Chapter 2

Working with Numbers

In the preceding chapters, we have used numbers extensively to manipulate graphical objects. They have been used to specify coodinates, dimensions and even colors. While we have used numbers to describe what we wanted to to to various graphical objects, we have not done much of interest with the numbers themselves. Numbers are of critial importance to Java and just as Java provides operations to let us work with graphical objects it provides operations to let us work with numbers. In this chapter we will explore some of these operations. We will see how to obtain numerical values describing properties of existing objects, how to perform basic arithmetic computations, how to work with numeric variables and how to display numeric values.

2.1 Introduction to Accessor Methods

When we perform a construction of the form

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

we combine two numbers to form a single `Location` object. What if we want to do the opposite? That is, what if we have a `Location` object and we want to access the numerical values of the x and y coordinates associated with that location? There are many situations where it would be useful to have this ability.

Suppose, for example, that we decided to change the `RisingSun` program so that rather than having to click the mouse to move the sun, the user could simply drag the mouse up and down and the sun would follow it. The behavior we have in mind is similar to the way in which the scrollbars found in many programs react to mouse movement. If you grab the scroll box displayed in a vertical scrollbar you can move the scroll box up and down by moving the mouse, but you can not move the scroll box to the left or right. Similarly, in the program we have in mind, even if the mouse is dragged about in spirals, the circle that represent the sun should only move straight up and down so that its y coordinate is always the same as the y coordinate of the mouse's current position.

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

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

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

    // A program that produces an animation of the sun rising and setting.
    // The animation is driven by dragging the mouse.
public class ScrollingSun extends WindowController {
    private FilledOval sun;     //  Circle that represents the sun


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

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

}
```

Figure 2.1: Program to make the sun scroll with the mouse

The question is what we should provide as parameter information to the `moveTo` method.

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

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

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

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

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

The complete text of the revised program is shown in Figure 2.1. With the exception of the substitution of the `onMouseDrag` method for the `onMouseClick` method, the only difference between

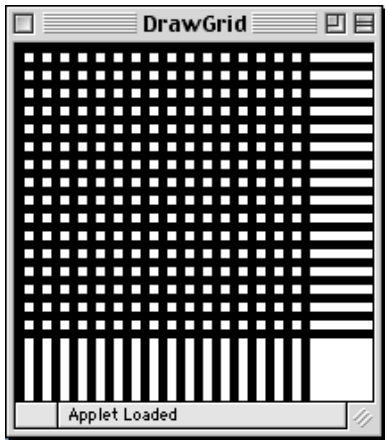

Figure 2.2:

this program and the one shown in Figure 1.7 is that the instructions displayed by the `begin` method have been modified.

Accessor methods are also associated with objects of the graphics classes introduced in the last chapter. The location and dimensions of a graphical object can be accessed using methods named `getX`, `getY`, `getWidth` and `getHeight`. Like the methods discussed above, these accessor methods provide numeric information about an object. In addition, there are accessor methods associated with graphical objects that provide other forms of information. There is a method named `getColor` that returns the `Color` of a graphical object. Similarly, `getLocation` returns a `Location` object describing an object's current position.

2.2 Accessing Numerical Attributes of the Canvas

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

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

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

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

Given that the canvas size is unpredictable, it would be best to write programs that determine the actual size of the canvas while running and adjusts the objects they draw accordingly. We

have seen that the canvas provides a mutator method named `clear`. It also supports two accessor methods named `getWidth` and `getHeight` which allow a program to determine the dimensions of the drawing area. Like the `getX` and `getY` methods associated with `Locations`, these methods do not expect any parameter values. To produce a version of `DrawGrid` that would work correctly in any size canvas, we would simply replace the occurrences of the number 200 in the constructors shown above with appropriate uses of accessor methods to obtain the following code:

```
new FilledRect(verticalCorner, 5, canvas.getHeight(), canvas );
new FilledRect(horizontalCorner, canvas.getWidth(), 5, canvas );
```

We mentioned above that the actual window size used by our programs is determined by a specification written in the language used to construct web pages, HTML. The programs we have been writing are all examples of what are called *applets*. They are Java programs that can be embedded within the contents of a web page. A fragment of HTML that could be used to include our DrawGrid program in a web page is shown below:

```
<applet archive="JavaClasses.jar"
             code="DrawGrid.class"
             width=200 height=200>
</applet>
```

If a web browser was used to visit a web page whose HTML description contained this specification, our DrawGrid program would be displayed in a 200 by 200 pixel rectangle within the larger web browser window. If the person who constructed the web page had instead included the specification

```
<applet archive="JavaClasses.jar"
             code="DrawGrid.class"
             width=400 height=300>
</applet>
```

then the program would be displayed in a 400 by 300 pixel region. Even if the environment you use to write Java programs does not use a web browser to run your programs, it probably does depend on a file containing at least a fragment of HTML like the one shown above to decide how big the window your program runs in should be. The key thing to note is that the individual who writes the HTML gets to specify the width and height of the program's canvas.

2.3 Expressions and Statements

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

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

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

Such an instruction would tell Java to ask the `Location` named `mousePosition` for some information but make no use of the information. Instead, we use accessor methods in places within a program where Java expects the programmer to describe an object or value. For example, an accessor methods can be used on the right hand side of assignment statements to describe the value that should be associated with the name on the left side of the equal sign as in the statement

```
lastY = mousePosition.getY().
```

or to describe a parameter to a construction as in

```
new FilledOval( 10, mousePosition.getY(), 2, 2, canvas);
```

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

This notion of a phrase that describes an object or value is important enough to deserve a name. Such phrases are called *expressions*. We have already seen several different sorts of phrases that Java recognizes as examples of expressions: numeric literals, invocations of accessor methods, constructions, and variable or parameter names.

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

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

In other situations, we have invoked accessor methods to describe numeric values as in

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

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

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

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

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

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

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

The invocation of a mutator method such as:

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

is an example of a phrase that might appear within a Java program that is not an expression. This phrase contains subparts that are expressions, the numeric literals 0 and -1, but is not an expression itself because it does not describe a value. Instead of describing a value, this phrase instructs Java to perform an action. Such phrases are called *instructions* or *statements*. Statements instruct Java to perform actions that either produces output visible to the user or alters the internal state for the computer in a way that will affect the future behavior of the program. The body of each method we define in a Java program must be a sequence of statements.

We have seen three types of statements at this point. The invocation of a mutator method, like the example

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

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

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

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

to place graphics on the canvas. We have already stated, however, that a construction is an expression. Which is it? The answer is both. A construction like the one shown above describes an object. Therefore it can be used in contexts where expressions are required. At the same time, the construction of a graphical object involves the action of changing the contents of the display. Accordingly, the construction by itself can be viewed as a statement.

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

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

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

2.4 Arithmetic Expressions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The complete `ScrollingSun` program using such formulae is shown in Figure 2.3. In most cases, we have simply replaced a number used as an expression by the appropriate formula. The only slight complication is the code to center the text. We can not use the `getWidth` method associated with the `Text` object until it has been constructed. So, when we construct the Text we just use 0 as its x coordinate value. Then, once it exists we use the `getWidth` method to figure out how big it is. Finally, we use `moveTo` to place the `Text` where it belongs.

The arithmetic expressions shown in the preceding examples use only two of the arithmetic operations, subtraction and division. It is also possible to use multiplication and addition. The symbols used to indicate addition, subtraction and division are the standard symbols from mathematics: +, -, and /. Multiplication is represented using an asterisk, *. Thus, to say "2 times the width of the canvas" one would write

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

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

+	addition
-	subtraction
*	multiplication
/	division

2.4.1 Ordering of Arithmetic Operations

Two of the arithmetic expressions used in our revised version of the the rising sun program illustrate an issue a programmer must be aware of when writing such expressions: the rules used to determine the order in which operations are performed. The first of these is the expression

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

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

```
    // A program that produces an animation of the sun rising and setting.
    // The animation is driven by dragging the mouse button.
public class ResizingSun 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( canvas.getWidth()/4,
                            canvas.getHeight() - canvas.getWidth()/4,
                            canvas.getWidth()/2,
                            canvas.getWidth()/2, canvas);

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

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

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

Figure 2.3: Program to make the sun scroll with the mouse

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

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

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

The rule that division is performed before subtraction is an example of what is called a *precedence rule*. When evaluating simple arithmetic expressions, Java follows two basic precedence rules.

1. Perform divisions and multiplications before additions and subtractions. We therefore say that division has higher precedence than addition but that division and multiplication are of equal precedence. Similarly, addition and subtraction are of equal precedence.

2. When performing operations of equal precedence (i.e. additions and subtractions or divisions and multiplications) perform the operations in order from left to right as written.

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

These rules should not be new to you. The same rules are followed when interpreting formulae written in standard mathematics. Remembering them is more important when programming, however, because the way programs are written provides us with fewer means to make the intended interpretation of a formula clear. If we are writing a formula on paper or using a good word processor we can spread the formula over several lines to make things clear. For example, the two formulae shown above could be written as:

$$\frac{width - height}{2}$$

and

$$height - \frac{width}{4}$$

The order in which the operations are to be performed in these expressions is clear even though no parentheses are included. When we are forced to write all the symbols in a formula on one line, as we are in Java and other programming langauges, it is much easier to accidentally write a formula in which the order of evaluation dictated by the precedence rules is different from the order we had in mind.

The good news is that Java does not mind if you use extra parentheses. Therefore, while you should do your best to become familiar with the two precedence rules given above, whenever it will make an expression's meaning clearer, do not hesitate to use parentheses. For example, Java will treat the expressions

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

and

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

as equivalent. So, if you find yourself writing an expresison like the first example and are not quite sure which operator will be evaluated first, feel free to add parentheses like those in the second example to make your intent clear to the computer and to anyone who reads your code later.

2.5 Numeric Instance Variables

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

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

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

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

describes black,

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

describes a dark shade of gray,

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

would be a fairly light shade of gray and

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

is white.

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

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

To use such a name, of course, we must first declare the name and then add assignment statements to ensure that it is associated with the correct number at each point in time as the program executes.

In each instance variable declaration we must precede the name declared by the name of the type of information with which it will be associated. Accordingly, to declare a variable like `brightness`, we need to know the name Java associates with the collection of numeric values.

One might expect Java to use a single name like `Number` or `Numeric` to describe the collection of all numeric values. Instead, Java distinguishes between numbers that include fractional components like 3.14 and .95, and integers like 17 and -45. It uses the name `double` to describe numbers with fractional components and the name `int` to describe integers. In the next section, we will discuss why Java distinguishes between integers and non-integers in this way. For now, we merely observe that the values associated with `brightness` in our program will always be integers. Accordingly, to declare the name `brightness` we could say

```
private int brightness;
```

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

```
brightness = 50;
```

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

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

in `onMouseClick`. This statement tells the computer to take the current value associated with the name `brightness`, add one to it, and then associate the name brightness with the result. The first time the mouse is clicked, `brightness` will be associated with the value 50 specified in the `begin` method. The result of adding 1 to 50 is 51. So, after the assignment

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

is executed, `brightness` will be associated with the value 51. The next time the mouse is clicked, Java will add 1 to the new value of `brightness`, 51, and set it equal to 52. Thus, each time the mouse is clicked, the value of `brightness` will become one greater and the color generated by the contruction

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

will become a little bit brighter.

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

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

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

```
brightness++;
```

The notation

```
brightness--;
```

can also be used to tell Java to reduce the value associated with a numeric variable by one.

With these details we can complete the program. The code is shown in Figure 2.4.

```
    //   A program to simulate the brightening of the sky at sunrise
public class LightenUp extends WindowController {

    private FilledRect sky;   //  background rectangle
    private int brightness;    //  brightness of sky's color
    private FilledOval sun;   //  Circle that represents the sun


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

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

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

Figure 2.4: Using a numeric instance variable

2.6 Initializers

In the preceding sections, we have seen that we must complete two steps before using a variable name to refer to a number or any other type of information. We must include a declaration telling Java that we plan to use the name and telling Java the type of information with which the name will be associated during our program. We must also include an assignment statement associating a particular meaning with the name before using it.

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

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

In the example considered in the preceding section, the initial value of the variable named `brightness` is set to 50 in the `begin` method. Using Java's notation for initialized variable declarations, we could remove this assignment from the `begin` method and rewrite the variable's declaration as

```
private int brightness = 50;            // brightness of sky's color
```

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

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

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

Warning! The predefined name `canvas` is not initialized until just before your `begin` method is invoked. Java handles initialization expressions included in declarations of instance variables before this happens. Therefore, it is generally not safe to use the name `canvas` in any initializer within your `WindowController` class. In particular, you could **not** eliminate the assignments for `sky` and `sun` that appear in the `begin` method of the program in Figure 2.4 by declaring these variables with initializers because the expressions that are required to initialize the variables correctly (i.e. the expressions included in the `begin` method) reference the `canvas`.

2.7 Why double?

In Section 2.5, we mentioned that Java distinguishes between at least two collections of numeric values, `int`s and `double`s. In this section we will discuss the reasons Java makes this distinction and the impact the distinction has on a programmer when working with numbers.

To understand why Java distinguishes integers from other numbers, consider the following two problems. First, suppose you and two companions are stranded in a life boat. Among your supplies, you have 55 gallons of water. You decide that this water should be divided equally among the three of you. How much water does each person get to drink? (Go ahead. Take out your calculator.)

Now, suppose that you are the instructor of a programming course in which 55 students are currently registered and you want to assign the students to 3 afternoon laboratory sections. How many students should be assigned to each section?

With a bit of luck, your answer to the first question was 18 1/3 gallons. On the other hand, even though the same numbers, 55 and 3, appear in the second problem, "18 1/3 students" would probably not be considered an acceptable answer to the second problem (at least not by the student who had to be chopped in thirds to even things out). A better answer would be that there should be two labs of 18 students and a third lab with 19 students.

The point of this example is that there are problems in which fractional results are acceptable and other problems where we know that only integers can be used. If we use a computer to help us solve such problems, we need a way to inform the computer whether we want an integer result or not.

In Java, we do this by choosing to use `int`s or `double`s. For example, if we declare three instance variables:

```
private double gallons;
private double survivors;
private double waterration;
```

and then execute the assignment statements

```
gallons = 55;
survivors = 3;
waterration = gallons/survivors;
```

the number associated with the name `waterration` will be 18.33333... On the other hand, if we declare the variables

```
private int students;
private int labs;
private int labsize;
```

and then execute the assignments

```
students = 55;
labs = 3;
labsize = students/labs;
```

the number associated with `labsize` will be 18. In the first example, Java can see that we are working with numbers identified as `double`s, so when asked to do division, it gives the answer as a `double`. In the second case, since Java notices we are using `int`s, it gives us just the integer part of the quotient when asked to do the division.

Of course, the answer we obtain in the second case, 18, isn't quite what we want. If all the labs have exactly 18 students, there will be one student excluded. We would like do something about such leftovers. Java provides an additional operator with this in mind. When you first learned to perform division in school, you probably had not yet learned about decimal notation. So, you

were taught that the result of dividing 55 by 3 was 18 with a remainder of 1. That is, you were taught that the answer to a division problem had two parts, the quotient and the remainder. When working with integers in Java, the "/" operator produces the quotient. The percent sign can be used as an operator to produce the remainder. Thus, while 55/3 will yield 18 in Java, 55%3 will yield 1. Used in this way the percent sign is called the *mod* or *modulus* operator. We could use this operator to improve our solution to the problem of computing lab sizes by declaring an extra variable

```
private int extraStudents;
```

and adding the assignment

```
extraStudents = students % labs;
```

2.7.1 Arithmetic with `doubles` and `ints`

The distinction between `doubles` and `ints` in Java is a feature intended to allow the programmer to control the way in which arithmetic computations are performed more precisely. As a beginning programmer, however, you should be warned that this feature may produce some strange surprises.

Suppose that we want to construct an oval whose width is three quarters of the width of the canvas and we know that the width of the canvas is 300. According to the rules of normal arithmetic, it should not matter whether we say

```
300*3/4
```

or

```
300*(3/4)
```

In Java, however, the first version will produced the expected result, 225, while the second version will produce the number 0. In the first example, following its precedence rules leads Java to first multiply 300 by 3 yielding 900 and then to divide by 4 obtaining 225. In the second example, Java first divides 3 by 4. Since both of these numbers look like `ints` to Java, it decides the correct answer is 0 with a remainder of 3 and returns the quotient, 0, as the result of the division. Then, 300 times the result of this division, 0, yields 0.

To avoid unexpected results like this, one must understand how Java decides when we desire an integer result and when we would prefer to work with numbers with fractional components.

The first rule is simple. When we write numbers out explicitly, Java decides whether the number is an `int` or a `double` based on whether it contains a decimal point. Therefore, 3 is an `int` but 3.0 is a `double`. This can make a big difference. If we rewrite the second example above as

```
300.0*(3.0/4.0)
```

it will produce 225.0 rather than 0 as its result.

Second, one must understand that while Java distinguishes `ints` and `doubles`, it recognizes that they are related. In particular, in contexts where one should technically have to provide a `double`, Java will allow you to use an `int`. This was already illustrated in the example above where we declared an instance variable

```
private double gallons;
```

and then wrote the assignment

```
gallons = 55;
```

The number 55 is identified by Java as an `int`. The variable is declared to be a `double`. If a variable were declared to refer to a `Location`, Java would reject any assignment that attempted to associate the variable with anything other than a `Location`. Java will, however, accept an assignment in which an `int` is assigned to a variable that is supposed to refer to a `double`.

Java is willing to convert `int`s into `double`s because it knows there is only one reasonable way to convert an `int` into a `double`. It simply adds a ".0" to the end of the `int`. On the other hand, Java knows that there are several ways to convert a `double` into an `int`. It could drop the fractional part or it could round. It can not tell the correct technique to use without understanding the context or purpose of the program. Java refuses to convert a `double` into an `int` unless explictly told how to do so using mechanisms that we will discuss later. Therefore, given the instance variable declaration

```
private int students;
```

Java would reject the assignment

```
students = 18.333;
```

as erroneous. It would also reject the assignment

```
students = 55.0;
```

Even if the only digit that appears after the decimal point in a numeric literal is 0, the presence of the decimal point still makes Java think of the number as a `double`.

Java's willingness to convert `int`s to `double`s is also a factor in understanding how it decides whether the result of an arithmetic operation should be an `int` or a `double`. If both operands of an arithmetic operator are of the same numeric type, then Java will produce a result of this type. Thus, adding two `int`s produces an `int` and dividing a `double` by another `double` yields a `double`. The interesting question is what does Java do if one operand is an `int` and the other a `double` as in

```
3/4.0
```

The answer is that when Java sees an operator with two operands of different types, it tries to convert one of the operands to the other type. Since it likes to convert `int`s to `double`s but not the other way around, if an operation involves one `int` and one `double`, Java converts the `int` into a `double` (by adding .0) and then performs the operation yielding a result that is also a `double`. Accordingly, 3/4.0 would evaluate to 0.75.

The following table summarizes Java's procedure for determining the type of result an operator should produce given the types of its two operands. The type of the first operand determines the row of the table that applies. The type of the second operand determines the column of the table used. The contents of the cell where the appropriate row and column meet specifies the type of the result. Note that the only case where the result produced is an `int` is when both operands are integers.

	`int`	`double`
`int`	`int` result	`double` result
`double`	`double` result	`double` result

This means that once you introduce a number which is a `double` into a computation, you are likely to end up with a result that is a double. Even if the result has a fractional part that is 0, Java will not automatically convert this `double` into an `int`. This can lead to unexpected errors in your program. For example, several of the accessor methods associated with graphical objects including `getX`, `getY`, `getWidth`, and `getHeight` return values that are `double`s. Accordingly, the expression in the assignment statement

```
midPointX = ( currentPosition.getX() + previousPosition.getX() ) / 2
```

which might be used to compute the x coordinate of the midpoint of the line between two points on the canvas, would produce a `double`. This means that if you had defined the variable to hold the result as

```
private int midPointX;
```

Java would reject the assignment as an error. Instead, the variable would have to be declared as

```
private double midPointX;
```

2.7.2 Selecting a numeric type

The examples and rules above suggest that a reasonable guideline for dealing with the difference between `int`s and `double`s is to use `double`s whenever possible. Most of the surprising examples you are likely to encounter involve integer division. In addition, Java will sometimes refuse to accept an assignment of a value to an variable declared as an `int`, but will always allow you to assign any numeric value to a variable declared as a `double`, even if it has to turn an `int` into a `double` to make the assignment possible.

Of course, there are contexts where you will have to use `int`s. First, as suggested in the introduction to this section, there are programming problems where the only acceptable results are integers. You will have to use `int`s in such programs. Also, there are certain contexts where Java will demand an `int`. The three numbers provided in a `Color` construction, for example, must be `int`s.

2.7.3 Why are reals called `double`?

As a final topic in this section, we feel obliged to try to explain why Java chooses to call non-integers `double`s rather than something like `real` or `rational`. The explanation involves a bit of history and electronics.

To allow us to manipulate numbers in a program, a computer's hardware must encode the numbers we use in some electronic device. In fact, for each digit of a number there must be a tiny memory device to hold it. Each of these tiny memory devices costs some money and, not long ago, they cost quite a bit more. So, if a program is only working with small numbers, the programmer can reduce the hardware cost by telling the computer to only set aside a small number of memory devices for each number. On the other hand, when working with larger numbers, more memory devices should be used.

On many machines, the programmer is not free to pick any number of memory devices per number. Instead only two options are available: the "standard" one, and another that provides **double** the number of memory devices per number. The name of the Java type `double` derives from such machines.

While the cost of memory has decreased to the point where we don't need to worry that using `double`s might increase the amount of hardware memory our program uses, the name does serve to point up an important aspect of computer arithmetic. Since each number represented in a computer is stored in physical devices, the total number of digits stored is always limited.

In the computer's memory, numbers are stored in binary. Thirty one binary digits are used to store the numeric value of each `int`. One additional digit is used to encode the sign. As a result, the values that can be processed by Java as `int`s range in value from -2,147,483,648 ($= -2^{31}$) to 2,147,483,647 ($= 2^{31} - 1$). If you try to assign a number outside this range to an `int` variable, Java is likely to simply throw away some of the digits yielding an incorrect result. If you need to write a program that works with very large integers, there is another type that is limited to integer values but that can handle numbers with twice as many digits. This type is called `long`. There is also a type named `short` that uses half as much memory as `int` to represent a number. The examples in this text will not use `long` or `short`.

The range of numbers that can be stored as `double` values is significantly larger than even the `long` type. The largest `double` value is approximately $1.8x10^{308}$ and the smallest `double` is approximately $-1.8x10^{308}$. This is because Java stores `double` values as you might write numbers in scientific notation. It rewrites each number as a value, the mantissa, times 10 raised to an appropriate exponent. For example, the number

```
$32,953,923,804,836,184,926,273,582,140,929.584,289$
```

might be written in scientific notation as

```
$3.295,392,380,483,618,492,627,358,214,092,958,428,9 x 10^31$
```

Java, however, does not always encode all the digits of the mantissa of a number stored as a `double`. The amount of memory used to store a `double` enables java to record approximately 15 significant digits of each number. Thus, Java might actually record the number used as an example above as

```
$3.295,392,380,483,618 x 10^31$
```

This means that if you use numbers with long or repeating sequences of digits, Java will actually be working the approximations of the numbers you specified. As a result, the results produced will also be slightly inaccurate. Luckily, for most purposes, 15 digits of precision is sufficient.

There is one final aspect of the range of `double` values that is limited. First, the smallest number greater than 0 that can be represented as a `double` is approximately $5x10^{-324}$. Similarly, the largest number less than 0 that can be represented as a `double` is approximately $-5x10^{-324}$.

2.8 Displaying Numeric Information

We have seen how we can use the computer's ability to work with numbers to produce better drawings on the computer's screen. Sometimes, however, it is the numbers themselves rather than any drawing that we really want to see. The main purpose of many computer programs is to perform numerical calculations. Such programs are used to determine your taxes, determine your GPA, estimate the time required to travel from one point to another and solve many other kinds of numerical problems. Such programs often display the numbers they compute rather than a drawing on the screen. Even programs that are not primarily focused on numerical computations often need to display numerical information. For example, a word processor might need to display the current

page number. With all these examples in mind, in this section we will describe two mechanisms in Java that can be used to display numerical information.

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

2.8.1 Displaying numbers as `Text`

We have already introduced one mechanism that can be used to display numbers on the screen. We just didn't mention that it could be used with numbers at the time. In the very first program in Chapter 1, we used a construction of the form:

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

and explained that the Text construction requires four parameters:

- the information to be displayed,
- x and y coordinate values specifying the upper left corner of the region in which the information should be displayed, and
- the canvas.

In all the examples of Text constructions we have seen thus far, the first parameter has always been a sequence of characters surrounded by quotes. In fact, however, Java will accept many different types of information, including numerical information, for the first parameter of a Text construction and will display whatever information is provided in textual form. "Tex" in Java is not just letters, but any information that can be encoded using the symbols on a typical keyboard including the digits and punctuation marks.

In particular, if we define a variable

```
private int theCount = 1;
```

with the intent of using it to count up from one, and then we execute a construction of the form:

```
new Text( theCount, 100, 100, canvas);
```

Java will display the current value of the variable `theCount`, 1, at the point (100,100) in the program's window as shown in Figure 2.5.

Of course, displaying the number 1 isn't counting. The program we want to construct should start by displaying 1, but the first time the user clicks the mouse, we want to replace 1 by 2. On the next click, we want to replace 2 by 3 and so on.

In case you didn't notice, one of the examples considered in this chapter already demonstrates how to teach a computer to count. In the version of the rising sun program in which the background became brighter as the sun rose, the operations we performed on the variable named `brightness` essentially told the computer to count upwards starting at 50. The instruction that we used to progress through the different values of `brightness` was

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

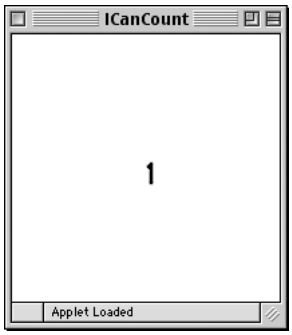

Figure 2.5: A computer counting program takes its first step

A very similar assignments statement involving the variable `theCount`:

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

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

Such a counting program is shown in Figure 2.6. The body of the `onMouseClick` method shown in this figure uses the assignment statement shown above to associate the next counting number with `theCount` each time the mouse is clicked and the `setText` method to update the number displayed each time the value of `theCount` is changed. `setText` was described briefly in the previous chapter, but this is the first time we have used it in an example. It expects a single parameter, the new information to be displayed. Like the first parameter expected in a `Text` construction, this information can be a quoted sequence of character or a numeric value or just about any other form of information we might want to display in textual form. The statement

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

included in the `onMouseClick` method tells Java to change the information displayed by the `Text` object named `countDisplay` that was created in the `begin` method.

An alternative to using `setText` that we could have used in this program is to clear the canvas and then construct a new `Text` object displaying the new value of `theCount`. Construction of a new object is a fairly time consuming process for the computer. When possible, it is better to reuse an existing object rather than create a new one. Accordingly, in an example like this it is preferrable to use `setText`.

2.8.2 Using `System.out.println`

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

All the Java output displayed by the Java programs we have considered so far appears in the window associated with the name canvas. These programs can, however, display output in another window provided by the Java system. There is no special name that can be used to refer to this window from within your program. It is simply known as the *console window*.

The console window is more limited than the `canvas` in that it can only be used for text. On the other hand, it is more convenient for the display of text than the `canvas`.

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

    //  A program to count as high as you can click.
public class ICanCount extends WindowController {

    private int theCount = 1;     //  how high we have counted
    private Text countDisplay;    //  current screen display of count

      //  Create the Text to display the current count
    public void begin() {
        countDisplay = new Text( theCount, 100, 100, canvas );
    }

      //  Increase the count with each click
    public void onMouseClick(Location point) {
        theCount = theCount + 1;
        countDisplay.setText( theCount );
    }
}
```

Figure 2.6: A simple counting program.

To tell Java to display information in the console window you use a method named `System.out.println`. This method takes a single parameter specifying the information to be displayed. Anything that could be used as a parameter to the `setText` method or as the first parameter of a `Text` constructor can be used as a parameter to `System.out.println`. In particular, you can certainly use either a quoted sequence of characters or a numeric value.

A revised version of our counting program that uses the Java console to display the values as it counts is shown in Figure 2.7. The only text this version displays on the canvas is a message telling the user to click in order to make the program count. This will be displayed instead of the number 1 when the program first starts. The first time the user clicks, 1 is placed in the Java console by the `System.out.println` in the `onMouseClick` method. Each succeeding click will place another value in the console window.

When `System.out.println` is used, you do not have to provide coordinates to specify where the text should be displayed. The Java console window displays the information you provide to `System.out.println` much as text might be displayed in a word processor's window. Each time your program executes a `System.out.println`, the text specified is placed after any text that had been placed in the Java console earlier. Once the window fills up, the older text scrolls off the top of the window leaving the newer lines visible. A scroll bar is provided so that a person running your program can look at the older items if desired. Figure 2.8 shows how both the canvas and the Java console window might look after this program is run and its user clicks 25 times.

2.8.3 Displaying `doubles`

We have hinted in the preceding discussion that the primitives we have used to display numerical information on a computer's screen are actually flexible enough to display information of many

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

    // A program to count as high as you can click.
public class ICanCount extends WindowController {

    private int theCount = 0;     //  how high we have counted

      //  Create the Text to display the current count
    public void begin() {
        new Text( "Click to make me count", 40, 100, canvas );
    }

      //  Increase the count with each click
    public void onMouseClick(Location point) {
        theCount = theCount + 1;
        System.out.println( theCount );
    }
}
```

Figure 2.7: Counting in the Java console

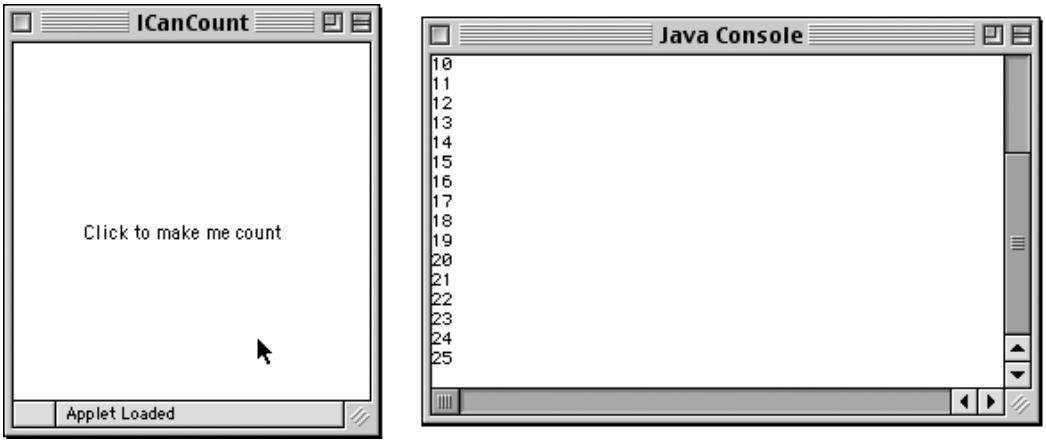

Figure 2.8: Counting using the Java console

types. At this point, however, we have only used them to display quoted strings of text and integers. In this and the following section, we will explore some of the other possibilities.

First, and most simply, these primitives can be used to display numbers of the type `double` just as they are used to display `int`s. For example, if we modify the last version of the counting program by simply changing the declaration of `theCount` to read

```
private double theCount = 0;     //  how high we have counted
```

the program will still work. The only difference is that the values will include a decimal point. Thus, the first value output will be "1.0" rather than "1".

Things get a bit more interesting when large values of type `double` are displayed as `Text` or using the `System.out.println` primitive. Suppose rather than having our program count by ones, we have it list off the powers of 10. That is, on the first click the program will display 10. It will display 100 on the second click, 1000 on the third click, 10000 on the fourth click and so on. The changes required are quite simple. We will stick with the name `theCount` (even though `thePower` might be more appropriate. The new declaration for this variable will be:

```
private double theCount = 1;
```

The only other change we will make is to replace the assignment statement in `onMouseClick` that added 1 to `theCount` with the assignment

```
theCount = theCount * 10;
```

which will compute the next power of 10 by multiplying the last value by 10.

If we run this program and click 10 times, the following ouput will appear in the Java console

```
10.0
100.0
1000.0
10000.0
100000.0
1000000.0
1.0E7
1.0E8
1.0E9
1.0E10
```

The first six lines are as you probably expected, but what does "1.0E7" or "1.0E8" mean.

The strange items Java has displayed are examples of Java's version of scientific notation. Where Java displays 1.0E7, you probably would have expected to see 10000000.0. This value is ten million or 10^7. The "E7" in the output Java produce is short for "times ten raised to the power 7". The "E" stands for "exponent". In general, to interpret a number output in this form you should raise 10 to the power found after the E and multiply the number before the E by the result. For example, the table below shows some examples of numbers written in this notation and the standard forms of the same values.

E-notation	Standard Representation
1.0E8	100,000,000
1.86E5	186,000
4.9E-6	.0000049

Not only does Java display large numbers using this notation when you use `System.out.println` or `Text` objects, it also recognizes numbers typed in this format as part of a program. So, if you really like scientific notation, you can include include a statement like

```
avogadro = 6.022E26;
```

in a Java program.

2.8.4 Mixing text and numbers

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

When specifying the information to be displayed in a `Text` object or on the Java console, we can use the "+" operator to combine quoted text with numeric information. Suppose, for example, that we wanted our counting program to display a message like "You have clicked 3 times" instead of just displaying 3 on the third click. We could accomplish this by replacing the command

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

with the command

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

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

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

```
You have clicked3times
```

instead of displaying

```
You have clicked 3 times
```

as desired.

It is also sometimes important to be aware of how Java decides when a "+" means addition and when it means to simply stick pieces of text together. For example, if the value of `theCount` is 10, then the command

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

will display the message

```
You have clicked 11 times
```

on the screen, while the command

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

will produce the message

```
You have clicked 101 times
```

This is because, in the version with parentheses around "`theCount + 1`" Java has to do the "+" operation within these parentheses first. Both operands of this "+" operator are numbers, so Java does addition yielding the number 11. Without the parentheses, Java processes the "+" operators in order from left to right. The first operand to the first "+" is a quoted string, so Java performs concatenation sticking the textual representation of the value of `theCount`, "10", together with the quoted text. The result of this first operation is then treated as the first operand of the next "+" operation. Since this first operand is text, Java now interprets the second "+" as another concatenation operator so it just sticks a "1" on the end of the text rather than performing a numeric addition.

2.9 Numbers are not Objects

You may have noticed that there are many similarities between the use of numbers in Java and the use of other pieces of information such as `Location`s, `Color`s, `FilledRect`s or any of the other graphical object types we have presented.

For example, the rules for associating and using a name to refer to a piece of information within a program are identical whether the name involved refers to an `int`, a `double`, a `Location` or any of the other types of information we have considered. Each name to be used must first be declared. The forms of the declarations used for numeric and non-numeric variables are identical. The syntax of the assignment statements used for numeric and non-numeric variables are also identical.

In other respects, however, there are significant differences between the way Java treats numbers and the other types of information we have considered. One example is the means Java provides to let us first introduce a specific piece of information in a program. For types other than int and double, one uses constructions. To describe the `Color` light blue we might say:

```
new Color(255, 100, 100)
```

and to describe the origin of the coordinate system we might say:

```
new Location( 0, 0)
```

For numeric types, on the other hand, Java provides the programmer with the ability to simply write *constant*s that directly describe the desired value. For example, a programmer would simply write

```
3
```

rather than

```
new int(3)
```

to describe the integer value 3 within a program.

This difference is partly a matter of convenience. Numbers are used so frequently in programs that it is important for a programming language to provide as simple a means as possible to let a programmer include numeric information. At the same time, this difference represents a more fundamental aspect of the way Java views the numeric types. Numbers, in Java's view, are in some sense less transient than the other kinds of information we have considered. Java doesn't let us say

```
new int( 3 )
```

because in Java's view 3 already exists before you use it. It would make no sense to Java to talk about making a new 3. How would the new 3 differ from the old 3?

In general, Java refers to pieces of information that we construct using `new` as objects and the types composed of such items are called *object types* or *classes*. [1] Pieces of information that are described using constants like "3" rather than constructions are called *values* and types composed of values are called *primitive types*. The underlying distinction here is not just between numeric and non-numeric types. In the next chapter, we will see another primitive type called `boolean` whose values are not numeric. Like the numeric types, there are no constructions for `boolean`s. Instead, Java provides constants to refer to `boolean` values.

The notion that in Java's view values are in some sense more permanent than objects becomes even clearer when we consider another difference between the way Java handles values and objects: the means Java provides to perform operations on values and objects.

When we want to apply an operation to an object, we use Java's notation for method invocation. For example, if `someRect` is a variable name associated with a `FilledRec`, we might say

```
someRect.setWidth(300);
```

to make the rectangle bigger. If Java wanted to make the handling of numbers and other values consistent with the way objects are manipulated, we might similarly say

```
numVar.add(10)
```

to describe the number 10 bigger that the current value associated with an integer variable named numVar. As we have seen, of course, this is not how we apply operations to numeric values. Instead, to perform a numeric operation Java lets us write a formula using notation very similar to standard mathematices. In particular, we use the operators +, -, * and /. For example, we could write the formula:

```
numVar+10
```

to describe the number 10 bigger than the current value associated with numVar.

Clearly, the designers of Java were trying to be nice to future Java programmers when they decided to make it possible to use familiar mathematical notation to describe operations on numbers. The difference between how Java treats numbers and the other types we have considered, however, represents more than a syntactic difference.

When we introduced method invocations, we distinguished between two types of methods: mutator methods and accessor methods. Mutator methods change an object. The operators we can apply to values, on the other hand, don't change the values to which they are applied. Instead, they produce distinct values.

We can appreciate this distinction better by considering a few examples. First, suppose that we declare two variables to refer to filled rectangles

[1] In a few chapters, you will discover there is one major exception to this rule, the class of objects provided to manipulate textual data, the class `String`.

```
FilledRect firstVar;
FilledRect otherVar
```

and then assigned values to these variables as follows:

```
firstVar = new FilledRect( 25, 25, 100, 100, canvas);
otherVar = firstVar;
```

After this is done, both variables refer to the same object, a 100 by 100 pixel filled rectangle displayed near the upper left corner of the program's canvas. Now suppose that we decide we want to make this rectangle 50 pixels wider. We could say

```
firstVar.setWidth(150);
```

Doing so actually changes the already existing rectangle. It does not make a new rectangle. It just makes the existing rectangle bigger. So, after telling Java to set the width of `firstVar`, if we ask it to show us the width of the rectangle named `otherVar` by executing

```
new Text( otherVar.getWidth(), 50, 180, canvas);
```

the number 150 will appear on the screen. Even though we never told Java to change `otherVar`, because the object it refers to was changed through another name associated with the same object, the object associated with `othervar` will have changed.

By contrast, consider what happens if we perform as similar a sequence of operations as possible using a pair of `int` variables instead of `FilledRects`. In particular, assume that in some other program we declare

```
int firstVar;
int otherVar;
```

and execute the assignments

```
firstVar = 100;
otherVar = firstVar;
```

Now, if we want to increase the value associated with `firstVar` by 50 we could say

```
firstVar = firstVar + 50;
```

Before this assignment is executed, the name `firstVar` is associated with the value 100. Evaluating the expression `firstVar + 50` therefore produces the value 150, but it doesn't change 100 or `firstVar` into 150 in any sense. While the `setWidth` method modifies the object to which is is applied, the additon operator produces a new value without in any way modifying its operands. Java then completes the assignment command by associating the value the expression produced, 150, with the variable `firstVar`.

In the rectangle example, the change in the rectangle made by `setWidth` changed an object refered to by both the variable names. So, after the assignment, the object associated with `otherVar` had a new width. Because numbers are values rather than objects, the change made in the int variable `firstVar` will have no effect on `otherVar`. `otherVar` will still refer to the same, unmodified value with which it became associated when the assignment

```
otherVar = firstVar;
```

was executed, the number 100.

In general, changes made by an assignment to a numeric variable only effect the variable explitly mentioned in the assignment. Changes made by applying a mutator method to an object, however, will be visible through all names associated with the object modified.

2.10 Naming numeric constants

In chapter 1, we introduced the comment. Comments are a rather interesting construct precisely because they have no effect on how the programs that contains them actually behave. As far as the computer is concerned, comments are useless. Nevertheless, a special notation is included in Java (and in almost every other programming languages), to enable us to include these "useless" comments in our program. This reflects the fact that while it is clearly important that a computer be able to understand any program you write, it is also very important that your programs be as easy to understand as possible for human readers.

Comments are just one mechanism Java provides to help you improve the readability of your programs. Another feature of Java that we have been using to maximize the readability of our example code is the ability to add blanks and empty lines to a program without effecting its interpretation by the computer. Careful indenting of code so that its physical appearance reflects its logical organization can be an important aid to an individual trying too understand the instructions.

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

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

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

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

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

By now, you have seen this instruction often enough that you may recognize it and know what it is for. It is the construction that creates the circle representing the sun in our rising sun example. Suppose, however, that you encountered this construction while reading through a complex Java program composed of thousands of lines of code. How would you guess the purpose of the program's author? How could you understand the significance of the number 50, 100 and 150 that appear in the statement?

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

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

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

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

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

To avoid this problem, Java provides a mechanism through wich the programmer can ensure the reader that the initial value assigned to a variable in its declaration will not be changed anywhere else in the program. To do this, the programmer simply adds the word "final" to the declaration after the word "private". The declaration above would then appear as

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

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

```
sunSize = 200;
```

it would be reported as an error to the programmer and Java would refuse to run the program.

There are two conventions followed by most Java programmers when using `final` in declarations. First, so that it is easy to identify names with fixed values when reading a program, such names are usually composed of all upper case letters. Second, because doing so may in some cases improve program efficiency it is customary to add the modifier `static` to declarations that contain the modifier `final`. So, following these conventions, the declarations of our constants would be rewritten as

```
    //  Constants that determine position and size of the sun
private final int SUNCORNERX = 50;
private final int SUNCORNERY = 150;
private final int SUNSIZE = 100;
```

Of course, the same use of upper case letter would have to appear in uses of these names like

```
new FilledOval ( SUNCORNERX, SUNCORNERY, SUNSIZE, SUNSIZE, canvas );
```

2.11 Handy sources of numeric information

There are a number of features of the libraries provided by Java and the library provided with this text that can be used to generate useful numerical information. Several of these features are described in this section.

2.11.1 Random number

"Pick a number. Any number. ..."

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

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

Suppose that you wanted to write a program to simulate some board game in which at each turn the player rolls two dice. Our class for generating random integers can be used to create a Java object that behaves just like a single die.[2] To illustrate the use of this class, we will construct a simple program that simulate the rolling of a pair of dice each time the mouse is clicked.

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

```
private RandomIntGenerator die;
```

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

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

In general, the first parameter value determines the smallest value that should ever be produced by the random number generator while the second number specifies the largest value. We could include this construction in our program's `begin` method or as an initializer in the variable declaration as shown below

```
private RandomIntGenerator die = new RandomIntGenerator( 1, 6 );
```

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

```
die.nextValue()
```

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

The complete code of a simple program to simulate rolling two dice is shown in Figure 2.10. A sample of the program's output is shown in Figure 2.9. Note that even though the program

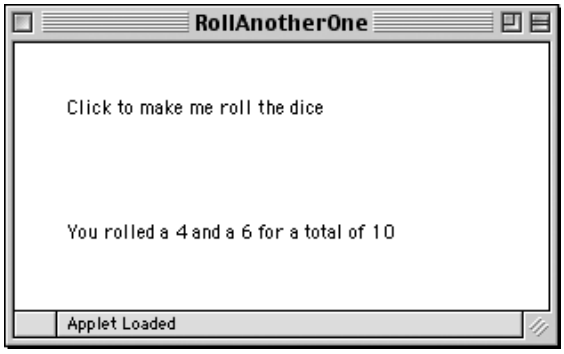

Figure 2.9: Sample messaged drawn by dice simulation program

simulates the rolling of two dice it only uses a single random number generator named `die`. As long as we have a created a single random number generator that generates values in the desired range, we can (and should) use it over and over again whenever we need a random number selected from that range. In the example, we therefore use the `nextValue` method of `die` to determine the values seen on the first die (`roll1`) and the second die (`roll2`).

As mentioned above, there is another randon number generator class provided for situations where you need random values including fractional parts. This class is named `RandomDoubleGenerator`. The class `RandomDoubleGenerator` behaves just like `RandomIntGenerator` except that

- the parameters used in a `RandomDoubleGenerator` construction to specify the range from which numbers should be selected can be `double`s.
- the values returned when the `nextValue` method of a `RandomDoubleGenerator` is invoked will be `double`s.

2.11.2 What time is it?

There are many signs that computers keep track of the time. Most likely, the computer you use displays the current time of day somewhere on your screen as you work. Your computer can tell you when each of your files was created and last modified. While your web browser downloads large files, it probably displays an estimate of how much longer it will take before the process is complete.

To support all the ways in which time is used in programs, Java provides a sophisticated collection of primitives that allow your program to determine the hour, the year, the day of the week or just about anything else you might imagine that is related to measuring time. These mechanims provides more than you need at this point. Accordingly, we have included in our library a simple operation named `getTime` which you can use to ask "What time is it?"

Unfortunately, this question isn't quite as simple as it seems. The `getTime` operation returns a `double` telling you what time it is. For example, if you run a program containing the instruction

```
System.out.println("The time is now " + getTime() );
```

Java might produce the following answer:

[2] A die is also nothing more than a single dice. That is, the English word for the cute little square things you roll while playing many board games has a non-standard plural form. The plural form is dice. The singular form is die.

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

    // A program to simulate the rolling of a pair of dice.
public class RollAnotherOne extends WindowController {

        // Coordinates to determine positions of text displayed
    private final int TEXTX = 30;
    private final int PROMPTY = 30;
    private final int RESULTY = 100;


        // The object that represents a single die
    private RandomIntGenerator die =  new RandomIntGenerator( 1, 6);

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

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

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

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

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

Figure 2.10: Simulating the rolling of a pair of dice

```
The time is now 1.012233623888E12
```

Even if Java displayed the strange number shown above without using scientific notation the result:

```
The time is now 1012233623888
```

would still be mysterious. Apparently, Java uses a different system for telling time than most of us!

Whenever we use a number to answer the question "What time is it?" we are using the number to describe how long it has been since some fixed reference time. If you asked when this book was written and we answered 1422, our answer would appear to be nonsense if you assumed we were using the Gregorian calendar in which times are measured relative to the year in which Christ was born (at least approximately). In fact, however, the number 1422 describes when our book was written based on the Islamic calendar which measure time from the migration of the Prophet and his followers from Mecca to Medina.

Java isn't very religious about time, so it measures time from midnight on January 1, 1970. Also, since Java sometimes needs to be very precise about what time it is, Java measure time in milliseconds (i.e. thousandths of a second). The number shown above, 1012233623888, tells you exactly how many milliseconds passed from midnight, January 1, 1970 until the moment at which the Java instruction shown above produced the output

```
The time is now 1012233623888
```

The `getTime` operation would be very awkward to use if you wanted to display the current time of day in a form a human could understand. Fortunately, it is a very appropriate tool for measuring short intervals of time within a program. This is an ability we will need in many programs in the following chapters. To use `getTime` to measure an interval, we simply ask Java for the time at the beginning of the interval we need to measure and then again at the end. The length of the interval can then be determined by subtracting the starting time from the ending time.

As an example of such a use of `getTime`, a program that will measure the duration of a click of the mouse button is shown in Figure 2.11. The program first uses `getTime` in the `onMousePress` method. It assigns the time returned to the variable `startingTime` so that it can use the value later after the mouse has been releases. In `onMouseRelease`, the program computes the difference between the time at which the mouse was pressed and released using subtraction. Then, in order to display the result in units of seconds rather than milliseconds, it divides by 1000. A sample of what the program's output might look like is shown in Figure **??**.

2.11.3 Sines and Wonders

If you review all the things you have learned to do with numbers in Java, you should be unimpressed at best. Think about it. For just $10 you could go to almost any store that sells office or school supplies and buy a pocket calculator that can compute trigonometric functions, take logarithms, raise any number to any power and do many other complex operations. On the other hand, with Java and a computer that probably costs 100 times as much as a calculator, all you have learned to do so far is add, subtract, divide and multiply. In this section we will improve this situation by introducing a collection of methods that provide the means to perform more advanced mathematical calculations.

```
    // A program to measure the duration of mouse clicks.
public class ClickTimer extends WindowController {

        // coordinates where messages should be displayed
    private final static double TEXTTOP = 50;
    private final static double TEXTLEFT = 30;

        // When the mouse button was depressed
    private double startingTime;
        // Used to display length of click
    private Text message;

        // Create the Text to display the current count
    public void begin() {
        message = new Text( "Please depress and release the mouse",
                            TEXTLEFT, TEXTTOP, canvas );
    }

        // Record the time that the button is pressed
    public void onMousePress(Location point) {
        startingTime = getTime();
    }

        // Display the duration of the latest click
    public void onMouseRelease(Location point) {
        message.setText("You held the button down for " +
                          ( getTime() - startingTime)/1000 + " seconds"     );
    }
}
```

Figure 2.11: Java program to measure mouse click duration.

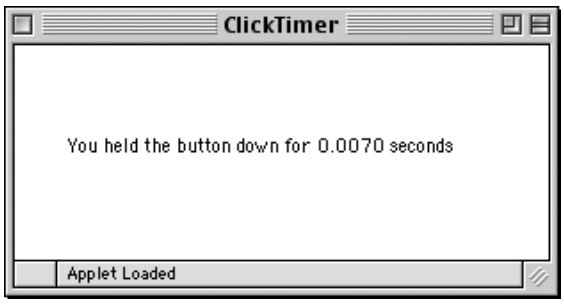

Figure 2.12: Sample output of ClickTimer program shown in Figure 2.11.

Roots, logs and powers

A common arithmetic operation available on most calculators is the taking of a square root. In Java, this is done using a method named `Math.sqrt`. `Math` is a class in which many interesting mathematical methods have been collected. `Math.sqrt` is a method that takes any numeric value as a parameter and returns a `double` approximating the number's square root. Thus, if we created a `Text` object named `message` and later executed the instruction

```
message.setText("The square root of 2 is " + Math.sqrt(2) );
```

the message

```
The square root of 2 is 1.4142135623730951
```

would appear on the canvas.

Of course, the parameter to a method like `Math.sqrt` can be described using any expression that produces a numeric result and the invocation of `Math.sqrt` can be used as a sub-part of a larger arithmetic expression. For example, if the variable names `a`, `b`, and `c` were associated with the coefficients of a quadratic polynomial, we could display a solution to the associated quadratic equation by translating the quadratic formula (simplified to produce only the largest answer):

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

into Java. The result would look like:

```
( -b + Math.sqrt( b*b - 4*a*c) ) / (2 * a)
```

Note that all the parentheses included in this Java fragement are required to ensure that the operations specified are performed in the desired sequence.

Java provides several other methods for performing operations related to raising values to powers:

`Math.pow(a,b)` raises the number `a` to the power b and returns the result,

`Math.exp(b)` raises the constant e (≈ 2.718) to the power `b`,

`Math.ln( a )` returns the natural logarithm (i.e. the logarithm base e) of the number `a`.

In addition, if you need to work with the number e and don't feel like looking it up and typing in enough digits to approximate it accurately, the name `Math.E` is already associated with a very accurate approximation to e.

Trigonometric functions

Another important set of functions provided in Java through the `Math` class are the standard trigonometric functions sine, cosine and tangent and their inverses. The Java names for these functions are shown in the table below:

Java	Function
Math.sin(x)	returns the sine of x
Math.cos(x)	returns the cosine of x
Math.tan(x)	returns the tangent of x
Math.asin(x)	returns the arc sine of x
Math.acos(x)	returns the arc cosine of x
Math.atan(x)	returns the arc tangent of x

For all of these methods, Java assumes that the angles involved are measured in radians rather than degres. The `Math` class contains features to make it a bit easier for those who prefer degrees to work with radians. There are two methods named `Math.toRadians` and `Math.toDegrees` that can be used to convert from one set of units to the other. Also, the name `Math.PI` is associated with a very accurate approximation of the mathematical constant π. Thus, the cosine of a right angle would be computed by saying:

```
Math.cos( Math.toRadians( 90 ) )
```

or

```
Math.cos( Math.PI / 2 )
```