by its parameters. Other primitives allow you to draw just the outlines of rectangles and ovals 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 the online documentation of the library provided with this text. For now, the graphical object types, `Text`, `Line`, `FilledRect` and `FilledOval` will provide enough flexibility for our purposes.

1.1.5 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 must be identical except that `onMousePress` is replaced by the appropriate method name.

All of the mouse event handling methods are described below:

onMousePress specifies what should happen when the mouse button is depressed.

onMouseRelease specifies what should happen when the mouse button is released.

onMouseClick specifies what should happen 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 what should happen when the mouse enters the program's canvas.

onMouseExit specifies what should happen when the mouse leaves the program's canvas.

onMouseMove specifies what should happen periodically while the mouse is being moved about without its button depressed.

onMouseDrag specifies what should happen periodically while the mouse is being moved about with its button depressed.

1.1.6 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, the `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.

The code for this improved version of `TouchyWindow` is shown in Figure 1.4.

```
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)  {
            new Text( "I'm Touched.", 40, 50, canvas);
      }

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

}
```

Figure 1.4: A Simple Program with Instructions

1.2 Naming and Modifying Objects

Construction primitives like:

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

provide the means to place a variety of graphic images on a computer screen. Most programs that display graphics, however, do more than just place graphics on the screen. Instead, as they 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 how to apply operations called *mutator methods* to objects after they have been constructed.

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:

```
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 which 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 start the construction of such a program. Basically, from the description it is clear that the program needs to construct an oval 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:

```
public void begin() {
    new FilledOval( 50, 150, 100, 100, canvas);
    new Text( "Please click the mouse repeatedly", 20, 20, canvas);
}
```

This code should produce an image looking quite a bit like that shown in Figure 1.5. Of course, it might help your imagination if the "sun" were yellow, but we will have to wait until we learn a bit more Java before we can fix that.

Now, consider how we would complete the program by writing the `onMouseClick` method. To make an object move directly upwards, we need to use the `move` method specifying 0 as the distance to move horizontally and some negative number for the amount of vertical motion desired since y coordinates decrease as we move up the screen. Something like:

```
move(0,-5);
```

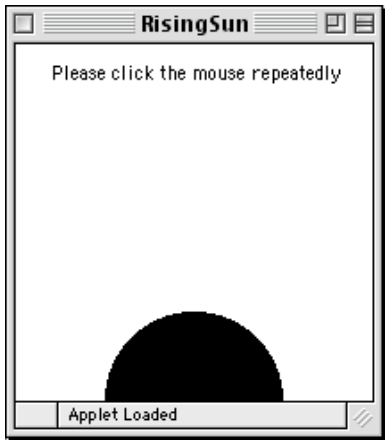


Figure 1.5:

might seem appropriate. The problem is that if this is all we say, Java will not know what to move. In the version of the `begin` method above, we construct two graphical objects, the filled circle and the text displaying the instructions. Since they are both graphical objects, we could move either of them. If all we say is "move", then Java has no way of knowing which one we want moved.

To avoid ambiguities like this, Java won't let us simply say `move`. Instead we have to tell a particular object to move. In general, Java requires us to identify a particular object as the target whenever we wish to use a mutator method. You have already seen an example of the Java syntax used to provide such information. In our first example program, when we wanted to remove a message from the screen we included a line of the form:

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

"clear" is a mutator method associated with drawing areas. The word "canvas" is the name Java gives to the area in which we can draw. Saying "canvas.clear" tells the area in which we can draw that all previous drawings should be erased. When we tell an object to perform a method in this way we say we have invoked or applied the method.

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.

First, we have to choose a name to use. Java puts very few restrictions on the names we can pick. A name must start with a letter. After the first letter, you can use either letters, digits or underscores. So, we could name our oval something like `sunspot`, `oval2move` or `ra`. Case is significant. A name can be as long (or short) as you like, but it must be just one word (i.e. no blanks are allowed in the middle of a name). 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 names starting with lower case letters for everything except classes.

We can use a sequence of letters, numbers and underscores as a name in a Java program even if it has no meaning in English. Java would be perfectly happy if we named our box `e2d_iw0`. 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

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

public class RisingSun extends WindowController {

   private FilledOval sun;

   public void begin () {

...
```

Figure 1.6:

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. The declaration 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 tells Java the type of object that will eventually be associated with the name. The purpose of such a declaration is to enable Java to give you helpful feedback if you make a mistake. Suppose that after deciding to use the name "sun" in our program we made a typing mistake and typed "sin" in one line where we meant to type "sun". It would be nice if when Java tried to run this program it could notice such a mistake and provide advice on how to fix the error similar to that provided by 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 the names used provide this dictionary. If Java encounters a name that was not declared it reports it as the equivalent of a spelling mistake.

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 semi-colon. So, to declare the name `sun`, which we intend to use to refer to a `FilledOval`, we would type the declaration:

```
private FilledOval sun;
```

The form and placement of a declaration within a program determines where in the program the name can be used. 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. We recommend that such declarations be placed before all the method declarations. Names declared in this way are called *instance variables*. The inclusion of the word `private` indicates that only code within the class we are defining should be allowed to refer to this name. With this declaration added, the contents of the program file for our animation of the sunrise might begin with the code shown in Figure 1.6. Recall, that a declaration does not tell Java to what a name refers. It only informs Java of the type of things to which the name *might* refer. Before the name can be used in a command like:

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

we must associate the name with a particular object using a command Java calls an assignment statement. As an example, the form of assignment needed to associate the name `sun` with the oval on our screen is

```
sun = new FilledOval(50, 150, 100, 100, canvas);
```

This particular command includes a construction for the new filled oval. Ordering is critical in an assignment. The name being defined must be placed on the left side of 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 non-intuitive since Java must obviously first perform the construction described on the right before it can associate the new object with the name on the left and we are used to processing information left to right. Java, however, will reject the command as nonsense if we interchange the order of the name and the construction.

Given this introduction to associating names with objects, we can now show the complete code for a "rising sun" program. It appears in Figure 1.7. It includes examples of all three of the basic constructs involved in using names: declarations, assignments and references.

- The declaration:

  ```
  FilledOval sun;
  ```

  appears at the beginning of the class body.

- An assignment of a meaning to a name appears in the `begin` method:

  ```
  sun = new FilledOval( 50, 150, 100, 100, canvas);
  ```

- A reference to an object through a name appears in `onMouseClick`:

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

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, one 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 an 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.

```
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

// Place the sun and some brief instructions on the screen
   public void begin() {
      sun = new FilledOval( 50, 150, 100, 100, canvas);
      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(0,-5);
   }

}
```

Figure 1.7: Code for Rising Sun Example

1.2.1 Additional Mutator 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 mutator method named `moveTo` which moves an object to a specific location on the screen. There are also a pair of 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 often 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 was 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.

All that 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.

```
   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 mutator methods 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.

An obvious name for this object is "instructions". If this is our choice, then we would need to tell Java that we planned to use this name by adding a declaration of the form:

```
private Text instructions;
```

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

```
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. Using multiple lines for one command like this is perfectly acceptable in Java. It is the semi-colon 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.

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

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

in the `onMouseClick` method. Finally, when the program is reset, the instructions should reappear. To do this, we would include and an instruction of the form

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

in the `onMouseExit` 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 and a single `show` will still make them reappear.

Note that even though `hide` and `show` expect no parameters, Java still expects us to place the parentheses that would surround the parameters after the method's name when it is used.

The complete text of this revised program is shown in Figure 1.8.

1.3 Non-graphical Classes of Objects

We have seen how to construct graphical objects of several varieties, associate names with them and apply methods to these objects. All of the objects we have worked with, however, have shared the property that they are graphical in nature. When we create any of these objects, they actually appear somewhere on the computer's screen. This is not a required property of objects that can be manipulated by a Java program. In this section we will introduce two classes of objects that are related to producing graphics but do not correspond to particular shapes that appear on your screen.

```
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 onMouseClick(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 1.8: Rising Sun Program with Reset Feature

1.3.1 The Class of Colors

So far, all the graphics we have drawn on the screen have appeared in black. Black is the color in which graphics are drawn by default. To add a little variety to the display, we can change the color of any graphical object using a mutator method named "setColor". As an example, it would certainly be an improvement to make the sun displayed by our `RisingSun` program appear yellow or any other more "sun-like" color than black.

It is quite simple to make this change. All we have to do is add an invocation of `setColor` to our `begin` method. The revised method would look like:

```
public void begin() {
   sun = new FilledOval( 50, 150, 100, 100, canvas);
   sun.setColor(Color.yellow);
   instructions = new Text( "Please click the mouse repeatedly",
                                20, 20, canvas);
}
```

The `setColor` method expects us to include a parameter specifying the color to be used. The simplest way to specify the color is to use one of Java's built-in color names. We chose yellow for our sun, but if we wanted a more dramatic sunrise we could have instead used `Color.orange` or even `Color.red`. Java provides names for all the basic colors including `Color.blue`, `Color.green`, and even `Color.black` and `Color.white`. Of course, there are too many shades of each color to assign a name to every one. Accordingly, Java provides an alternative mechanism for describing colors of a wider range.

We have seen that names in Java can be associated with objects like the drawing area (canvas), or graphical objects we have created. You might suspect that if a color can be associated with a name it might be thought of as an object. In this case, you would be right.

What properties do colors share with the other classes of objects we have seen? First, they can be constructed. Just as we have been able to say "new FilledRect(...)" to create rectangles, we can construct new colors. This is how Java gives us access to the vast range of colors that can be displayed on the typical computer screen. If we want to set the color of our sun to something not included in the small set of colors that have names, we can describe the particular color we want by saying:

```
new Color(...)
```

as long as we know the right information to use in place of the dots between the parentheses.

When we want to construct a new color for use in a program, we must provide a numerical description of the color as parameters to the construction. Java uses a system that is fairly common for specifying colors on computer systems. Each color is described as a mixture of the three primary colors: red, green and blue.[1] The mixture desired is specified by giving three numbers, each of which specifies how much of a particular color should be included in the mixture. Each of the numbers can range from 0 ("use none of this particular primary color") to 255 ("use as much of this color as possible"). The numbers are listed in the order "red, green, blue". So "0,0,0" is black.

[1] If you thought the primary colors were red, yellow and blue you aren't confused. Those are the primary colors when mixing materials that absorb light (like paint). When mixing light itself (as in flashlight beams or the light given off by the phosphors on a computer screen), red, blue and green act as the primary colors. Mixing red and blue lights produce purple. Mixing red and green produces yellow. Mixing all three primary colors together produces white.

"255,255,255" is white. "255,0,255" is a shade of purple. So, if you want to make the `sun` purple you could construct the desired color by saying:

```
new Color(255,0,255)
```

Colors are also similar to objects such as FilledOvals in that you can associate names with them. So, if you wanted to use the color purple in a program you might first declare an instance variable named "purple" as:

```
private Color purple;
```

Then, in the `begin` method you could associate the name with the actual color by saying:

```
purple = new Color( 255, 0, 255);
```

You could then make the sun purple by replacing the the `setColor` used to make the sun yellow by:

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

It is worth noting that it is not actually necessary to introduce a name to use a color created by providing a numeric description of a color in a construction. When we use the `setColor` method, we have to provide the color we wish to use as a parameter to the method, but we can describe the desired color in several ways. Java does not insist that we use a name to identify the color. In particular, we could make the sun purple by simply saying

```
sun.setColor( new Color( 255, 0, 255) );
```

The construction describes the color just as well as the name `color` from Java's point of view.

In general, wherever Java allows us to identify an object or value by name, it will accept any other phrase that describes the equivalent object or value. This applies not just to names used to describe parameter values but also to names used to indicate the object we wish to alter. For example, it is legal to replace the name `sun` in the method invocation

```
sun.setColor( Color.yellow );
```

with a construction that describes a graphical object as in:

```
new FilledOval( 50, 150, 100, 100, canvas).setColor( Color.yellow);
```

This command tells Java to create a new FilledOval on the screen and to immediately set its color to yellow.

While a command of this form is legal in Java, this particular example could not be used practically in our RisingSun program. The problem is that such a command creates a new graphical object but does not associate a name with the object. As we have seen, without a name through which we can refer to the FilledOval that represents the sun, we could not make the sun appear to rise by telling it to move in our `onMouseClick` method.

There are situations where we wish to set the color of a graphical object but then have no further need to refer to the object. In such situations, combining a construction and an invocation of `setColor` into a single command might make sense. For example, suppose we wanted to display a blue background behind our rising sun. We could do this by adding the command:

```
new FilledRect( 0, 0, 200, 200).setColor( Color.blue );
```

as the first line of the program's `begin` method.

1.3.2 The Location Class

As we have seen, we can turn a triple of numbers into a single object by making a new color. We can turn pairs of numbers into objects by constructing an object of another class called `Location`. The constructor for the class `Location` treats pairs of numbers as coordinates within the graphical coordinate system and produces objects that represent the positions on the screen described by the specified coordinates. The name `Location` should sound familiar. It is part of the thus far un-explained notation `(Location point)` that appears in the headers of all mouse event handling methods. Once we introduce this class of objects, we can explain the function of this notation in method declarations.

To construct a new `Location` object you simply say something like:

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

replacing the parameters "50, 150" with the coordinates of whatever point you are trying to describe. A line containing just this construction would have no effect on the behavior of a program. Nothing appears on the screen when a `Location` is created and if no name is associated with an object when it is created you can never refer to it later. It is far more likely that one first would declare a name that can refer to `Location` objects by typing something like:

```
Location initialPosition;
```

and then associate the name with an actual coordinate pair through an assignment of the form:

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

The real worth of `Location`s comes from the fact that they can be used to take the place of typing a pair of numbers to describe a point in the coordinate system when writing a construction for any of the graphical classes we have described or invoking a method that requires a coordinate pair such as `moveTo`. For example, the instruction used to construct the sun in our `RisingSun` program looks like:

```
sun = new FilledOval( 50, 150, 100, 100, canvas );
```

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 FilledOval( 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( initalPosition );
```

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 `moveTo` written 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 `Location`s 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 can see 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 a `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` is later used to position some graphical object.

To clarify how `translate` works, consider the sample program shown in Figure 1.9. In its

```
import objectdraw.*;
import java.awt.*;
    // A program that uses the translate method to draw a
    // grid of thick black lines on the canvas
public class DrawGrid extends WindowController {

    Location verticalCorner;    // Upper left corners of the next two
    Location horizontalCorner;  //   lines to draw

        // Set Locations to position first pair of lines at upper
        // left corner of the canvas
    protected void begin() {
        horizontalCorner = new Location(0,0);
        verticalCorner = new Location(0,0);
    }

        // Draw a pair of lines and move Locations so that the next
        // pair of lines will appear further down and to the right
    protected void onMouseClick(Location point) {
        new FilledRect(verticalCorner, 5, 200, canvas );
        new FilledRect(horizontalCorner, 200, 5, canvas );

        verticalCorner.translate(10,0);
        horizontalCorner.translate(0,10);
    }
}
```

Figure 1.9: An Application of the Translate Method

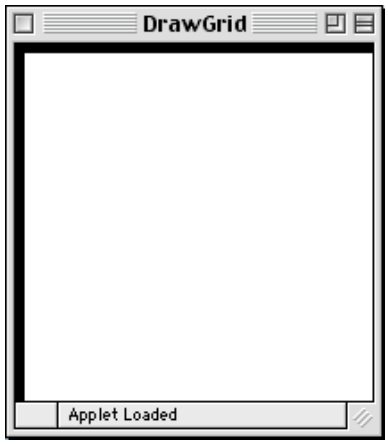


Figure 1.10: Display after 1 Click

`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 is named `verticalCorner` and the other is named `horizontalCorner`. Although they are created to describe the same position initially, two distinct objects are needed because they will be modified to describe different locations using the `translate` method in other parts of the program.

The names assigned to these two `Locations` reflect the way they are used in the program's other method, `onMouseClick`. Each time the mouse is clicked, this method creates two long, thin rectangles on the screen. The rectangle created by the first construction in `onMouseClick` is a long vertical rectangle. The position of its upper left corner is determined by the `Location` named `verticalCorner`. The other rectangle is a long horizontal rectangle and its position is determined by the `Location` named `horizontalCorner`.

Since both of the `Locations` initially describe the upper left corner of the canvas, the first time the mouse is clicked, the rectangles created by the execution of the first two lines of `onMouseClick` will produce a drawing like that shown in Figure 1.10.

The last two commands in `onMouseClick` tell the `Location` objects named `verticalCorner` and `horizontalCorner` to translate so that they describe new positions on the screen. `verticalCorner` is changed to describe the position 10 pixels to the right of its initial position. `horizontalCorner` is translated 10 pixels down from its initial position. When these commands are completed, nothing changes on the screen. The program's window will appear as shown in Figure 1.10 before the two translations are performed and it will still look the same after they have been performed. The `Location`s will then describe new positions, but the two rectangles will remain where they were initially placed even though the `Location`s were used to describe their positions when they were constructed.

The next time the mouse is clicked the two constructions at the beginning of `onMouseClick` are performed based on the translated positions of the two `Locations`. Accordingly, the vertical rectangle created will appear a bit farther to the right and the horizontal rectangle will appear a bit farther down the screen as shown in Figure 1.11. After these rectangles are created, the last two lines of the method will shift the `Locations` used farther to the right and down the screen so that the rectangles produced by the next click will appear at the correct locations.

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 1.12. Just a few additional

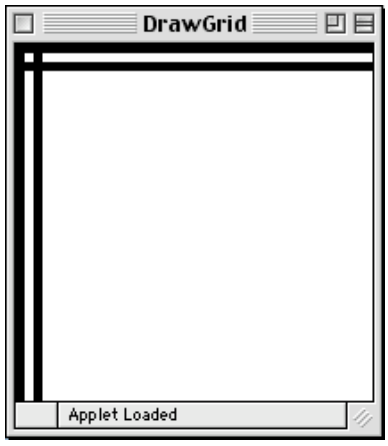


Figure 1.11: Display after Second Click

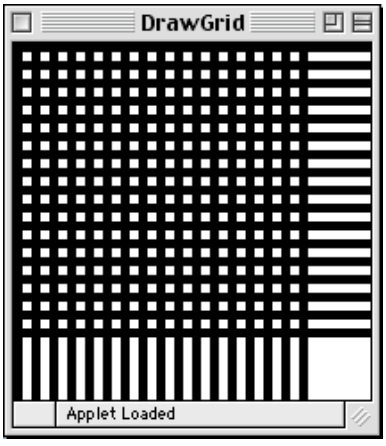


Figure 1.12: Display after many clicks

clicks will complete the process of filling the screen.

1.4 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` 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.

```
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 at 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 an 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 an instance variable declaration, 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 following program, which uses the name `mousePosition` as its formal parameter instead of `point` will behave exactly like the version that used the name `point`.

```
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);
    }
}
```

Another important aspect of the behavior of formal parameter names like `point` or `mousePosition` is that each parameter name is meaningful only within the method whose header contains its declaration. To illustrate this, consider the program shown below:

```
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);
    }
}
```

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`.

1.5 Sharing Parameter Information between Methods

Suppose now that we wanted to write another program that would display the words "Pressed" and "Released" just as 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

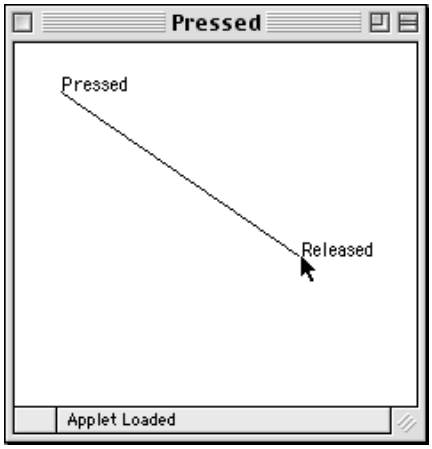


Figure 1.13:

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 1.13.

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:

```
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 in `onMouseRelease` as an error and refuse to run the program. In general, formal parameter names can not 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, the `begin` method created the oval that represented the sun and the `onMousePress` later moved the oval. This was arranged by associating the name `sun` with the oval. Similarly, if we want two methods to share a `Location` that describes some mouse position we must associate the `Location` with an instance variable.

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 this 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 1.14.

When the mouse is pressed, the assignment

```
firstPoint = pressPoint;
```

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

    // A program that displays the words "Pressed" and "Released"
    // where the mouse button is pressed and released while connecting
    // each such pair of points with a line.
public class ConnectTwo extends WindowController {

    private Location firstPoint;   // The location where button was pressed

        // Display "Pressed" when the button is pressed.
    public void onMousePress(Location pressPoint) {
        new Text("Pressed", pressPoint, canvas);
        firstPoint = pressPoint;
    }

        // Display "Released" and draw a line from where the mouse
        // was last pressed.
    public void onMouseRelease(Location releasePoint) {
        new Text("Released", releasePoint, canvas);
        new Line( firstPoint, releasePoint, canvas);
    }
}
```

Figure 1.14: A Program to Track Mouse Actions

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 using 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 with 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

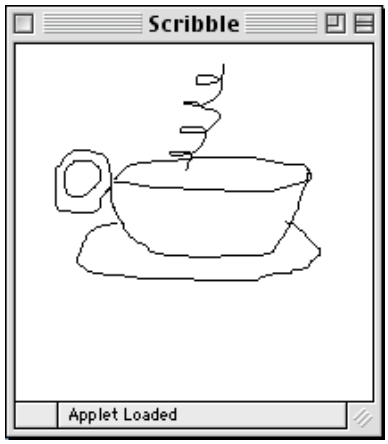


Figure 1.15:

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 was a pencil. A sample of the kind of scribbling we have in mind is shown in Figure 1.15. 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 what 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. So, within this method, we want to include an instruction like:

```
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 `Location`s.

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 header we use when declaring `onMouseDrag` looks like:

```
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. Think carefully for a moment about the beginning of the process of drawing with this program. 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 where 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 the point 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 of the `onMousePress` method but we need to access it in the `onMouseDrag` method because this is the method that will actually draw the line. We can arrange for `onMousePress` to share the needed information with `onMouseDrag` by declaring the name `previousPosition` as an instance variable and including an appropriate assignment in `onMousePress` to associate this name with the position where the mouse button is first pressed.

Given this analysis, we will include an instance variable declaration of the form:

```
private Location previousPosition;
```

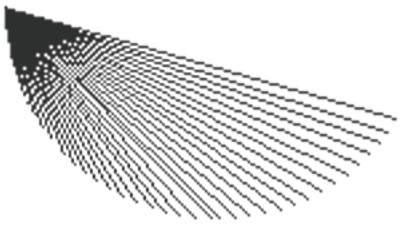
and then write the following definition for `onMousePress`:

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

We also know that the `onMouseDrag` method must contain the `Line` construction shown above and that its formal parameter should be named `currentPosition`. So, a first draft of this method would be:

```
public void onMouseDrag( Location currentPosition) {
    new Line( previousPosition, currentPosition, canvas);
}
```

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 arc counterclockwise 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` often enough. 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:

```
previousPosition = currentPosition;
```

at the end of the `onMousePress` yielding the complete program shown in Figure 1.16.

1.6 Summary

In this chapter we have covered a selection of Java's mechanism designed to bring you to the point where you can understand the overall structure of some simple but useful programs.

We have seen how the notation of method declarations makes it possible to specify a sequence of operations that describes how the program should react to a particular type of mouse event. In subsequent chapters we will see how to generalize these ideas to write programs that can react to a wider variety of external events.

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

We explored the importance of the use of names to refer to the objects our programs manipulate. Instance variable names were used to share information between methods and formal 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.

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.

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

    // A Simple Drawing Program
    //
    // This program allows its user to draw simple lines on the screen
    // by depressing the mouse button and then dragging the mouse
    // on the screen with the button depressed.

public class Scribble extends WindowController {

    Location previousPosition;   // Last known position of mouse

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

        // When the mouse is dragged, connect its current and previous
        // positions with a line
    public void onMouseDrag( Location currentPosition) {
        new Line( previousPosition, currentPosition, canvas);
        previousPosition = currentPosition;
    }

}
```

Figure 1.16: A Simple Sketching Program