CHAPTER 2

What's in a Name?

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

2.1 Naming and Modifying Objects

Constructions like:

```
    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 about operations that change the properties of objects after they have been constructed. These operations are called *mutator methods*, based on the nonbiological meaning of the word “mutate”, to change or alter.

2.1.1 Mutator Methods

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

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

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

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

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

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

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

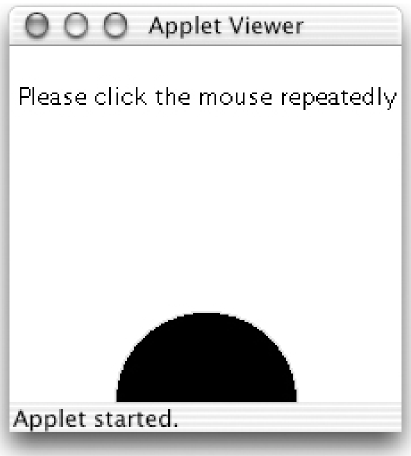


Figure 2.1 An oval rises over the horizon

This code will produce an image like that shown in Figure 2.1. In an effort to make it look like the sun is rising over the horizon, the oval is positioned so that its bottom half extends off the bottom of the window, leaving only the top half visible. 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 upward, 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 );
```

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. Alternately, we might say that we sent a `clear` message to the `canvas`.

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

2.1.2 Instance Variable Declarations

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

We can use a sequence of letters, numbers, and underscores as an identifier in a Java program even if it has no meaning in English. Java would be perfectly happy if we named our 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 that you could think of the display produced by the program we are trying to write as an animation of the sun rising. In this case, `sun` would be an excellent name for the oval. We will use this name to complete this example.

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

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

```
private FilledOval sun;
```

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

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

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

public class RisingSun extends WindowController {

   private FilledOval sun;

   public void begin () {

       ...
```

Figure 2.2 Declaring `sun` in the `RisingSun` program

The declaration of an instance variable does not determine to which object the name will refer. Instead, it merely informs Java that the name will be used at some point in the program and 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.

2.1.3 Assigning Meanings to Variable Names

Before a 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*. An assignment statement consists of a name followed by an equals sign and a phrase that describes the object we would like to associate with that name. As an example, the assignment statement needed to associate a name with the oval that represents the sun in our program is:

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

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

Ordering is critical in an assignment. The name being defined must be placed on the left side 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 nonintuitive, 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.

Java will also reject an assignment statement that attempts to associate a name with an object that is not of the type included in the name's declaration. The declaration included in Figure 2.2 states that `sun` will be used to refer to a `FilledOval`. If we included the assignment

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

in our program, it would be identified as an error, because it attempts to associate the name with an object that is a `FilledRect` rather than with a `FilledOval`.

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

- The declaration:

  ```
  private 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 );
  ```

2.1.4 Comments

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

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

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

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

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

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

   private FilledOval sun;      //  Circle that represents the sun

   // 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 2.3 Code for rising sun example

Many programmers prefer to format multiline comments as follows:

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

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

2.1.5 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 two methods named `hide` and `show` which can be used to temporarily remove a graphical item from the screen. We can use these three methods to extend the behavior of our `RisingSun` program.

First, the version of the program shown above becomes totally uninteresting after the mouse has been clicked 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 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 need to tell Java that we plan to use this name by adding a declaration of the form:

```
private Text instructions;
```

to the beginning of our class. We 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. It can make your code much easier to read and understand. Using multiple lines for one command like this is perfectly acceptable in Java. It is the semicolon rather than the end of a line that tells Java where a command ends. You can split an instruction between two lines at any point where you could type a space except within quoted text. You will also find that the use of indentation and blank lines can make groups of related commands stand out to the reader. These issues are discussed further in Appendix A.

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

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

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

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

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

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

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

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

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

   private FilledOval sun;     //  Circle that represents the sun
   private Text instructions;  //  Display of instructions

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

   // Move the sun up a bit each time the mouse is clicked
   public void 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();
   }
}
```

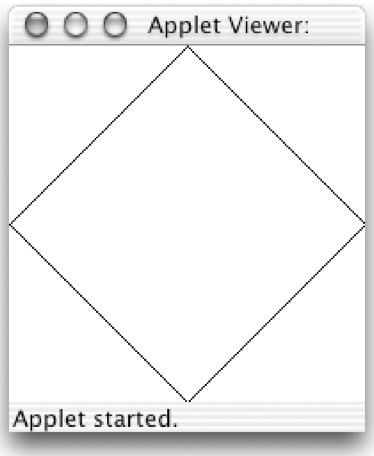
Figure 2.4 Rising sun program with reset feature

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

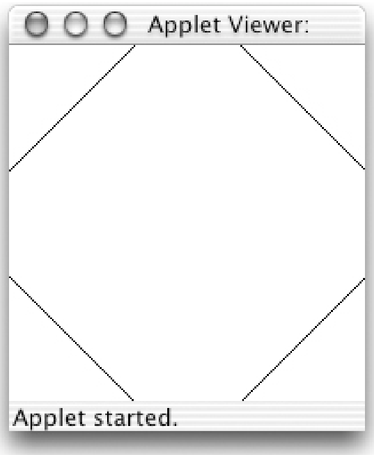
2.1.6 Exercises

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

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



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



Exercise 2.1.1

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

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

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

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

In each case, briefly explain your answer.

```
3rdLine        thirdLine          line3
Leftlower      lower right line   lowerLeft
southEast      S.E.               south-east
```

❖

Exercise 2.1.2

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

Exercise 2.1.3

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

Exercise 2.1.4

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

EXERCISE 2.1.5

Assuming that the names `leftToTop`, `topToRight`, `rightToBottom`, *and* `bottomToLeft` *had been declared appropriately and associated with the lines of the diamond using assignment statements, describe the effect the following statements would have on the lines each time they were executed. In addition, sketch the contents of the window after these lines were executed 100 times.*

```
leftToTop.move( 0, 1 );
topToRight.move( 0, 1 );
rightToBottom.move( 0, −1 );
bottomToLeft.move( 0, −1 );
```

2.2 Nongraphical 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.

2.2.1 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 "sunlike" 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`.[1] 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.

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.[2] 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, "255, 255, 255" is white, "0, 255, 0" is solid green, and "255, 0, 255" is a shade of purple. If you wanted to make the `sun` purple, you could construct the desired color by saying:

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

`Color`s are objects. Therefore, just as you can associate names with objects such as FilledOvals, you can associate names with `Color`s. 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 `setColor` used to make the sun yellow by:

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

[1] In early versions of Java, the names used for colors used lowercase letters. For example, the name `Color.red` was used instead of `Color.RED`. These names are still supported, but their use is discouraged. Throughout the text, we will use the capitalized versions of color names.

[2] 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 produces purple. Mixing red and green produces yellow. Mixing all three primary colors together produces white.

It is worth noting that it is not actually necessary to introduce a new name to use a color created using 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. 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 `purple` 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.

Exercise 2.2.1

Write a method that creates a red oval when the mouse is clicked. The oval should be 100 pixels in width and 75 pixels in height and should appear 50 pixels down from the top of the canvas and 50 pixels in from the left of the canvas.

2.2.2 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`. In the construction of a `Location`, two numbers are provided as parameters. These numbers are treated as coordinates within the graphical coordinate system. The construction produces an object that represents the position on the screen described by the specified coordinates. The name `Location` should sound familiar. It is part of the thus far unexplained 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:

```
private 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( initialPosition );
```

These changes would not alter the behavior of the program, but they would make it easier to read. A human looking at your program is more likely to understand the purpose of the `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 2.5. Each time the mouse is clicked, this program will draw a pair of thick, perpendicular lines. The first lines drawn will intersect at the upper left corner of the window. With each click, the lines drawn will be placed to the right and below the preceding lines, so that the window is eventually filled with a grid pattern.

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

In its `begin` method, this program creates two `Location` objects that both describe the point at the origin of the coordinate system, the upper left corner of the canvas. One of these objects is associated with the name `verticalCorner` and the other with the name `horizontalCorner`.

```
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 {

    // The corners of the next two rectangles to draw
    private Location verticalCorner, horizontalCorner;

    // Set Locations to position first pair of lines at upper
    // left corner of the canvas
    public 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
    public 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 2.5 An application of the `translate` method

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 2.6.

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 still

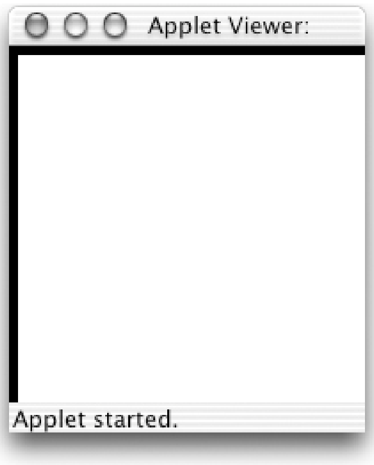


Figure 2.6 Display after one click

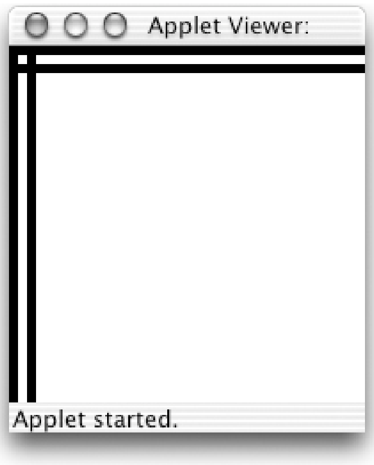


Figure 2.7 Display after second click

appear as shown in Figure 2.6 after they have been performed. The `Location`s will 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 `Location`s. 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 2.7. After these rectangles are created, the last two lines of the method will shift the `Location`s used farther to the right and down the screen, so that the rectangles produced by the next click will appear at the correct locations.

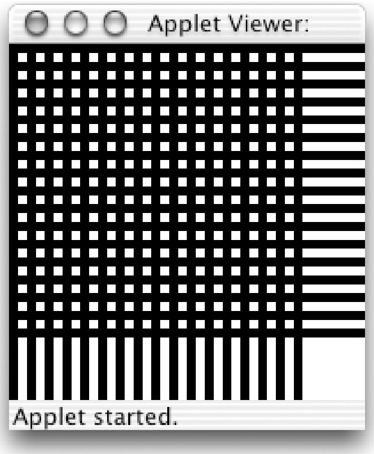


Figure 2.8 Display after many clicks

This process will be repeated each time the mouse is clicked. After about 15 additional clicks, the window will be nearly filled with a grid of lines, as shown in Figure 2.8. Just a few more clicks will complete the process of filling the screen.

2.3 Layering on the Canvas

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

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

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

Then we must construct the desired objects:

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

and finally set their colors appropriately:

```
sky.setColor( new Color( 60, 60, 60 ) );
moon.setColor( Color.WHITE );
cloud.setColor( Color.GRAY );
```

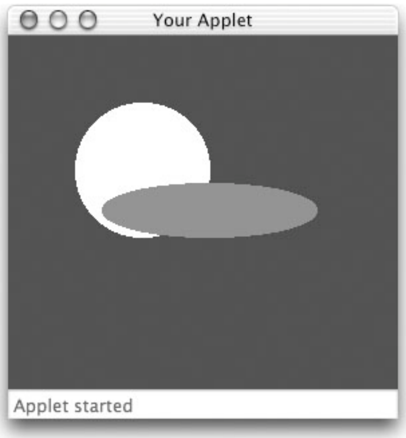


Figure 2.9 Tonight's forecast: Partly cloudy

The parameters "60, 60, 60" provided to the `Color` construction used to set the color of `sky` describe a dark shade of gray.

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

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

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

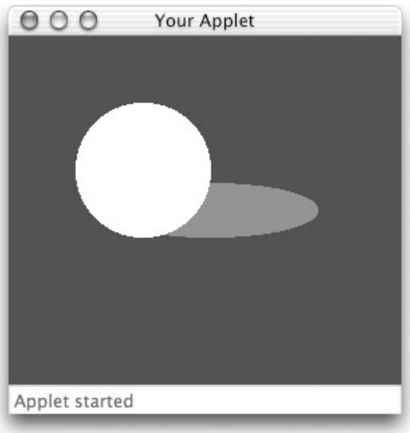


Figure 2.10 Tonight's forecast: Partly moony?

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

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

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

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

or the command

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

the picture displayed would change to look like Figure 2.10.

2.4 Accessing the Location of the Mouse

In the header of every mouse-event-handling method we have included the phrase `Location point` in parentheses after the name of the method. Now that we have explained what a `Location` 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 instance variable declarations, we are free to use any name we want when we declare a formal parameter. There is nothing special about the name `point` (except that we have used it in all our examples so far). We can choose any name we want for the mouse location as long as we place the name after the word `Location` in the header of the method. For example, the program shown in Figure 2.11, 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 );
    }
}
```

Figure 2.11 Using a different parameter name

2.5 Sharing Parameter Information between Methods

Another important aspect of the behavior of formal parameter names like `mousePosition` and `point` is that each parameter name is meaningful only within the method whose header contains its declaration. To illustrate this, consider the program shown in Figure 2.12. This program displays the word "Pressed" at the current mouse position each time the mouse button is pushed, and it displays the word "Released" at the current mouse position each time the mouse is released. The `onMousePress` method in this example is identical to the corresponding method from the `Pressed` example except for the word it displays and the name chosen for its formal parameter. The `onMouseRelease` method is also quite similar to the earlier program's `onMousePress`.

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

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

```
    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

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

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

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

Figure 2.12 A program to record mouse-button changes

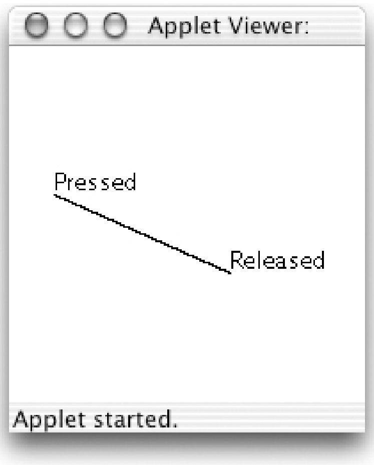


Figure 2.13 Connecting the ends of a mouse motion

in `onMouseRelease` as an error and refuse to run the program. In general, formal parameter names cannot be used to share information between two different methods.

If we want to share information about the mouse location between two event-handling methods, we must use formal parameters and instance variables together. We have already seen that instance variables can be used to share information between methods. In the `RisingSun` example, 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.

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 2.14.

When the mouse is pressed, the assignment

```
firstPoint = pressPoint;
```

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

```
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 2.14 A program to track mouse actions

coordinates (5,5). As soon as you do this, `onMousePress` is invoked and the name `firstPoint` is associated with a `Location` that represents the point (5,5). If you then drag the mouse across the screen, release the mouse button, and then press it again at the point (150,140), `onMousePress` is invoked again and the assignment statement in its body tells Java to associate `firstPoint` with a `Location` that describes the point (150,140). At this point, Java forgets that `firstPoint` ever referred to (5,5). A name in Java may refer to different objects at different times, but at any given time it refers to exactly one thing (or to nothing if no value has yet been assigned to the name).

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

Complex drawing programs provide many tools for drawing shapes, lines, and curves on the screen. We will write a program to implement the behavior of just one of these tools, the one that allows the user to scribble on the screen with the mouse as if it were a pencil. A sample of the kind of scribbling we have in mind is shown in Figure 2.15. The program should allow the user to trace a line on the screen by depressing the mouse button and then dragging the mouse around the screen with the button depressed. The program should not draw anything if the mouse is moved without depressing the button.

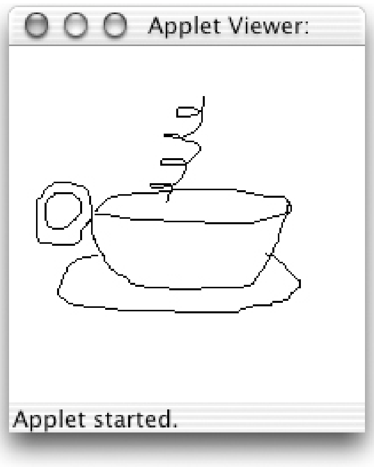


Figure 2.15 Scribbling with a computer's mouse

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;
```

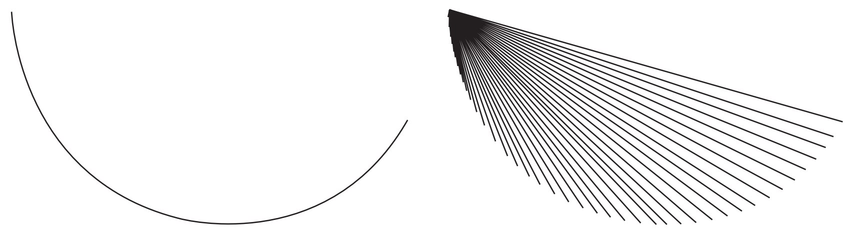
and then write the following definition for `onMousePress`:

```
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` as often as we should. In `onMousePress`, we tell the computer to make this name refer to the point where the mouse is first pressed, and it continues to refer to this point until the mouse is released and pressed again. As a result, as we drag the mouse, all the lines created start at the point where the mouse button was first pressed. Instead, after the first line has been drawn, we always want `previousPosition` to refer to the mouse's position when the last line was drawn. To do this, we must add the assignment statement:

```
previousPosition = currentPosition;
```

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

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

// This program allows its user to draw simple lines on the screen
// using the mouse as if it were a pencil
public class Scribble extends WindowController {

    private Location previousPosition; // Last known position of mouse

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

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

Figure 2.16 A simple sketching program

Exercise 2.5.1

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

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

❖

2.6 Summary

In this chapter we explored the importance of the use of names to refer to the objects our 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.

We learned more about displaying simple graphical objects on our program's canvas and learned how to modify the properties of these objects using mutator methods. In addition, we learned 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.

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.

2.7 Chapter Review Problems

Exercise 2.7.1

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

Exercise 2.7.2

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

Exercise 2.7.3

What is wrong with the following line of code?

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

Exercise 2.7.4

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

Exercise 2.7.5

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

Exercise 2.7.6

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

Exercise 2.7.7

Look at the two pieces of code. Do they produce the same or different outcomes? If the outcomes are different, explain the difference.

a. i.
```
public void onMouseClick ( Location point ){
       new Text ( "Hello.", point, canvas );
}
```
ii.
```
public void onMouseClick ( Location clickPoint ){
       new Text ( "Hello.", clickPoint, canvas );
}
```
b. *Assume* `clickPoint` *is an instance variable of type* `Location` *that has been initialized in a* `begin` *method.*

i.
```
public void onMouseClick ( Location point ){
       point = clickPoint;
       new Text ( "Hi", clickPoint, canvas );
}
```
ii.
```
public void onMouseClick ( Location point ){
       clickPoint = point;
       new Text ( "Hi", clickPoint, canvas );
}
```
❖

2.8 Programming Problems

Exercise 2.8.1

Write a simple program that does the following:

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

Exercise 2.8.2

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

Exercise 2.8.3

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

- *When the program begins there should be a message telling the user "Measles: Click to catch the disease!"*

- *When the user clicks the mouse, a red dot should form at the location where the mouse was clicked. Make the diameter of the dots 5 pixels. The introductory message should no longer be visible.*
- *When the mouse exits the window, the user should be cured. Thus all the spots should disappear and a message saying "You have been cured!" should appear.*
- *When the mouse re-enters the window, the original message should reappear.*

Hint: *Hidden items are also removed from the canvas when you clear it.*

b. *Modify your code from part (a) so that the spot is centered at the point where the mouse was clicked.*

❖