CHAPTER
2

Abstract Data Types and Java Classes

The happiest way to deal with a man is never to tell him anything he does not need to know.

ROBERT A. HEINLEIN
Time Enough for Love

Object-oriented programming (**OOP**) is an approach to programming where data occurs in tidy packages called *objects*. Manipulation of an object happens with functions called *methods*, which are part and parcel of their objects. The Java mechanism to create objects and methods is called a **class**. In fact, the keyword `class` at the start of each Java application program indicates that the program is itself a class with its own methods to carry out tasks.

This chapter moves you beyond small Java application programs. Your goal is to be able to write general purpose classes that can be used by many different programs. Each general purpose class will capture a certain functionality, and an application programmer can look through the available classes to select those that are useful for the job at hand.

For example, consider a programmer who is writing an application to simulate a Martian lander as it goes from orbit to the surface of Mars. This programmer could use classes to simulate the various mechanical components of the lander—the throttle that controls fuel flow, the rocket engine, and so on. If such classes are readily available in a package of "mechanical component classes," then the programmer could select and use the appropriate classes. Typically, one programming team designs and implements such classes, and other programmers use the classes. The programmers who use the classes must be provided with a *specification* of how the classes work, but they need no knowledge of how the classes are *implemented*.

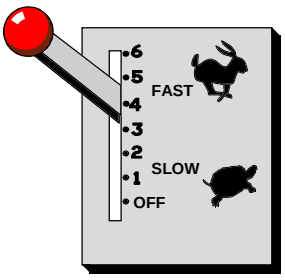
The separation of specification from implementation is an example of *information hiding*, which was presented as a cornerstone of program design in Chapter 1. Such a strong emphasis on information hiding is partly motivated by mathematical research about how programmers can improve their reasoning about data types that are used in programs. These mathematical data types are called **abstract data types,** or ADTs—and therefore, programmers sometimes use the term **ADT** to refer to a class that is presented to other programmers with information hiding. This chapter presents two examples of such classes. The examples illustrate the features of Java classes, with emphasis on information hiding. By the end of the chapter you will be able to implement your own classes in Java. Other programmers could *use* one of your classes without knowing the details of *how* you implemented the class.

ADTs emphasize the specification rather than the implementation

2.1 CLASSES AND THEIR MEMBERS

A class is a new kind of data type. Each of your classes includes various *data*, such as integers, characters, and so on. In addition, a class has the ability to include two other items: *constructors* and *methods*. Constructors are designed to provide initial values to the class's data; methods are designed to manipulate the data. Taken together, the data, constructors, and methods of a class are called the class **members**.

But this abstract discussion does not really tell you what a class *is*. We need some examples. As you read the first example, concentrate on learning the techniques for implementing a class. Also notice how you use a class written by another programmer, without knowing details of the class's implementation.



PROGRAMMING EXAMPLE: The Throttle Class

Our first example of a class is a new data type to store and manipulate the status of a mechanical throttle. An object of this new class holds information about a throttle, as shown in the picture. The throttle is a lever that can be moved to control fuel flow. The throttle we have in mind has a single shutoff point (where there is no fuel flow) and a sequence of several on positions where the fuel is flowing at

progressively higher rates. At the topmost position, the fuel flow is fully on. At intermediate positions, the fuel flow is proportional to the location of the lever. For example, with six possible positions, and the lever in the fourth position, the fuel flows at $\frac{4}{6}$ of its maximum rate.

A constructor is designed to provide initial values to a class's data. The throttle constructor permits a program to create a new throttle with a specified number of "on positions" above the shutoff position. For instance, a throttle for a lawn mower could specify six positions, whereas a throttle for a Martian lander could specify 1000 positions. The throttle's lever is initially placed in the shutoff position.

Once a throttle has been initialized, there two methods to shift the throttle's lever: One of the methods shifts the lever by a given amount, and the other method returns the lever to the shutoff position. We also have two methods to examine the status of a throttle. The first of these methods returns the amount of fuel currently flowing, expressed as a proportion of the maximum flow. For example, this method will return approximately 0.667 when a six-position throttle is in its fourth position. The other method returns a true-or-false value, telling whether the throttle is currently on (that is, whether the lever is above the shutoff position). Thus, the throttle has one constructor and four methods listed here:

one throttle constructor and four throttle methods

- A constructor to create a new throttle with one shutoff position and a specified number of on positions (the lever starts in the shutoff position)
- A method that returns the fuel flow, expressed as a proportion of the maximum flow
- A method to tell us whether the throttle is currently on
- A method to shift a throttle's lever by a given amount
- A method to set the throttle's lever back to the shutoff position

Defining a New Class

We're ready to define a new Java class called `Throttle`. The new class includes data (to store information about the throttle) plus the constructor and methods listed above. Once the `Throttle` class is defined, a programmer can create objects of type `Throttle` and manipulate those objects with the methods.

Here's an outline of the `Throttle` class definition:

declaring the Throttle class

```
public class Throttle
{
   private int top;      // The topmost position of the lever
   private int position; // The current position of the lever

   This part of the class definition provides the implementations
   of the constructor and methods.
}
```

This class definition defines a new data type called `Throttle`. The definition starts with the **class head,** which consists of the Java keywords `public class`, followed by the name of the new class. The keyword `public` is necessary before the `class` because we want to allow all other programmers (the "public") to use the new class. The name of the class may be any legal identifier. We chose the name `Throttle`. We always use a capital letter for the first character of names of new classes—this isn't required by Java, but it's a common programming style, making it easy to identify class names.

three varieties of class members appear in the class definition

The rest of the class definition, between the two brackets, lists all the components of the class. These components are called **members** of the class and they come in three varieties: instance variables, constructors, and methods.

Instance Variables

The first kind of member is a variable declaration. These variables are called **instance variables** (or sometimes "member variables"). The `Throttle` has two instance variables:

```
private int top;      // The topmost position of the lever
private int position; // The current position of the lever
```

Each instance variable stores some piece of information about the status of an object. For example, consider a throttle with six possible positions and the lever in the fourth position. This throttle would have `top=6` and `position=4`.

The keyword `private` occurs in front of each of our instance variables. This keyword means that programmers who use the new class have no way to read or assign values directly to the private instance variables. It is possible to have public instance variables that can be accessed directly, but public instance variables tend to reveal too much information about how a class is implemented, violating the principle of information hiding. Therefore, our examples will use private instance variables. All access to private instance variables is carried out through the constructors and methods that are provided with the class.

Constructors

The second kind of member is a constructor. A constructor is a method that is responsible for initializing the instance variables. For example, our constructor creates a throttle with a specified number of on positions above the shutoff position. This constructor sets the instance variable `top` to a specified number, and sets `position` to zero (so that the throttle is initially shut off).

For the most part, implementing a constructor is no different than your past work (such as implementing a method for a Java application). The primary difference is that a constructor has access to the class's instance variables, and is responsible for initializing these variables. Thus, a throttle constructor must provide initial values to `top` and `position`. Before you implement the throttle constructor, you must know the several rules that make constructors special:

- Before any constructor begins its work, all instance variables are assigned Java "default values." For example, the Java default value for any number variable is zero.
- If an instance variable has an initialization value with its declaration, the initialization value replaces the default value. For example, suppose we have this instance variable:

  ```
  int jackie = 42;
  ```

  The instance variable `jackie` is first given its default value of zero; then the zero is replaced by the initialization value of 42.
- The name of a constructor must be the same as the name of the class. In our example, the name of the constructor is `Throttle`. This seems strange: Normally we *avoid* using the same name for two different things. But it is a requirement of Java that the constructor use the same name as the class.
- A constructor is not really a method, and therefore it does not have *any* return value. Because of this, you must *not* write `void` (or any other return type) at the front of the constructor's head. The compiler knows that every constructor has no return value, but a compiler error occurs if you actually write `void` at the front of the constructor's head.

With these rules, we can write the throttle's constructor as shown here (with its specification following the format from Section 1.1):

◆ **Constructor for the Throttle**

`public Throttle(int size)`

Construct a `Throttle` with a specified number of on positions.

Parameters:

`size` – the number of on positions for this new `Throttle`

Precondition:

`size > 0`.

Postcondition:

This `Throttle` has been initialized with the specified number of on positions above the shutoff position, and it is currently shut off.

Throws: `IllegalArgumentException`

Indicates that `size` is not positive.

```
public Throttle(int size)
{
   if (size <= 0)
      throw new IllegalArgumentException("Size <= 0: " + size);
   top = size;
   // No assignment needed for position -- it gets the default value of zero.
}
```

This constructor sets `top` according to the parameter, `size`. It does not explicitly set `position`, but the comment in the implementation indicates that we did not

just forget about `position`—the default value of zero is its correct initial value. The implementation is preceded by the keyword `public` to make it available to all programmers.

a class may have many different constructors

The throttle has just one constructor, just one way of setting the initial values of the instance variables. Some classes may have many different constructors that set initial values in different ways. If there are several constructors, then each constructor must have a distinct sequence of parameters to distinguish it from the other constructors.

No-Arguments Constructors

Some classes have a constructor with no parameters, called a **no-arguments** constructor. In effect, a no-arguments constructor does not need any extra information to set the initial values of the instance variables.

If you write a class with no constructors at all, then Java automatically provides a no-arguments constructor that initializes each instance variable to its initialization value (if there is one) or to its default value (if there is no specified initialization value). There is one situation where Java does not provide an automatic no-arguments constructor, and you'll see this situation when you write subclasses in Chapter 13.

Methods

The third kind of class member is a method. A method does computations that access the class's instance variables. Classes tend to have two kinds of methods:

1. Accessor methods. An **accessor method** gives information about an object without altering the object. In the case of the throttle, an accessor method can return information about the status of a throttle, but it must not change the position of the lever.

2. Modification methods. A **modification method** may change the status of an object. For a throttle, a modification method may shift the lever up or down.

Each class method is designed for a specific manipulation of an object—in our case, the manipulation of a throttle. To carry out the manipulations, each of the throttle methods has access to the throttle's instance variables, `top` and `position`. The methods can examine `top` and `position` to determine the current status of the throttle, or `top` and `position` can be changed in order to alter the status of the throttle. Let's look at the details of the implementations of the throttle methods, beginning with the accessor methods.

Accessor Methods

Accessor methods provide information about an object without changing the object. Accessor methods are often short, just returning the value of an instance

variable or performing a computation with a couple of instance variables. The first of the throttle accessor methods computes the current flow as a proportion of the maximum flow. The specification and implementation are shown here:

- **getFlow**

  `public double getFlow( )`

  Get the current flow of this `Throttle`.

  Returns:

  the current flow rate (always in the range [0.0 ... 1.0]) as a proportion of the maximum flow

```
public double getFlow( )
{
   return (double) position / (double) top;
}
```

accessor methods often have no parameters

Accessor methods often have no parameters, no precondition, and only a simple return condition in the specification. How does an accessor method manage with no parameters? It needs no parameters because all of the necessary information is available in the instance variables.

Pitfall: Integer Division Throws Away the Fractional Part

PITFALL

The `getFlow` implementation computes and returns a fractional value. For example, if position is 4 and top is 6, then `getFlow` returns approximately 0.667. In order to get a fractional result in the answer, the integer numbers `position` and `top` cannot simply be divided with the expression `position/top`, since this would result in an integer division ($\frac{4}{6}$ results in the quotient 0, discarding any remainder). Instead, we must force Java to compute a fractional division by changing the integer values to double values. For example, expression `(double) position` is a "cast" that changes the integer value of position to a double value to use in the division.

The throttle's second accessor method returns a true-or-false value indicating whether the fuel flow is on. Here is this method with its specification:

- **isOn**

  `public boolean isOn( )`

  Check whether this `Throttle` is on.

  Returns:

  If this `Throttle`'s flow is above zero, then the return value is `true`; otherwise the return value is `false`.

```
public boolean isOn( )
{
   return (position > 0);
}
```

TIP

Programming Tip: Use the Boolean Type for True-or-False Values

Java's basic boolean type may be relatively unfamiliar. You should use the boolean type for any true-or-false value such as the return value of the `isOn` method. The return statement for a boolean method can be any boolean expression, for example a comparison such as `(position > 0)`. In this example, if `position` is greater than zero, then the comparison is `true`, and `isOn` returns `true`. On the other hand, if position is equal to zero, then the comparison is `false`, and `isOn` returns `false`.

By the way, the name "boolean" is derived from the name of George Boole, a 19th-century mathematician who developed the foundations of a formal calculus of logical values. Boole was a self-educated scholar with limited formal training. He began his teaching career at the age of 16 as an elementary school teacher and eventually took a position as professor at Queen's College in Cork. As a dedicated teacher, he died at the age of only 49—the result of pneumonia brought on by a two-mile trek through the rain to lecture to his students.

Modification Methods

There are two more throttle methods. These two are **modification methods**, which means that they are capable of changing the values of the instance variables. Here is the first modification method:

- **shutOff**
  `public void shutOff( )`
  Turn off this `Throttle`.
  Postcondition:
  This `Throttle`'s flow has been shut off.

```
public void shutOff( )
{
   position = 0;
}
```

modification methods are usually void

Modification methods are usually `void`, meaning that there is no return value. In the specification of a modification method, the method's work is fully described in the postcondition.

The throttle's `shutOff` method has no parameters—it doesn't need parameters because it just moves the throttle's position down to zero, shutting off the flow. However, most modification methods do have parameters, such as a throttle method to shift the throttle's lever by a specified amount. This `shift` method has one integer parameter called `amount`. If `amount` is positive, then the throttle's lever is moved up by that `amount` (but never beyond the topmost position). A negative `amount` causes the lever to move down (but never below zero). The specification and implementation appear at the top of the next page.

- **shift**

  `public void shift(int amount)`

  Move this `Throttle`'s position up or down.

  Parameters:

  `amount` – the amount to move the position up or down (a positive amount moves the position up, a negative amount moves it down)

  Postcondition:

  This `Throttle`'s position has been moved by the specified amount. If the result is more than the topmost position, then the position stays at the topmost position. If the result is less than the zero position, then the position stays at the zero position.

```
public void shift(int amount)
{
   if (amount > top - position)
      // Adding amount would put the position above the top.
      position = top;
   else if (position + amount < 0)
      // Adding amount would put the position below zero.
      position = 0;
   else
      // Adding amount puts position in the range [0...top].
      position += amount;
}
```

This might be the first time you've seen the += operator. Its effect is to take the value on the right side (such as `amount`) and add it to what's already in the variable on the left (such as `position`). This sum is then stored back in the variable on the left side of +=.

The `shift` method requires care to ensure that the position does not go above the topmost position nor below zero. For example, the first test in the method checks whether `(amount > top - position)`. If so, then adding `amount` to `position` would push the `position` over `top`. In this case, we simply set `position` to `top`.

It is tempting to write the test `(amount > top - position)` in a slightly different way, like this:

```
if (position + amount > top)
   // Adding amount would put the position above the top.
   position = top;
```

This seems okay at first glance, but there is a potential problem: What happens if both `position` and `amount` are large integers such as 2,000,000,000? The subexpression `position + amount` should be 4,000,000,000, but Java tries to temporarily store the subexpression as a Java integer, which is limited to the range –2,147,483,648 to 2,147,483,647. The result is an **arithmetic overflow**, which is defined as trying to compute or store a number that is beyond the legal

range of the data type. When an arithmetic overflow occurs, the program might stop with an error message or it might continue computing with wrong data.

We avoided the arithmetic overflow by rearranging the first test to avoid the troublesome subexpression. The test we use is:

```
if (amount > top - position)
   // Adding amount would put the position above the top.
   position = top;
```

This test uses the subexpression `top - position`. Since `top` is never negative, and `position` is in the range `[0...top]`, the subexpression `top - position` is always a valid integer in the range `[0...top]`.

What about the second test in the method? In the second test, we use the subexpression `position + amount`, but at this point, `position + amount` can no longer cause an arithmetic overflow. Do you see why? If `position + amount` is bigger than `top`, then the first test would have been `true` and the second test is never reached. Therefore, by the time we reach the second test, the subexpression `position + amount` is guaranteed to be in the range `[amount...top]`, and arithmetic overflow cannot occur.

PITFALL

Pitfall: Potential Arithmetic Overflows

Check all arithmetic expressions for potential arithmetic overflow. The limitations for Java variables and subexpressions are given in Appendix A. Often you can rewrite an expression to avoid overflow, or you can use `long` variables (with a range from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807). If overflow cannot be avoided altogether, then include a note in the documentation to describe the situation that causes overflow.

the name of the java file must match the name of the class

We have completed the `Throttle` class implementation and can now put the complete definition in a file called `Throttle.java`, as shown in Figure 2.1. The name of the file must be `Throttle.java` since the class is `Throttle`.

FIGURE 2.1 Specification and Implementation for the `Throttle` Class

Class Throttle

❖ **public class Throttle**

A `Throttle` object simulates a throttle that is controlling fuel flow.

(continued)

(FIGURE 2.1 continued)

Specification

- **Constructor for the Throttle**

  `public Throttle(int size)`

  Construct a `Throttle` with a specified number of on positions.

  Parameters:

  `size` – the number of on positions for this new `Throttle`

  Precondition:

  `size > 0`.

  Postcondition:

  This `Throttle` has been initialized with the specified number of on positions above the shutoff position, and it is currently shut off.

  Throws: `IllegalArgumentException`

  Indicates that `size` is not positive.

- **getFlow**

  `public double getFlow( )`

  Get the current flow of this `Throttle`.

  Returns:

  the current flow rate (always in the range [0.0 ... 1.0]) as a proportion of the maximum flow

- **isOn**

  `public boolean isOn( )`

  Check whether this `Throttle` is on.

  Returns:

  If this `Throttle`'s flow is above zero, then the return value is `true`; otherwise the return value is `false`.

- **shift**

  `public void shift(int amount)`

  Move this `Throttle`'s position up or down.

  Parameters:

  `amount` – the amount to move the position up or down (a positive amount moves the position up, a negative amount moves it down)

  Postcondition:

  This `Throttle`'s position has been moved by the specified amount. If the result is more than the topmost position, then the position stays at the topmost position. If the result is less than the zero position, then the position stays at the zero position.

- **shutOff**

  `public void shutOff( )`

  Turn off this `Throttle`.

  Postcondition:

  This Throttle has been shut off.

(continued)

(FIGURE 2.1 continued)

Implementation

```
// File: Throttle.java

public class Throttle
{
   private int top;       // The topmost position of the throttle
   private int position; // The current position of the throttle

   public Throttle(int size)
   {
      if (size <= 0)
         throw new IllegalArgumentException("Size <= 0: " + size);
      top = size;
      // No assignment needed for position -- it gets the default value of zero.
   }

   public double getFlow( )
   {
      return (double) position / (double) top;
   }

   public boolean isOn( )
   {
      return (getFlow( ) > 0);
   }

   public void shift(int amount)
   {
      if (amount > top - position)
         // Adding amount would put the position above the top.
         position = top;
      else if (position + amount < 0)
         // Adding amount would put the position below zero.
         position = 0;
      else
         // Adding amount puts position in the range [0...top].
         position += amount;
   }

   public void shutOff( )
   {
      position = 0;
   }
}
```

Methods May Activate Other Methods

The throttle's `isOn` method in Figure 2.1 has one change from the original implementation. The change is highlighted here:

```
public boolean isOn( )
{
   return (getFlow( ) > 0);
}
```

In this implementation, we have checked whether the flow is on by calling the `getFlow` method rather than looking directly at the `position` instance variable. Both implementations work: Using position directly probably executes quicker, but you could argue that using `getFlow` makes the method's intent clearer. Anyway, the real purpose of this change is just to illustrate that one method can call another to carry out a subtask. In this example, the `isOn` method calls `getFlow`. An OOP programmer usually would use slightly different terminology, saying that the `isOn` method **activated** the `flow` method. **Activating a method** is nothing more than OOP jargon for "calling a method."

Programming Tip: Private Versus Public

TIP

Our `Throttle` class follows a common pattern: The data about a throttle is stored in private instance variables, indicated by the keyword `private` before each declaration of an instance variable. A throttle is manipulated through public methods, indicated by the keyword `public` before each implementation of a method.

The pattern of "private data, public methods" is a good idea. It forbids other programmers from using our instance variables in unintended ways. Later you will see examples that include private methods (i.e., methods that can be activated within other methods of the class, but may not be used by other programmers). For now, though, the common pattern will serve you well.

Self-Test Exercises

1. Name and describe the three kinds of class members we have used. In this section, which kinds of members were public and which were private?
2. Write a new throttle constructor with no arguments. The constructor sets the top position to 1 and sets the current position off.
3. Write another throttle constructor with two arguments: the total number of positions for the throttle, and its initial position.
4. Add a new throttle method that will return `true` if the current flow is more than half. The body of your implementation should activate `getFlow`.

5. Design and implement a class called `Clock`. A `Clock` object holds one instance of a time value such as 9:48 P.M. Have at least these public methods:
   - A no-arguments constructor that initializes the time to midnight—see page 40 for the discussion of a no-arguments constructor
   - A method to explicitly assign a given time—you will have to give some thought to appropriate arguments for this method
   - Methods to retrieve information: the current hour, the current minute, and a boolean method to determine whether the time is at or before noon
   - A method to advance the time forward by a given number of minutes (which could be negative to move the clock backward or positive to move the clock forward)

2.2 USING A CLASS

programs can create new objects of a class

How do you use a new class such as `Throttle`? Within any program, you may create new throttles, and refer to these throttles by names that you define. We can illustrate the general syntax for creating and using these objects by an example.

Creating and Using Objects

Suppose a program needs a new throttle with 100 positions above the shutoff. Within the program, we want to refer to the throttle by the name `control`. The Java syntax has these parts:

```
Throttle control = new Throttle(100);
```

The first part of this statement—`Throttle control`—declares a new variable called `control`. The `control` variable is capable of refering to a throttle. The second part of the statement—`new Throttle(100)`—creates a new throttle and initializes `control` to refer to this new throttle. A new throttle that is created in this way is called a `Throttle` **object**.

There are a few points to notice about the syntax for creating a new `Throttle` object: `new` is a keyword to create a new object; `Throttle` is the data type of the new object; and `(100)` is the list of parameters for the constructor of the new object. So, we are creating a new throttle and 100 is the argument for the constructor, so the new throttle will have 100 positions above the shutoff.

Once the throttle is created, we can refer to the throttle by the name that we selected: `control`. For example, suppose we want to shift the lever up to its third notch. We do this by calling the `shift` method, as shown here:

```
control.shift(3);
```

Calling a method always involves these four pieces:

1. Start with a reference to the object that you are manipulating. In this example, we want to manipulate `control`, so we begin with "`control`". Remember that you cannot just call a method—you must always indicate which object is being manipulated.
2. Next, place a single period.
3. Next, write the name of the method. In our example, we call the `shift` method, so we write "`control.shift`"—which you can pronounce "control *dot* shift."
4. Finally, list the parameters for the method call. In our example, `shift` requires one parameter, which is the amount (3) that we are shifting the throttle. Thus, the entire statement is `control.shift(3);`

how to use a method

Our example called the `shift` method. As you've seen before, OOP programmers like their own terminology and they would say that we **activated** the `shift` method. In the rest of the text, we'll try to use "activate" rather than "call." (This will keep us on the good side of OOP programmers.)

As another example, here is a sequence of several statements to set a throttle to a certain point, and then print the throttle's flow:

```
final int SIZE = 8;  // The size of the Throttle
final int SPOT = 3;  // Where to move the Throttle's lever

Throttle small = new Throttle(SIZE);

small.shift(SPOT);
System.out.print("My small throttle is now at position ");
System.out.println(SPOT + " out of " + SIZE + ".");
System.out.println("The flow is now: " + small.getFlow( ));
```

Notice how the return value of `small.getFlow` is used directly in the output statement. As with any other method, the return value of an accessor method can be used as part of an output statement or other expression. The output from this code is:

```
My small throttle is now at position 3 out of 8.
The flow is now: 0.375
```

A Program with Several Throttle Objects

A single program may have many throttle objects. For example, this code will declare two throttle objects, shifting each throttle to a different point:

```
Throttle tiny = new Throttle(4);
Throttle huge = new Throttle(10000);

tiny.shift(2);
huge.shift(2500);
```

Here's an important concept to keep in mind:

> When a program has several objects of the same type, each object has its own copies of the instance variables.

In the example above, `tiny` has its own instance variables (`top` will be 4 and `position` will be 2); `huge` also has its own instance variables (`top` will be 10000 and `position` will be 2500). When we activate a method such as `tiny.shift`, the method uses the instance variables from `tiny`; when we activate `huge.shift`, the method uses the instance variables from `huge`.

The variables in our examples—`control`, `small`, `tiny`, `huge`—are called **reference variables** because they are used to *refer* to objects (in our case, throttles). There are several differences between a reference variable (used by Java for all classes) and an ordinary variable (used by Java for the primitive data types of `int`, `char`, and so on). Let's look at these differences, beginning with a special value called `null` that is used only with reference variables.

Null References

The creation of a new object can be separated from the declaration of a variable. For example, the following two statements can occur far apart in a program:

```
Throttle control;
...
control = new Throttle(100);
```

Once both statements finish, `control` refers to a newly created throttle with 100 positions. But what is the status of `control` between the statements? At this point, `control` does not yet refer to any throttle, because we haven't yet created a throttle. In this situation, we can assign a special value to control, indicating that control does not yet refer to anything. The value is called the **null reference**, written with the keyword `null` in Java. So we could change the above example to this:

```
Throttle control = null;
...
control = new Throttle(100);
```

In this area, control does not refer to anything.

Null Reference

Sometimes a reference variable does not refer to anything. This is a **null reference**, and the value of the variable is called **null**.

Sometimes a program finishes using an object. In this case, the program may explicitly set a reference variable to `null`, as shown here:

```
Throttle control = new Throttle(100);

// Various statements that use the Throttle appear next...
...

// Now we are done with the control Throttle, so we can set
// the reference to null.
control = null;
```

Once a reference variable is no longer needed, it's a good idea to set it to `null`, allowing Java to economize on certain resources (such as the memory used by a throttle).

Pitfall: Null Pointer Exception

PITFALL

When a variable such as `control` becomes `null`, it no longer refers to any throttle. If `control` is `null`, then it is a programming error to activate a method such as `control.shift`. The result is an exception called `NullPointerException`.

Assignment Statements with Reference Variables

The usual assignment statement may be used with reference variables. For example, we might have two `Throttle` variables `t1` and `t2`, and an assignment such as `t2 = t1` is permitted. But what is the effect of the assignment? For starters, if `t1` is `null`, then the assignment `t2 = t1` also makes `t2` `null`. Here is a more complicated case where `t1` is not `null`:

```
Throttle t1;
Throttle t2;

t1 = new Throttle(100);
t1.shift(25);
t2 = t1;
```

The effect of the assignment `t2 = t1` is somewhat different than assignments for integers or other primitive data types. The effect of `t2 = t1` is to "make `t2`

refer to the same object that `t1` is already refering to." In other words, we have two reference variables (`t1` and `t2`), but we created only one throttle (with one `new` statement). This one throttle has 100 positions, and is currently in the 25th position. After the assignment statement, both `t1` and `t2` refer to this one throttle.

As an example, let's start with the two declarations:

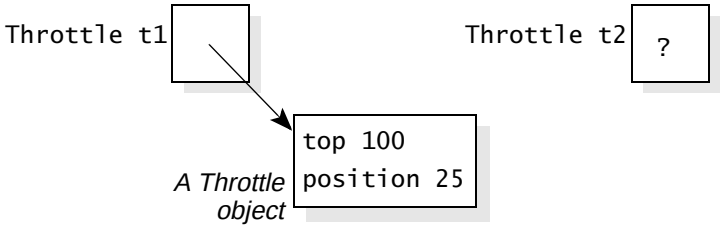
```
Throttle t1;
Throttle t2;
```

We now have two variables, `t1` and `t2`. If these variables are declared in a method, then they don't yet have an initial value (not even `null`). We can draw this situation with a question mark for each value, as shown here:

The next two statements are:

```
t1 = new Throttle(100);
t1.shift(25);
```

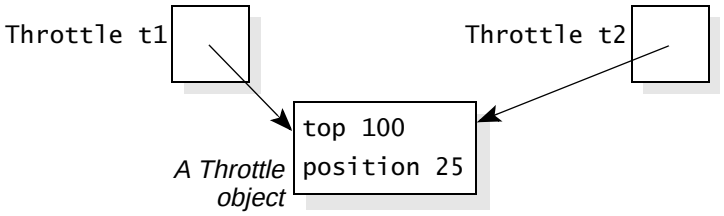
These statements create a new throttle for `t1` to refer to, and shift the throttle's position to 25. We will draw a separate box for the throttle and indicate its instance variables (`top` at 100 and `position` at 25). To show that `t1` refers to this throttle, we draw an arrow from the `t1` box to the throttle, like this:



At this point, we can execute the assignment:

```
t2 = t1;
```

After the assignment, `t2` will refer to the same object that `t1` refers to, as shown here:



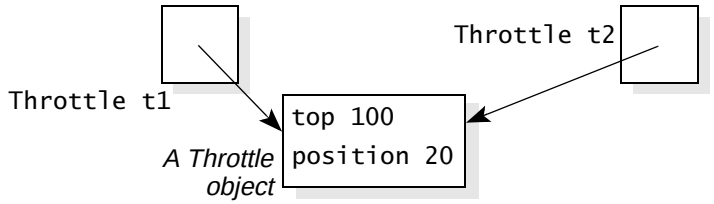
There are now two references to the same throttle, which can cause some surprising results. For example, suppose we shift `t2` down five notches and then print the flow of `t1`, like this:

```
t2.shift(-5);
System.out.println("Flow of t1 is: " + t1.getFlow( ));
```

What flow rate is printed? The `t1` throttle was set to position 25 out of 100, and we never directly altered its position. But `t2.shift(-5)` moves the throttle's position down to 20. Since `t1` refers to this same throttle, `t1.getFlow` now returns 20/100, and the output statement prints "Flow of t1 is: 0.2". Here's the entire code that we executed and the final situation drawn as a picture:

```
Throttle t1;
Throttle t2;

t1 = new Throttle(100);
t1.shift(25);
t2 = t1;
t2.shift(-5);
```



Assignment Statements with Reference Variables

If `t1` and `t2` are reference variables, then the assignment `t2 = t1` is allowed.

If `t1` is `null`, then the assignment also makes `t2` `null`.

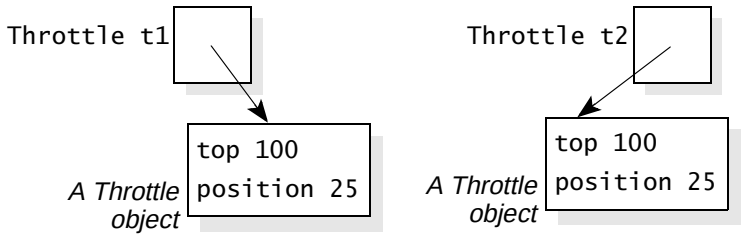
If `t1` is not `null`, then the assignment changes `t2` so that it refers to the same object that `t1` already refers to. At this point, changes can be made to that one object through either `t1` or `t2`.

The situation of an assignment statement contrasts with a program that actually creates two separate throttles for `t1` and `t2`. For example, two separate throttles can be created with each throttle in the 25th position out of 100, as shown in the code at the top of the next page.

```
Throttle t1;
Throttle t2;

t1 = new Throttle(100);
t1.shift(25);
t2 = new Throttle(100);
t2.shift(25);
```

With this code, we have two separate throttles:



Changes that are now made to one throttle will not effect the other, because there are two completely separate throttles.

Clones

A programmer sometimes needs to make an exact copy of an existing object. The copy must be just like the existing object, but separate. Subsequent changes to the copy should not alter the original, nor should subsequent changes to the original alter the copy. A separate copy such as this is called a **clone**.

An assignment operation `t2 = t1` does not create a clone, and in fact the `Throttle` class does not permit the easy creation of clones. But many other classes have a special method called `clone` for just this purpose. Writing a useful `clone` method has some requirements that may not be evident just now, so we will postpone a complete discussion until Section 2.4.

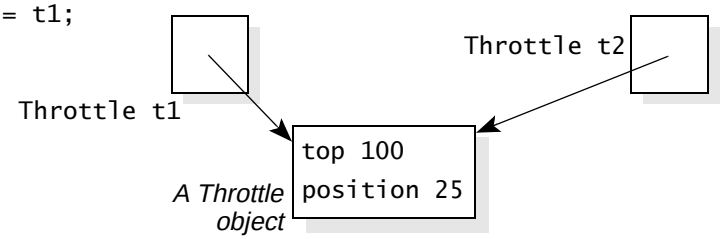
Testing for Equality

A test for equality (`t1 == t2`) can be carried out with reference variables. The equality test `(t1 == t2)` is `true` if both `t1` and `t2` are null, or if they both refer to the exact same object (not two different objects that happen to have the same values for their instance variables). An inequality test `(t1 != t2)` can also be carried out. The result of an inequality test is always the opposite of an equality test. Let's look at two examples.

The first example creates just one throttle; `t1` and `t2` both refer to this throttle as shown in the following picture:

```
Throttle t1;
Throttle t2;

t1 = new Throttle(100);
t1.shift(25);
t2 = t1;
```

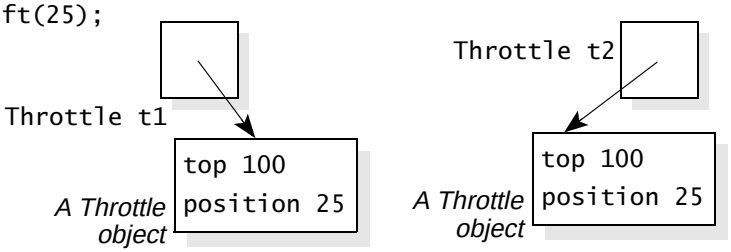


At this point in the computation, `(t1 == t2)` is `true`. Both reference variables refer to the same object.

On the other hand, consider this code, which creates two separate throttles:

```
Throttle t1;
Throttle t2;

t1 = new Throttle(100);
t1.shift(25);
t2 = new Throttle(100);
t2.shift(25);
```



After this computation, `(t1 == t2)` is `false`. The two throttles have the same value (with `top` at 100 and `position` at 25), but the equality test returns false because they are two separate throttles.

Test for Equality with Reference Variables

For reference variables `t1` and `t2`, the test `(t1 == t2)` is `true` if both references are null, or if `t1` and `t2` refer to the exact same object (not two different objects that happen to have the same values for their instance variables).

Terminology Controversy: "The Throttle That t Refers To"

A declaration such as `Throttle t = new Throttle(42)` declares a reference variable `t`, and makes it refer to a newly created throttle. We can then talk about "the throttle that `t` refers to." This is the correct terminology, but sometimes a programmer's thinking is clarified by shortening the terminology and saying things like "the throttle t is on" rather than "the throttle that t refers to is on."

Which is right? In general, use the longer terminology when there may be several different variables referring to the same throttle. Otherwise use the shorter phrase "the throttle t is on," but somewhere, deep in your mind, remember that you are shortening things for convenience and that the longer phrase is right.

Self-Test Exercises

6. Write some Java code that creates a new throttle with six positions, shifts the throttle halfway up (to the third position), and prints the current flow.
7. A method declares a `Throttle` variable called `control`, but there is not yet a throttle. What value should be assigned to `control`?
8. Suppose that `control` is a null reference. What happens if a program tries to activate `control.shift`?
9. What is the output of this code:

```
Throttle t1;
Throttle t2;
t1 = new Throttle(100);
t2 = t1;
t1.shift(40);
t2.shift(2);
System.out.println(t1.getFlow( ));
```

10. Consider the code from the previous question. At the end of the computation, is `(t1 == t2) true` or `false`?
11. Write some code that will make `t1` and `t2` refer to two different throttles with 100 positions each. Both throttles are shifted up to position 42. At the end of your code, is `(t1 == t2) true` or `false`?

2.3 PACKAGES

You now know enough to write a Java application program that uses a throttle. The `Throttle` class would be in one file (`Throttle.java` from Figure 2.1 on page 46) and the program that uses the `Throttle` class would be in a separate file. However, there's one more level of organization that will make it easier for other programmers to use your classes. The organization, called a Java **package**, is a group of related classes put together in a way that makes it easy for programs to use the classes.

Declaring a Package

The first step in declaring a package of related classes is to decide on a name for the package. For example, perhaps we are declaring a bunch of Java classes to simulate various real-world devices such as a throttle. A good short name for the package is the `simulations` package. But there's a problem with good short names: Other programmers might decide to use the same good short name for their packages, resulting in the same name for two different packages.

use your Internet domain name

The solution is to include your Internet domain name as part of the package name. For example, at the University of Colorado the Internet domain name is `colorado.edu` (my e-mail address is `main@colorado.edu`). Therefore, instead of using the package name `simulations`, I will use the longer package name `edu.colorado.simulations` (package names may include a "dot" as part of the name). Many programmers follow this convention, using the Internet domain name in reverse. The only likely conflicts are with other programmers at your own Internet domain, and those conflicts can be prevented by internal cooperation.

Once you have decided on a package name, a *package declaration* must be made at the top of each source file of the package. The **package declaration** consists of the keyword `package` followed by the full package name and a semicolon. The declaration appears at the start of each source file, before any class declarations. For example, the start of `Throttle.java` is changed to include the package declaration shown here:

```
package edu.colorado.simulations;
```

The revised `Throttle.java`, with a package declaration, is shown in Figure 2.2. Some Java development environments require you to create a directory structure for your classes to match the structure of package names. For example, suppose that you are doing your code development in your own directory called `classes`, and you want to use the `edu.colorado.simulations` package. Then you would follow these steps:

- Make sure that your Java development environment can find and run any classes in your `classes` directory. The exact method of setting this up varies from one environment to another, but a typical approach is to define a system `CLASSPATH` variable to include your own `classes` directory.
- Underneath the `classes` directory, create a subdirectory called `edu`.
- Underneath `edu`, create a subdirectory called `colorado`.
- Underneath `colorado`, create a subdirectory called `simulations`.
- All the `.java` and `.classes` files for the package must be placed in the `simulations` subdirectory.

If the `edu.colorado.simulations` package has other classes, then their files are also placed in the `simulations` subdirectory, and the package declaration is placed at the start of each `.java` file.

FIGURE 2.2 Defining `Throttle.java` as Part of the `edu.colorado.simulations` Package

Implementation

```
// File: Throttle.java from the package edu.colorado.simulations
// Documentation is in Figure 2.1 on page 44 or from the Throttle link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.simulations;

public class Throttle
{
   private int top;      // The topmost position of the throttle
   private int position; // The current position of the throttle

   public Throttle(int size)
   {
      if (size <= 0)
         throw new IllegalArgumentException("Size <= 0: " + size);
      top = size;
      // No assignment needed for position -- it gets the default value of zero.
   }

   public double getFlow( )
   {
      return (double) position / (double) top;
   }

   public boolean isOn( )
   {
      return (getFlow( ) > 0);
   }

   public void shift(int amount)
   {
      if (amount > top - position)
         // Adding amount would put the position above the top.
         position = top;
      else if (position + amount < 0)
         // Adding amount would put the position below zero.
         position = 0;
      else
         // Adding amount puts position in the range [0...top].
         position += amount;
   }
```

the package declaration

(continued)

(FIGURE 2.2 continued)

```
}

   public void shutOff( )
   {
      position = 0;
   }
}
```

The Import Statement to Use a Package

a program can use an entire package or just parts of a package

Once a package is set up and in the correct directory, the package's `.java` files can be compiled to create the various `.class` files. Then any other code that you write may use part or all of the package. To use another package, a `.java` file places an import statement after its own package statement but before anything else. An **import statement for an entire package** has the keyword `import` followed by the package name plus ".*" and a semicolon. For example, we can import the entire `edu.colorado.simulations` package with the import statement:

```
import edu.colorado.simulations.*;
```

If only a few classes from a package are needed, then each class can be imported separately. For example, this statement imports only the `Throttle` class from the `edu.colorado.simulations` package:

```
import edu.colorado.simulations.Throttle;
```

After this import statement, the `Throttle` class can be used . For example, a program can declare a variable:

```
Throttle control;
```

A sample program using our throttle appears in Figure 2.3. The program creates a new throttle, shifts the throttle fully on, and then steps the throttle back down to the shut off position.

The JCL Packages

The Java language comes with many useful packages called the **Java Class Libraries** (**JCL**). Any programmer can use various parts of the JCL by including an appropriate import statement. In fact, one of the packages, `java.lang`, is so useful that it is automatically imported into every Java program. Some parts of the JCL are described in Appendix D.

FIGURE 2.3 Implementation of the `Throttle` Demonstration Program with an Import Statement

Java Application Program

```
// FILE: ThrottleDemonstration.java
// This small demonstration program shows how to use the Throttle class
// from the edu.colorado.simulations package.

import edu.colorado.simulations.Throttle;

class ThrottleDemonstration
{
   public static void main(String[ ] args)
   {
      final int SIZE = 8;  // The size of the demonstration Throttle

      Throttle small = new Throttle(SIZE);

      System.out.println("I am now shifting a Throttle fully on, and then I");
      System.out.println("will shift it back to the shut off position.");

      small.shift(SIZE);
      while (small.isOn( ))
      {
         System.out.println("The flow is now " + small.getFlow( ));
         small.shift(-1);
      }

      System.out.println("The flow is now off");
   }
}
```

the import statement

Output from the Application

```
I am now shifting a Throttle fully on, and then I
will shift it back to the shut off position.
The flow is now 1.0
The flow is now 0.875
The flow is now 0.75
The flow is now 0.625
The flow is now 0.5
The flow is now 0.375
The flow is now 0.25
The flow is now 0.125
The flow is now off
```

More about Public, Private, and Package Access

As you have seen, the `Throttle` class uses private instance variables (to keep track of the current status of a throttle) and public methods (to access and manipulate a throttle). The keywords `public` and `private` are called the **access modifiers** because they control access to the class members.

What happens if you declare a member with no access modifier—neither `public` nor `private`? In this case, the member can be accessed only by other classes in the same package. This kind of access is called **default access** (because there is no explicit access modifier); some programmers call it **package access**, which is a nice descriptive name. We won't use package access much because we prefer the pattern of private instance variables with public methods.

One other kind of access—protected access—will be discussed later when we cover derived classes and inheritance.

Self-Test Exercises

12. Suppose you are writing a package of classes for a company that has the Internet domain `knafn.com`. The classes in the package perform various statistical functions. Select a good name for the package.
13. Describe the directory structure that must be set up for the files of the package in the previous question.
14. Write the `import` statement that must be present to use the package from the previous two questions.
15. What import statement is needed to use the `java.lang` package?
16. Describe public access, private access, and package access. What keywords are needed to obtain each kind of access for a method?

2.4 PARAMETERS, EQUALS METHODS, AND CLONES

Every programmer requires an unshakable understanding of methods and their parameters. This section illustrates these issues and other issues that arise in Java, such as how to test whether two objects are equal to each other and how to make a copy of an object. The examples use a new class called `Location`, which will be placed in a package called `edu.colorado.geometry`.

The purpose of a `Location` object is to store the coordinates of a single point on a plane, as in the picture shown here. The location `p` in the picture lies at coordinates x = –1.0 and y = 0.8. For future reference, you should know that Java has a similar class called `Point` in the `java.awt` package. But Java's `Point` class is limited to integer coordinates and used primarily to describe points on a computer's screen. I thought about using the same name `Point` for the example class of this section, but I decided against it because a program might want to use both classes. It's not legal to import two different classes with the same names (though you can use a full type name such as `java.awt.Point` without an import statement).

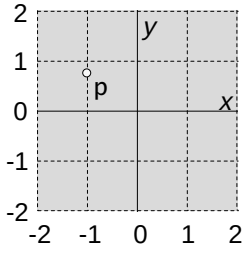
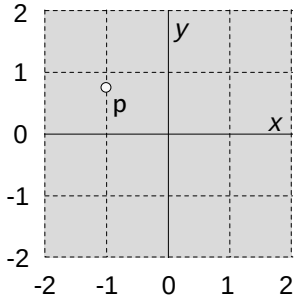
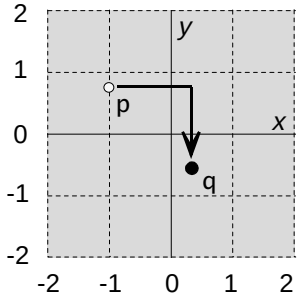


FIGURE 2.4 Three Locations in a Plane

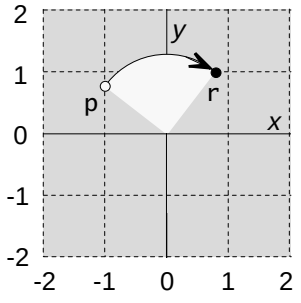
(a) The white dot labeled p is a location with coordinates $x = -1.0$ and $y = 0.8$.



(b) The black dot labeled q was obtained by shifting p by 1.3 units along the *x* axis and by −1.4 units along the *y* axis. The coordinates of q are $x = 0.3$ and $y = -0.6$.



(c) The black dot labeled r was obtained by rotating p by 90° in a clockwise direction around the origin. The coordinates of r are $x = 0.8$ and $y = 1.0$.



The Location Class

Figure 2.4 shows several sample locations. We'll use these sample locations to describe the `Location` constructor and methods.

- There is a constructor to initialize a location. The constructor's parameters provide the initial coordinates. For example, the location `p` in Figure 2.4(a) can be constructed with the statement:
  ```
  Location p = new Location(-1, 0.8);
  ```
- There is a modification method to shift a location by given amounts along the *x* and *y* axes, as shown in Figure 2.4(b).
- There is a modification method to rotate a location by 90° in a clockwise direction around the origin, as shown in Figure 2.4(c).
- There are two assessor methods that allow us to retrieve the current *x* and *y* coordinates of a location.
- There are a couple of methods to perform computations such as the distance between two locations. These are *static* methods—we'll discuss the importance of the static property in a moment.
- There are three methods called `clone`, `equals`, and `toString`. These methods have special importance for Java classes. The `clone` method allows a programmer to make an exact copy of an object. The `equals` method tests whether two different objects are identical. The `toString` method generates a string that represents an object. Special considerations for implementing these three methods are discussed next.

The `Location` class is small, yet it forms the basis for an actual data type that is used in drawing programs and other graphics applications. All the methods and the constructor are listed in the specification of Figure 2.5. The figure also shows one way to implement the class. After you've looked through the figure, we'll discuss that implementation.

FIGURE 2.5 Specification and Implementation for the `Location` Class

Class Location

❖ **public class Location from the package edu.colorado.geometry**

A `Location` object keeps track of a location on a two-dimensional plane.

Specification

◆ **Constructor for the Location**

```
public Location(double xInitial, double yInitial)
```

Construct a `Location` with specified coordinates.

Parameters:

`xInitial` – the initial x coordinate of this `Location`
`yInitial` – the initial y coordinate of this `Location`

Postcondition:

This `Location` has been initialized at the given coordinates.

◆ **clone**

```
public Object clone( )
```

Generate a copy of this `Location`.

Returns:

The return value is a copy of this `Location`. Subsequent changes to the copy will not affect the original, nor vice versa. Note that the return value must be typecast to a `Location` before it can be used.

◆ **distance**

```
public static double distance(Location p1, Location p2)
```

Compute the distance between two `Locations`.

Parameters:

`p1` – the first `Location`
`p2` – the second `Location`

Returns:

the distance between `p1` and `p2`

Note:

The answer is `Double.POSITIVE_INFINITY` if the distance calculation overflows. The answer is `Double.NaN` if either `Location` is null.

(continued)

(FIGURE 2.5 continued)

- **equals**

  `public boolean equals(Object obj)`

  Compare this `Location` to another object for equality.

  Parameters:

  `obj` – an object with which this `Location` is compared

  Returns:

  A return value of `true` indicates that `obj` refers to a `Location` object with the same value as this `Location`. Otherwise the return value is `false`.

  Note:

  If `obj` is null or it is not a `Location` object, then the answer is `false`.

- **getX** and **GetY**

  `public double getX( )` –and– `public double getY( )`

  Get the x or y coordinate of this `Location`.

  Returns:

  the x or y coordinate of this `Location`

- **midpoint**

  `public static Location midpoint(Location p1, Location p2)`

  Generates and returns a `Location` halfway between two others.

  Parameters:

  `p1` – the first `Location`

  `p2` – the second `Location`

  Returns:

  a `Location` that is halfway between `p1` and `p2`

  Note:

  The answer is null if either `p1` or `p2` is null.

- **rotate90**

  `public void rotate90( )`

  Rotate the `Location` 90° in a clockwise direction.

  Postcondition:

  This `Location` has been rotated clockwise 90° around the origin.

- **shift**

  `public void shift(double xAmount, double yAmount)`

  Move this `Location` by given amounts along the x and y axes.

  Postcondition:

  This `Location` has been moved by the given amounts along the two axes.

  Note:

  The shift may cause a coordinate to go above `Double.MAX_VALUE` or below `–Double.MAX_VALUE`. In these cases, subsequent calls of `getX` or `getY` will return `Double.POSITIVE_INFINITY` or `Double.NEGATIVE_INFINITY`.

(continued)

(FIGURE 2.5 continued)

- **toString**

  `public String toString( )`

  Generate a string representation of this `Location`.

  Returns:

  a string representation of this `Location`

Implementation

```
// File: Location.java from the package edu.colorado.geometry
// Documentation is available on pages 63-64 or from the Location link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.geometry;

public class Location implements Cloneable
{
   private double x; // The x coordinate of the Location
   private double y; // The y coordinate of the Location

   public Location(double xInitial, double yInitial)
   {
      x = xInitial;
      y = yInitial;
   }

   public Object clone( )
   {  // Clone a Location object.
      Location answer;

      try
      {
         answer = (Location) super.clone( );
      }
      catch (CloneNotSupportedException e)
      {  // This exception should not occur. But if it does, it would indicate a programming
         // error that made super.clone unavailable. The most common cause would be
         // forgetting the "implements Cloneable" clause at the start of the class.
         throw new RuntimeException
         ("This class does not implement Cloneable.");
      }

      return answer;
   }
```

the meaning of "implements Cloneable" and the clone method are discussed on page 76

(continued)

(FIGURE 2.5 continued)

```
public static double distance(Location p1, Location p2)
{
   double a, b, c_squared;

   // Check whether one of the Locations is null.
   if ((p1 == null)  ||  (p2 == null))
      return Double.NaN;

   // Calculate differences in x and y coordinates.
   a = p1.x - p2.x;
   b = p1.y - p2.y;

   // Use Pythagorean Theorem to calculate the square of the distance
   // between the Locations.
   c_squared = a*a + b*b;

   return Math.sqrt(c_squared);
}

public boolean equals(Object obj)
{
   if (obj instanceof Location)
   {
      Location candidate = (Location) obj;
      return (candidate.x == x) && (candidate.y == y);
   }
   else
      return false;
}

public double getX( )
{
   return x;
}

public double getY( )
{
   return y
}
```

the meaning of a static method is discussed on page 68

the Java constant, double.NaN, is discussed on page 70

the equals method is discussed on page 73

(continued)

(FIGURE 2.5 continued)

```
    public static Location midpoint(Location p1, Location p2)
    {
       double xMid, yMid;

       // Check whether one of the Locations is null.
       if ((p1 == null)  ||  (p2 == null))
          return null;

       // Compute the x and y midpoints.
       xMid = (p1.x/2) + (p2.x/2);
       yMid = (p1.y/2) + (p2.y/2);

       // Create a new Location and return it.
       Location answer = new Location(xMid, yMid);
       return answer;
    }

    public void rotate90( )
    {
       double xNew;
       double yNew;

       // For a 90 degree clockwise rotation, the new x is the original y
       // and the new y is -1 times the original x.
       xNew = y;
       yNew = -x;
       x = xNew;
       y = yNew;
    }

    public void shift(double xAmount, double yAmount)
    {
       x += xAmount;
       y += yAmount;
    }

    public String toString( )
    {
       return "(x=" + x + "  y=" + y + ")";
    }

}
```

Static Methods

The implementation of the `Location` class has several features that may be new to you. Some of the features are in a method called `distance`, with this specification:

- **distance**

  `public static double distance(Location p1, Location p2)`

  Compute the distance between two `Locations`.

  Parameters:

  `p1` – the first `Location`

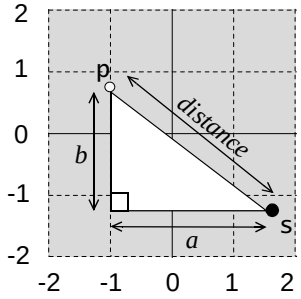
  `p2` – the second `Location`

  Returns:

  the distance between `p1` and `p2`

FIGURE 2.6 The Distance between Locations

The distance between p and s can be computed with the Pythagorean Theorem.



Horizontal distance: $a = 2.7$

Vertical distance: $b = 2.0$

Distance between the locations =

$$\sqrt{a^2 + b^2} = \sqrt{2.7^2 + 2.0^2} = \text{about } 3.36$$

For example, consider the locations `p` and `s` in Figure 2.6. Along a straight line, the distance between these two locations is about 3.36. Using the `distance` method, we can create these two locations and print the distance between them as follows:

```
Location p = new Location(-1, 0.8);
Location s = new Location(1.7, -1.2);

double d = Location.distance(p, s);
System.out.println(d);
```

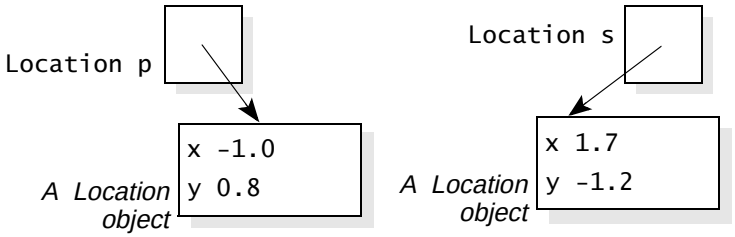
This code prints the distance between the two locations—a little bit more than 3.36.

The `distance` method is modified by an extra keyword: `static`. The `static` keyword means that the method is not activated by any one object. In other words, we do not write `p.distance` or `s.distance`. Instead we write `Location.distance`.

Because the `distance` method is not activated by any one object, the method does not have direct access to the instance variables of a location that activates the method. Within the `distance` implementation, we cannot write simply `x` or `y` (the instance variables). Instead, the implementation must carry out its computation based on the arguments that it's given. For example, if we activate `Location.distance(p, s)`, then the `distance` method works with its two arguments `p` and `s`. These two arguments are both `Location` objects. Let's examine exactly what happens when an argument is an object rather than a primitive value such as an integer.

Parameters That Are Objects

What happens when `Location.distance(p, s)` is activated? For example, suppose we have the two declarations shown previously for `p` and `s`. After these declarations, we have these two separate locations:



Now we can activate the method `Location.distance(p, s)`, which has an implementation that starts like this:

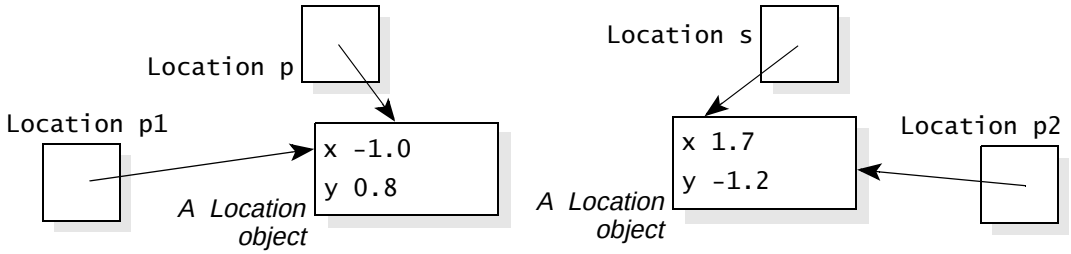
```
public static distance(Location p1, Location p2)
{
   ...
```

"parameters" versus "arguments"

The names used within the method (`p1` and `p2`) are usually called **parameters** to distinguish them from the values that are passsed in (`p` and `s`). On the other hand, the values that are passed in (`p` and `s`) are called the **arguments**. Anyway, the first step of any method activation is to use the *arguments* to provide initial values for the *parameters*. Here's the important fact you need to know about objects:

> When a parameter is an object, such as a `Location`, then the parameter is initialized so that it refers to the same object that the actual argument refers to.

In our example, `Location.distance(p, s)`, the parameters `p1` and `p2` are initialized to refer to the two locations that we created, like this:



Within the body of the `distance` method we can access `p1` and `p2`. For example, we can access `p1.x` to obtain the `x` coordinate of the first parameter. This kind of access is okay in a static method. The only forbidden expression is a direct `x` or `y` (without a qualifier such as `p1`).

be careful about changing the value of a parameter

Some care is needed in accessing a parameter that is an object. For instance, any change to `p1.x` will affect the actual argument `p.x`. We don't want the `distance` method to make changes to its arguments; it should just compute the distance between the two locations and return the answer. This computation occurs in the implementation of `distance` on page 66.

The implementation also handles a couple of special cases. One special case is when an argument is null. In this case, the corresponding parameter will be initialized as null, and the distance method executes this code:

```
// Check whether one of the Locations is null.
if ((p1 == null)  ||  (p2 == null))
   return Double.NaN;
```

the "not-a-number" constant

If either parameter is null, then the method returns a Java constant named `Double.NaN`. This is a constant that a program uses to indicate that a double value is "not a number."

Another special case for the `distance` method is the possibility of a numerical overflow. The numbers obtained during a computation may go above the largest `double` number or below the smallest `double` number. These numbers are pretty large, but the possibility of overflow still exists. When an arithmetic expression with double numbers goes beyond the legal range, Java assigns a special constant to the answer. The constant is named `Double.POSITIVE_INFINITY` if it is too large (above about 1.7^{308}), and it is named `Double.NEGATIVE_INFINITY` if it is too small (below about -1.7^{308}). Of course, these constants are not really "infinity." They are merely indications to the programmer that a computation has overflowed. In the distance method, we indicate the possibility of overflow with the following comment:

the "infinity" constant

> **Note:**
> The answer is `Double.POSITIVE_INFINITY` if the distance calculation overflows. The answer is `Double.NaN` if either `Location` is null.

The Return Value of a Method May Be an Object

The return value of a method may also be an object, such as a `Location` object. For example, the `Location` class has this static method that creates and returns a new location that is halfway between two other locations. The method's specification and implementation are shown at the top of the next page.

- **midpoint**

  `public static Location midpoint(Location p1, Location p2)`

  Generates and returns a `Location` halfway between two others.

  Parameters:

  `p1` – the first `Location`

  `p2` – the second `Location`

  Returns:

  a `Location` that is halfway between `p1` and `p2`

  Note:

  The answer is null if either `Location` is null.

```
public static Location midpoint(Location p1, Location p2)
{
   double xMid, yMid;

   // Check whether one of the Locations is null.
   if ((p1 == null)  ||  (p2 == null))
      return null;

   // Compute the x and y midpoints.
   xMid = (p1.x/2) + (p2.x/2);
   yMid = (p1.y/2) + (p2.y/2);

   // Create a new Location and return it.
   Location answer = new Location(xMid, yMid);
   return answer;
}
```

The method creates a new location using the local variable `answer`, and then returns this location. Often the return value is stored in a local variable such as `answer`, but not always. For example, we could have eliminated `answer` by combining the last two statements in our implementation to a single statement:

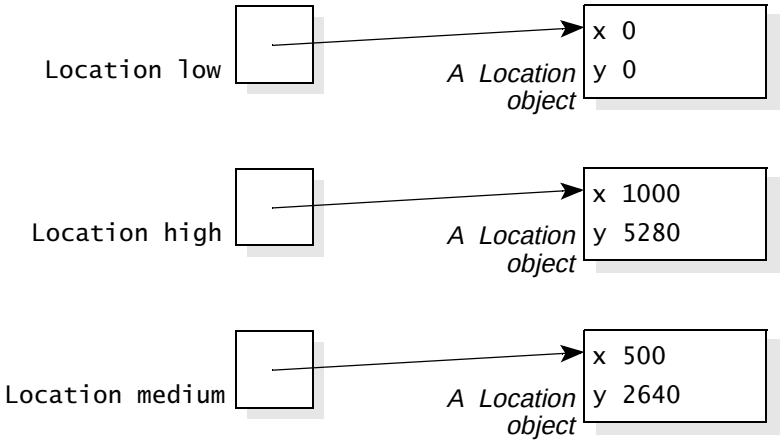
```
return new Location(xMid, yMid);
```

Either way—with or without the local variable—is fine.

Here's an example to show how the static `midpoint` method is used. The method creates two locations and then computes their midpoint:

```
Location low = new Location(0, 0);
Location high = new Location(1000, 5280);
Location medium = Location.midpoint(low, high);
```

In this example, the answer from the `midpoint` method is stored in a variable called `medium`. After the three statements, we have three locations, drawn at the top of the next page.



TIP

Programming Tip: How to Choose the Names of Methods

Accessor methods: The name of a boolean accessor method will usually begin with "is" followed by an adjective (such as "isOn"). Methods that convert to another kind of data start with "to" (such as "toString"). Other accessor methods start with "get" or some other verb followed by a noun that describes the return value (such as "getFlow").

Modification methods: A modification method can be named by a descriptive verb (such as "shift") or or a short verb phrase (such as "shutOff").

Static methods that return a value: Try to use a noun that describes the return object (such as "distance" or "midpoint").

Rules like these make it easier to determine the purpose of a method.

Java's Object Type

One of the `Location` methods is an accessor method called `equals` with this heading:

```
public boolean equals(Object obj)
```

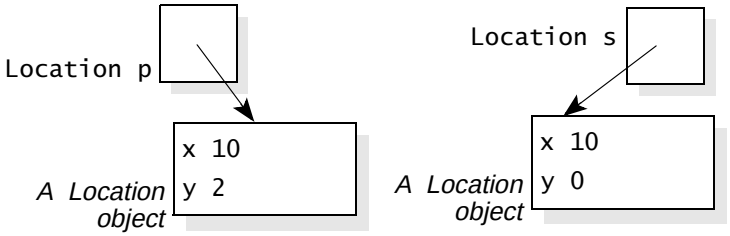
An accessor method with this name has a special meaning in Java. Before we discuss that meaning, you need to know a bit about the parameter type "`Object`." In Java, `Object` is a kind of "super data type" that encompasses all data except the eight primitive types. So a primitive variable (`byte`, `short`, `int`, `long`, `char`, `float`, `double`, or `boolean`) is *not* an `Object`, but everything else is. A `String` is an `Object`, a `Location` is an `Object`, even an array is an `Object`.

Using and Implementing an Equals Method

As your programming progresses, you'll learn a lot about Java's `Object` type, but to start you need just a few common patterns that use `Object`. For example, many classes implement an `equals` method with the heading that we have seen. An `equals` method has one argument: an `Object` called `obj`. The method should return `true` if `obj` has the same value as the object that activated the method. Otherwise, the method returns `false`. Here is an example to show how the `equals` method works for the `Location` class:

```
Location p = new Location(10, 2); // Declare p at coordinates (10,2)
Location s = new Location(10, 0); // Declare s at coordinates (10,0)
```

After these two declarations, we have two separate locations:

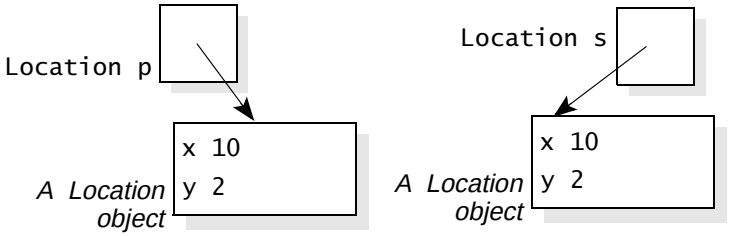


In this example, `p` and `s` refer to two separate objects with different values (their y coordinates are different), so both `p.equals(s)` and `s.equals(p)` are `false`.

Here's a slightly different example:

```
Location p = new Location(10, 2); // Declare p at coordinates (10,2)
Location s = new Location(10, 0); // Declare s at coordinates (10,0)
s.shift(0, 2);                    // Move s to (10,2)
```

We have the same two declarations, but afterward we shift the y coordinate of `s` so that the two separate locations have identical values, like this:



Now `p` and `s` refer to identical locations, so both `p.equals(s)` and `s.equals(p)` are `true`. However, the test `(p == s)` is still `false`. Remember that `(p == s)` returns `true` only if `p` and `s` refer to the exact same location (as opposed to two separate locations that happen to contain identical values).

a location can be compared to any object

The argument to the `equals` method can be any object, not just a location. For example, we can try to compare a location with a string, like this:

```
Location p = new Location(10, 2);
System.out.println(p.equals("10, 2"); // Prints false.
```

This example prints `false`; a `Location` object is not equal to the string `"10, 2"` even if they are similar. You can also test to see whether a location is equal to null, like this:

```
Location p = new Location(10, 2);
System.out.println(p.equals(null)); // Prints false.
```

The location is not null, so the result of `p.equals(null)` is `false`. Be careful with the last example: The argument to `p.equals` may be null and the answer will be `false`. However, when `p` itself is null, it is a programming error to activate any method of `p`. Trying to activate `p.equals` when `p` is null results in a `NullPointerException` (see page 51).

implementing an equals method

Now you know how to use an `equals` method. How do you write an `equals` method so that it returns `true` when its argument has the same value as the object that activates the method? A typical implementation follows an outline that is used for the `equals` method of the `Location` class, as shown here:

```
public boolean equals(Object obj)
{
   if (obj is actually a Location)
   {
        Figure out whether the location that obj refers to has the same
        value as the location that activated this method. Return true if
        they are the same, otherwise return false.
   }
   else
      return false;
}
```

the instanceof operator

The method starts by determining whether `obj` actually refers to a `Location` object. In pseudocode we wrote this as "`obj` is actually a `Location`". In Java, this is accomplished with the test `(obj instanceof Location)`. This test uses the keyword `instanceof`, which is a boolean operator. On the left of the operator is a variable, such as `obj`. On the right of the operator is a class name, such as `Location`. The test returns `true` if it is valid to convert the object (`obj`) to the given data type (`Location`). In our example, suppose that `obj` does not refer to a valid `Location`. It might be some other type of object, or perhaps it is simply null. In either case, we go to the else-statement and return `false`.

On the other hand, suppose that `(obj instanceof Location)` is `true`, so the code enters the first part of the if-statement. Then `obj` does refer to a `Location`

object. We need to determine whether the x and y coordinates of `obj` are the same as the location that activated the method. Unfortunately, we can't just look at `obj.x` and `obj.y` because the compiler thinks of `obj` as a bare object with no x and y instance variables. The solution is an expression `(Location) obj`. This expression is called a *typecast*, as if we were pouring `obj` into a casting mold that creates a `Location` object. The expression can be used to initialize a `Location` reference variable, like this:

```
Location candidate = (Location) obj;
```

The **typecast**, on the right side of the declaration, consists of the new data type `(Location)` in parentheses, followed by the reference variable that is being cast. After this declaration, `candidate` is a reference variable that refers to the same object that `obj` refers to. However, the compiler *does* know that `candidate` refers to a `Location` object, so we can look at `candidate.x` and `candidate.y` to see if they are the same as the x and y coordinates of the object that activated the `equals` method. The complete implementation looks like this:

```
public boolean equals(Object obj)
{
   if (obj instanceof Location)
   {
      Location candidate = (Location) obj;
      return (candidate.x == x) && (candidate.y == y);
   }
   else
      return false;
}
```

The implementation has the return statement:

```
return (candidate.x == x) && (candidate.y == y);
```

The boolean expression in this return statement is `true` if `candidate.x` and `candidate.y` are the same as the instance variables `x` and `y`. As with any method, these instance variables come from the object that activated the method. For future reference, the details of using a typecast are given in Figure 2.7.

Pitfall: Class Cast Exception

PITFALL

Suppose that you have a variable such as `obj`, which is an `Object`. You can try a typecast to use the object as if it were another type. For example, we used the typecast `Location candidate = (Location) obj`.

What happens if `obj` doesn't actually refer to a `Location` object? The result is a runtime exception called `ClassCastException`. To avoid this, you must ensure that a typecast is valid before trying to execute the cast. For example, the `instanceof` operator can validate the actual type of an object before a typecast.

FIGURE 2.7 Typecasts

A Simple Pattern for Typecasting an Object

A common situation in Java programming is a variable or other expression that is an `Object`, but the program needs to treat the `Object` as a specific data type such as `Location`. The problem is that when a variable is declared as an `Object`, that variable cannot immediately be used as if it were a `Location` (or some other type). For example, consider the parameter `obj` in the equals method of the `Location` class:

```
public boolean equals(Object obj)
```

Within the implementation of the `equals` method, we need to treat `obj` as a `Location` rather than a mere `Object`. The solution has two parts: (1) Check that `obj` does indeed refer to a valid `Location`, and (2) Declare a new variable of type `Location`, and initialize this new variable to refer to the same object that `obj` refers to, like this:

```
public boolean equals(Object obj)
{
   if (obj instanceof Location)
   {
      Location candidate = (Location) obj;
      ...
```

The parameter, obj, is an Object

Use the instanceof operator to check that obj is a valid Location

After this declaration, candidate refers to the Location object that obj also refers to.

The expression `(Location) obj`, used in the declaration of `candidate`, is a typecast to tell the compiler that `obj` may be used as a `Location`.

Every Class Has an Equals Method

You may write a class without an `equals` method, but Java automatically provides an `equals` method anyway. The `equals` method that Java provides is actually taken from the `Object` class, and it works exactly like the == operator. In other words, it returns `true` only when the two objects are the exact same object—but it returns `false` for two separate objects that happen to have the same values for their instance variables.

Using and Implementing a Clone Method

Another feature of our `Location` class is a method with this heading:

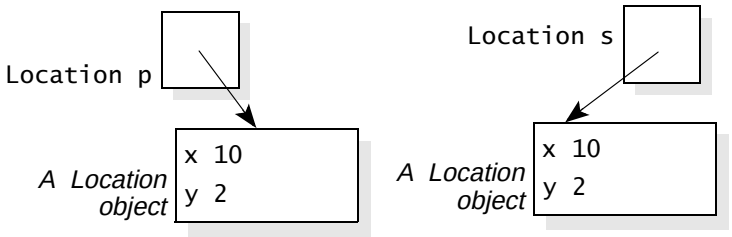
```
public Object clone( )
```

The purpose of a `clone` method is to create a copy of an object. The copy is separate from the original, so that subsequent changes to the copy won't change

the original, nor will subsequent changes to the original change the copy. Here's an example showing how the `clone` method is used for the `Location` class:

```
Location p = new Location(10, 2);    // Declare p at (10,2)
Location s = (Location) p.clone( ); // Initialize as a copy of p
```

The expression `p.clone( )` activates the `clone` method for `p`. The method creates and returns an exact copy of `p`, which we use to initialize the new location `s`. After these two declarations, we have two separate locations, as shown in this picture:



As you can see, `s` and `p` have the same values for their instance variables, but the two objects are separate. Changes to `p` will not affect `s`, nor will changes to `s` affect `p`.

Pitfall: A Typecast Is Needed to Use the Clone Return Value

PITFALL

The data type of the return value of the `clone` method is actually an `Object` and not a `Location`. This is a requirement of Java. Because of this requirement, we usually cannot use the `clone` return value directly. For example, we cannot write a declaration:

```
Location s = p.clone( );
```

this has a compile-time error

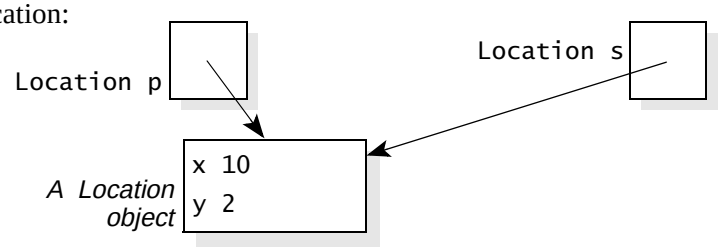
Instead, we must apply a typecast to the `clone` return value, converting it to a `Location` before we use it to initialize the new variable `s`, like this:

```
Location s = (Location) p.clone( );
```

Cloning is considerably different than using an assignment statement. For example, consider this code that does not make a clone:

```
Location p = new Location(10, 2); // Declare p at coordinates (10,2)
Location s = p;                   // Declare s and make it refer
                                  // to the same object that p
                                  // refers to
```

After these two declarations, we have just one location, and both variables refer to this location:



This is the situation with an ordinary assignment. Subsequent changes to the object that p refers to will affect the object that s refers to, because there is only one object.

implementing a clone method

You now know how to use a `clone` method. How do you implement such a method? You should follow a three-step pattern outlined here:

1. Modify the class head. You must add the words "`implements Cloneable`" in the class head, as shown here for the `Location` class:

```
public class Location implements Cloneable
```

The modification informs the Java compiler that you plan to implement certain features that are specified elsewhere in a format called an *interface*. The full meaning of interfaces will be discussed in Chapter 5. At the moment, it is enough to know that `implements Cloneable` is necessary when you implement a `clone` method.

By the way, "Cloneable" is a misspelling of "Clonable." Some future version of Java may correct the spelling, but for now it's nice to know that spell checkers haven't completely taken over the world.

2. Use `super.clone` to make a copy. The implementation of a `clone` method should begin by making a copy of the object that activated the method. The best way to make the copy is to follow this pattern from the `Location` class:

```
public Object clone( )
{ // Clone a Location object.
   Location answer;

   try
   {
      answer = (Location) super.clone( );
   }
   catch (CloneNotSupportedException e)
   {
      throw new RuntimeException
      ("This class does not implement Cloneable.");
   }
   ...
```

In an actual implementation, you would use the name of your own class (rather than `Location`), but otherwise you should follow this pattern exactly.

It's useful to know what's happening in this pattern. The pattern starts by declaring a local `Location` variable called `answer`. We then have this block:

```
try
{
   answer = (Location) super.clone( );
}
```

This is an example of a *try block*. If you plan extensive use of Java exceptions, then you should read all about try blocks in Appendix C. But for your first try block, all you need to know is that the code in the try block is executed, and the try block will be able to handle some of the possible exceptions that may arise in the code. In this example, the try block has just one assignment statement: `answer = (Location) super.clone( )`. The right side of the assignment activates a method called `super.clone( )`. This is actually the `clone` method from Java's `Object` type. It checks that the `Location` class specifies that it "`implements Cloneable`," and then correctly makes a copy of the location, assigning the result to the local variable `answer`.

After the try block is a sequence of one or more *catch blocks*. Each catch block can catch and handle an exception that may arise in the try block. Our example has one catch block:

```
catch (CloneNotSupportedException e)
{
   throw new RuntimeException
   ("This class does not implement Cloneable.");
}
```

This catch block will handle a `CloneNotSupportedException`. This exception is thrown by the `clone` method from Java's `Object` type when a programmer tries to call `super.clone( )`, without including the `implements Cloneable` clause as part of the class definition. The best solution is to throw a new `RuntimeException`, which is the general exception used to indicate a programmer error.

Anyway, after the try and catch blocks, the local variable `answer` refers to an exact copy of the location that activated the `clone` method, and we can move to the third part of the `clone` implementation.

3. Make necessary modifications and return. The answer is present, and it refers to an exact copy of the object that activated the `clone` method. Sometimes, further modifications must be made to the copy before returning. You'll see the reasons for such modifications in Chapter 3. However, the `Location` clone needs no modifications, so the end of the `clone` method consists of just the return statement: `return answer`.

The complete `clone` implementation for the `Location` class looks like this, including an indication of the likely cause of the `CloneNotSupportedException`:

```
public Object clone( )
{  // Clone a Location object.
   Location answer;

   try
   {
      answer = (Location) super.clone( );
   }
   catch (CloneNotSupportedException e)
   {  // This exception should not occur. But if it does, it would indicate a
      // programming error that made super.clone unavailable. The
      // most common cause would be forgetting the
      // "implements Cloneable" clause at the start of the class.
      throw new RuntimeException
      ("This class does not implement Cloneable.");
   }

   return answer;
}
```

The method returns the local variable, `answer`, which is a `Location` object. This is allowed, even though the return type of the `clone` method is `Object`. A Java `Object` may be anything except the eight primitive types. It might be better if the actual return type of the `clone` method was `Location` rather than `Object`. Using `Location` for the return type would be more accurate and would make the `clone` method easier to use (without having to put a typecast with every usage). Unfortunately, the improvement is not allowed: The return type of the `clone` method must be `Object`.

Programming Tip: Always Use super.clone for Your Clone Methods

Perhaps you thought of a simpler way to create a clone. Instead of using `super.clone` and the try/catch blocks, could you write this code:

```
Location answer = new Location(x, y);
return answer;
```

You could combine these into one statement: `return new Location(x, y)`. This creates and returns a new location, using the instance variables `x` and `y` to initialize the new location. These instance variables come from the location that activated the `clone` method, so `answer` will indeed be a copy of that location. This is a nice direct approach, but the direct approach will encounter problems when we start building new classes that are based on existing classes (See page 655). Therefore, it is better to stick with the pattern that uses `super.clone` and a try/catch block.

Programming Tip: When to Throw a Runtime Exception

TIP

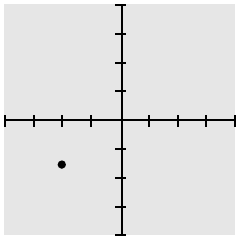
A `RuntimeException` is thrown to indicate a programming error. For example, the `clone` method from Java's `Object` type is not supposed to be called by an object unless that object's class has implemented the `Cloneable` interface. If we detect that the exception has been thrown by the `Object clone` method, then the programmer probably forgot to include the "`implements Cloneable`" clause.

When you throw a `RuntimeException`, include a message with your best guess about the programming error.

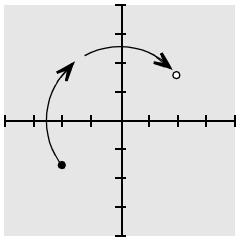
A Demonstration Program for the Location Class

As one last example, let's look at a program that creates two locations called `still` and `mobile`. Both are initially placed at $x = -2$ and $y = -1.5$, as shown in Figure 2.8(a). To be more precise, the `still` location is placed at this spot, and then `mobile` is initialized as a clone of the `still` location. Because the `mobile` location is a clone, later changes to one location will not affect the other.

FIGURE 2.8



(a) The still and mobile locations are initially placed at x = –2 and y = –1.5.



(b) The still (black) location remains at the same spot, but the mobile (white) location undergoes two 90° rotations.

The program prints some information about both locations, and then the `mobile` location undergoes two 90° rotations as shown in Figure 2.8(b). The information about the locations is then printed a second time.

The complete program is shown in Figure 2.9 on page 82. Pay particular attention to the `specifiedRotation` method, which illustrates some important principles about what happens when a parameter is changed within a method. We'll look at those principles in a moment, but first let's take a look at the complete output from the program, as shown here:

```
The still location is at:  (x=-2.0 y=-1.5)
The mobile location is at: (x=-2.0 y=-1.5)
Distance between them:  0.0
These two locations have equal coordinates.

I will rotate one location by two 90 degree turns.
The still location is at:  (x=-2.0 y=-1.5)
The mobile location is at: (x=2.0 y=1.5)
Distance between them:  5.0
These two locations have different coordinates.
```

FIGURE 2.9 A Demonstration Program for the `Location` Class

Java Application Program

```
// FILE: LocationDemonstration.java
// This small demonstration program shows how to use the Location class
// from the edu.colorado.geometry package.

import edu.colorado.geometry.Location;

class LocationDemonstration
{
   public static void main(String[ ] args)
   {
      final double STILL_X = -2.0;
      final double STILL_Y = -1.5;
      final int ROTATIONS = 2;

      Location still = new Location(STILL_X, STILL_Y);
      Location mobile = (Location) still.clone( );
      printData(still, mobile);

      System.out.println("I will rotate one location by two 90 degree turns.");
      specifiedRotation(mobile, ROTATIONS);
      printData(still, mobile);
   }

   // Rotate a Location p by a specified number of 90 degree clockwise rotations.
   public static void specifiedRotation(Location p, int n)
   {
      while (n > 0)
      {
         p.rotate90( );
         n--;
      }
   }

   // Print some information about two locations: s (a "still" location) and m (a "mobile" location).
   public static void printData(Location s, Location m)
   {
      System.out.println("The still location is at:  " + s.toString( ));
      System.out.println("The mobile location is at: " + m.toString( ));
      System.out.println("Distance between them:  " + Location.distance(s, m));
      if (s.equals(m))
         System.out.println("These two locations have equal coordinates.");
      else
         System.out.println("These two locations have different coordinates.");
      System.out.println( );
   }
}
```

What Happens When a Parameter Is Changed within a Method?

Let's examine the program's `specifiedRotation` method to see exactly what happens when a parameter is changed within a method. Here is the method's implementation:

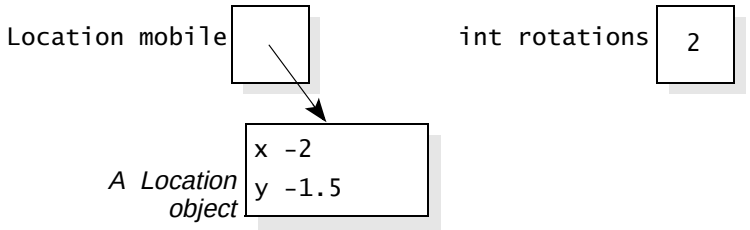
```
// Rotate a Location p by a number of 90 degree clockwise rotations.
public static void specifiedRotation(Location p, int n)
{
   while (n > 0)
   {
      p.rotate90( );
      n--;
   }
}
```

The method rotates the location `p` by `n` 90° clockwise rotations.

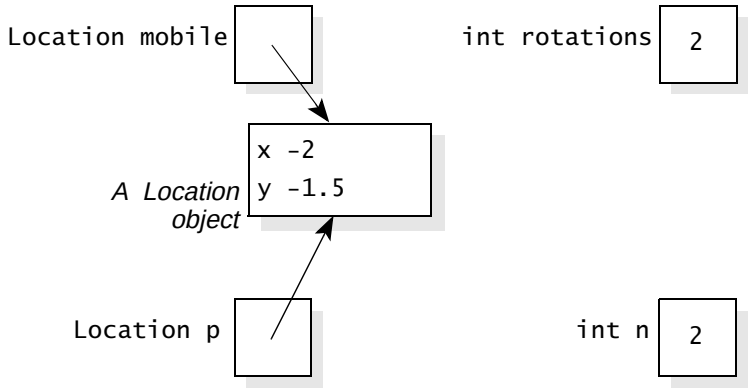
In Java, a parameter that is a reference variable (such as the `Location p`) has different behavior than a parameter that is one of the eight primitive types (such as `int n`). Here is the difference:

- When a parameter is one of the eight primitive types, the actual argument provides an initial value for that parameter. To be more precise, the parameter is implemented as a local variable of the method and the argument is used to initialize this variable. Changes that are made to the parameter do not affect the actual argument.
- When a parameter is a reference variable, the parameter is initialized so that it refers to the same object as the actual argument. Subsequent changes to this object do affect the actual argument's object.

For example, suppose that we have initialized a location called `mobile` at the coordinates $x = -2$ and $y = -1.5$. Suppose that we also have an integer variable called `rotations`, with a value of 2, as shown here:



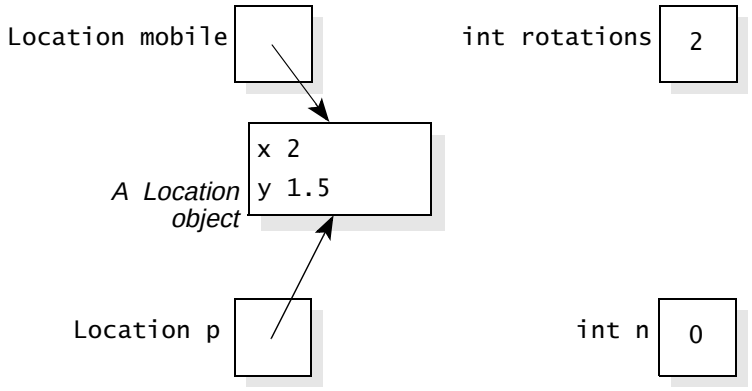
Now, suppose the program activates `specifiedRotation(mobile, rotations)`. The method's first parameter, `p`, is initialized to refer to the same location that `mobile` refers to. And the method's second parameter, `n`, is initialized with the value 2 (from the `rotations` argument). So, when the method begins its work, the situation looks like the picture at the top of the next page.



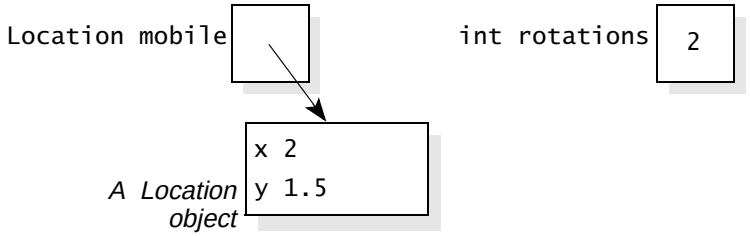
The method now executes its loop:

```
while (n > 0)
{
   p.rotate90( );
   n--;
}
```

The first iteration of the loop rotates the location by 90° and decreases `n` to 1. The second iteration does another rotation of the location and decreases `n` to 0. Now the loop ends, with these values for the variables:



Notice the difference between the two kinds of parameters. The integer parameter `n` has changed to zero without affecting the actual argument `rotations`. On the other hand, rotating the location `p` has changed the object that `mobile` refers to. When the method returns, the parameters `p` and `n` disappear, leaving the situation shown at the top of the next page.



Java Parameters

The eight primitive types (`byte`, `short`, `int`, `long`, `char`, `float`, `double`, or `boolean`)**:** The parameter is initialized with the value of the argument. Subsequent changes to the parameter do not affect the argument.

Reference variables: When a parameter is a reference variable, the parameter is initialized so that it refers to the same object as the actual argument. Subsequent changes to this object do affect the actual argument's object.

Self-Test Exercises

17. Write some code that declares two locations: one at the origin and the other at the coordinates $x = 1$ and $y = 1$. Print the distance between the two locations, then create a third location that is at the midpoint between the first two locations.
18. The location's `distance` method is a static method. What effect does this have on how the method is used? What effect does this have on how the method is implemented?
19. What is the purpose of the Java constant `Double.NaN`?
20. What is the result when you add two `double` numbers and the answer is larger than the largest possible `double` number?
21. In the `midpoint` method we used the expression `(p1.x/2) + (p2.x/2)`. Can you think of a reason why this expression is better than `(p1.x + p2.x)/2`?
22. Implement an `equals` method for the `Throttle` class from Section 2.1.
23. If you don't implement an `equals` method for a class, then Java automatically provides one. What does the automatic `equals` method do?
24. Implement a `clone` method for the `Throttle` class from Section 2.1.
25. When should a program throw a `RuntimeException`?

26. Suppose that a method has an `int` parameter called `x`, and the body of the method changes `x` to zero. When the method is activated, what happens to the argument that corresponds to `x`?
27. Suppose that a method has a `Location` parameter called `x`, and the body of the method activates `x.rotate90( )`. When the method is activated, what happens to the argument that corresponds to `x`?

CHAPTER SUMMARY

- In Java, object-oriented programming (OOP) is supported by implementing *classes*. Each class defines a collection of data, called its *instance variables*. In addition, a class has the ability to include two other items: *constructors* and *methods*. Constructors are designed to provide initial values to the class's data; methods are designed to manipulate the data. Taken together, the instance variables, constructors, and methods of a class are called the class *members*.
- We generally use *private instance variables* and *public methods*. This approach supports information hiding by forbidding data components of a class to be directly accessed outside of the class.
- A new class can be implemented in a Java package that is provided to other programmers to use. The package includes documentation to tell programmers what the new class does without revealing the details of how the new class is implemented.
- A program uses a class by creating new objects of that class, and activating these objects' methods through *reference variables*.
- When a method is activated, each of its parameters is initialized. If a parameter is one of the eight primitive types, then the parameter is initialized by the value of the argument, and subsequent changes to the parameter do not affect the actual argument. On the other hand, when a parameter is a reference variable, the parameter is initialized so that it refers to the same object as the actual argument. Subsequent changes to this object do affect the actual argument's object.
- Java programmers must understand how these items work for classes:
  - — the assignment operator (`x = y`)
  - — the equality test (`x == y`)
  - — a `clone` method to create a copy of an object
  - — an `equals` method to test whether two separate objects are equal to each other

Solutions to Self-Test Exercises

1. We have used *private* instance variables, *public* constructors, and *public* methods.

2. In this solution, the assignment to `position` is not really needed since `position` will be given its default value of zero before the constructor executes. However, including the assignment makes it clear that we intended for `position` to start at zero:

```
public Throttle( )
{
   top = 1;
   position = 0;
}
```

3. Notice that our solution has the precondition that `size` is positive, and `initial` lies in the range from zero to `size`.

```
public Throttle(int size, int initial)
{
   if (size <= 0)
      throw new
      IllegalArgumentException
      ("Size <= 0:" + size);
   if (initial < 0)
      throw new
      IllegalArgumentException
      ("Initial < 0:" + initial);
   if (initial > size)
      throw new
      IllegalArgumentException
      ("Initial too big:" + initial);
   top = size;
   position = initial;
}
```

4. The method implementation is:

```
public boolean isAboveHalf( )
{
   return (getFlow( ) > 0.5);
}
```

5. You'll find part of a solution in Figure 13.1 on page 618.

6. The program should include the following statements:

```
Throttle exercise = new Throttle(6);
exercise.shift(3);
System.out.println(exercise.flow( ));
```

7. The `control` should be assigned the value of null. By the way, if it is an instance variable of a class, then it is initialized to null.

8. A `NullPointerException` is thrown.

9. Both `t1` and `t2` refer to the same throttle, which has been shifted up 42 positions. So the output is 0.42.

10. At the end of the code (`t1 == t2`) is `true` since there is only one throttle that both variables refer to.

11. Here is the code (and at the end `t1==t2` is `false` since there are two separate throttles):

```
Throttle t1;
Throttle t2;
t1 = new Throttle(100);
t2 = new Throttle(100);
t1.shift(42);
t2.shift(42);
```

12. `com.knafn.statistics`

13. Underneath your `classes` directory, create a subdirectory `com`. Underneath `com` create a subdirectory `knafn`. Underneath `knafn` create a subdirectory `statistics`. Your package is placed in the `statistics` subdirectory.

14. `import com.knafn.statistics.*;`

15. Java automatically imports `java.lang`; no explicit import statement is needed.

16. Public access is obtained with the keyword `public`, and it allows access by any program. Private access is obtained with the keyword `private`, and it allows access only by the methods of the class. Package access is obtained with no keyword, and it allows access within the package but not elsewhere.

17. Here is the code:

```
Location p1 = new Location(0, 0);
Location p2 = new Location(1, 1);
System.out.println
   (Location.distance(p1, p2));
Location p3 =
  Location.midpoint(p1, p2);
```

18. A static method is not activated by any one object. Instead, the class name is placed in front of the method to activate it. For example, the distance between two locations `p1` and `p2` is computed by:

```
Location.distance(p1, p2);
```

Within the implementation of a static method, we cannot directly refer to the instance variables.

19. The constant `Double.NaN` is used when there is no valid number to store in a double variable ("not a number").

20. The result is the constant `Double.POSITIVE_INFINITY`.

21. The alternative `(p1.x + p2.x)/2` has a subexpression `p1.x + p2.x` which could result in an overflow.

22. Here is the implementation for the throttle:

```
public boolean equals(Object obj)
{
   if (obj instanceof Throttle)
   {
     Throttle candidate = (Throttle) obj;
     return
        (candidate.top==top)
        &&
        (candidate.position==position);
   }
   else
     return false;
}
```

23. The automatic `equals` method returns `true` only when the two objects are the exact same object (as opposed to two separate objects that have the same value).

24. The solution is the same as the `Location` clone on page 65, but change the `Location` type to `Throttle`.

25. A `runtimeException` indicates a programming error. When you throw a `RuntimeException`, you should provide an indication of the most likely cause of the error.

26. The argument remains unchanged.

27. The object that the argument refers to has been rotated 90°.

PROGRAMMING PROJECTS

1 Specify, design, and implement a class that can be used in a program that simulates a combination lock. The lock has a circular knob, with the numbers 0 through 39 marked on the edge, and it has a three-number combination, which we'll call *x*, *y*, *z*. To open the lock, you must turn the knob clockwise at least one entire revolution, stopping with *x* at the top; then turn the knob counterclockwise, stopping the *second* time that *y* appears at the top; finally turn the knob clockwise again, stopping the next time that *z* appears at the top. At this point, you may open the lock.

Your `Lock` class should have a constructor that initializes the three-number combination. Also provide methods:

(a) To alter the lock's combination to a new three-number combination
(b) To turn the knob in a given direction until a specified number appears at the top
(c) To close the lock

(d) To attempt to open the lock
(e) To inquire the status of the lock (open or shut)
(f) To tell what number is currently at the top

2 Specify, design, and implement a class called `Statistician`. After a statistician is initialized, it can be given a sequence of double numbers. Each number in the sequence is given to the statistician by activating a method called `nextNumber`. For example, we can declare a statistician called `s`, and then give it the sequence of numbers 1.1, –2.4, 0.8 as shown here:

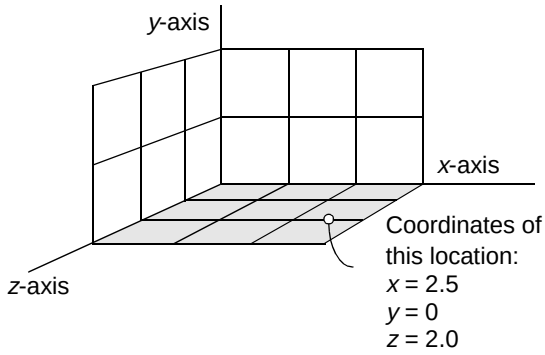
```
Statistician s = new Statistician;
s.nextNumber(1.1);
s.nextNumber(-2.4);
s.nextNumber(0.8);
```

After a sequence has been given to a statistician, there are various methods to obtain information about the sequence. Include methods that will provide the length of the sequence, the last number of the sequence, the sum of all the numbers in the sequence, the arithmetic mean of the numbers (i.e., the sum of the numbers divided by the length of the sequence), the smallest number in the sequence, and the largest number in the sequence. Notice that the length and sum methods can be called at any time, even if there are no numbers in the sequence. In this case of an "empty" sequence, both length and sum will be zero. The other methods should return `Double.NaN` if they are called for an empty sequence.

Notes: Do not try to store the entire sequence (because you don't know how long this sequence will be). Instead, just store the necessary information about the sequence: What is the sequence length, what is the sum of the numbers in the sequence, what are the last, smallest, and largest numbers? Each of these pieces of information can be stored in a private instance variable that is updated whenever `nextNumber` is activated.

3 Write a new static method to allow you to "add" two statisticians from the previous project. If `s1` and `s2` are two statisticians, then the result of adding them should be a new statistician that behaves as if it had all of the numbers of `s1` followed by all of the numbers of `s2`.

4 Specify, design, and implement a class that can be used to keep track of the position of a location in three-dimensional space. For example, consider the location drawn here:



The location shown in the picture has three coordinates: $x = 2.5$, $y = 0$, and $z = 2.0$. Include methods to set a location to a specified point, to shift a location a given amount along one of the axes, and to retrieve the coordinates of a location. Also provide methods that will rotate the location by a specified angle around a specified axis.

To compute these rotations, you will need a bit of trigonometry. Suppose you have a location with coordinates x, y, and z. After rotating this location by an angle θ, the location will have new coordinates, which we'll call x', y', and z'. The equations for the new coordinates use the `java.lang` methods `Math.sin` and `Math.cos`, as shown here:

After a θ rotation around the x-axis:

$$x' = x$$
$$y' = y\cos(\theta) - z\sin(\theta)$$
$$z' = y\sin(\theta) + z\cos(\theta)$$

After a θ rotation around the y-axis:

$$x' = x\cos(\theta) + z\sin(\theta)$$
$$y' = y$$
$$z' = -x\sin(\theta) + z\cos(\theta)$$

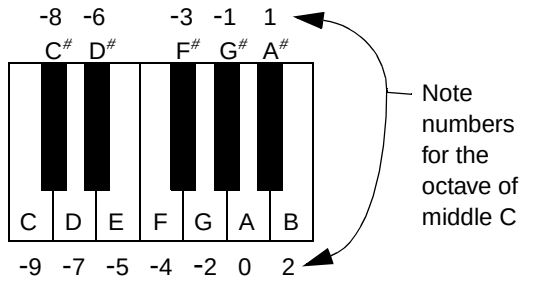
After a θ rotation around the z-axis:

$$x' = x\cos(\theta) - y\sin(\theta)$$
$$y' = x\sin(\theta) + y\cos(\theta)$$
$$z' = z$$

5 In three-dimensional space, a line segment is defined by its two endpoints. Specify, design, and implement a class for a line segment. The class should have two private instance variables that are 3D locations from the previous project.

6 Specify, design, and implement a class for a card in a deck of playing cards. The class should contain methods for setting and retrieving the suit and rank of a card.

7 Specify, design, and implement a class that can be used to hold information about a musical note. A programmer should be able to set and retrieve the length of the note and the value of the note. The length of a note may be a sixteenth note, eighth note, quarter note, half note, or whole note. A value is specified by indicating how far the note lies above or below the A note that orchestras use in tuning. In counting "how far," you should include both the white and black notes on a piano. For example, the note numbers for the octave beginning at middle C are shown here:



The constructor should set a note to a middle C quarter note. Include methods to set a note to a specified length and value. Write methods to retrieve information about a note, including methods to tell you the letter of the note (A, B, C, etc.), whether the note is natural or sharp (i.e., white or black on the piano), and the frequency of a note in hertz. To calculate the frequency, use the formula $440 \times 2^{n/12}$, where *n* is the note number. Feel free to include other useful methods.

8 A one-variable **quadratic expression** is an arithmetic expression of the form $ax^2 + bx + c$, where *a*, *b*, and *c* are some fixed numbers (called the **coefficients**) and *x* is a variable that can take on different values. Specify, design, and implement a class that can store information about a quadratic expression. The constructor should set all three coefficients to zero, and another method should allow you to change these coefficients. There should be accessor methods to retrieve the current values of the coefficients. There should also be a method to allow you to "evaluate" the quadratic expression at a particular value of *x* (i.e., the method has one parameter *x*, and returns the value of the expression $ax^2 + bx + c$).

Also write the following static methods to perform these indicated operations:

```
public static Quadratic sum(
    Quadratic q1,
    Quadratic q2
)
// Postcondition: The return value is the
// quadratic expression obtained by adding
// q1 and q2. For example, the c coefficient
// of the return value is the sum of q1's c
// coefficient and q2's c coefficient.

public static Quadratic scale(
    double r,
    Quadratic q
)
// Postcondition: The return value is the
// quadratic expression obtained by
// multiplying each of q's
// coefficients by the number r.
```

Notice that the first argument of the `scale` method is a double number (rather than a quadratic expression). For example, this allows the method activation `Quadratic.scale(3.14, q)` where `q` is a quadratic expression.

9 This project is a continuation of the previous project. For a quadratic expression such as $ax^2 + bx + c$, a **real root** is any double number *x* such that $ax^2 + bx + c = 0$. For example, the quadratic expression $2x^2 + 8x + 6$ has one of its

real roots at $x = -3$, because substituting $x = -3$ in the formula $2x^2 + 8x + 6$ yields the value:

$$2 \times (-3^2) + 8 \times (-3) + 6 = 0$$

There are six rules for finding the real roots of a quadratic expression:

(1) If *a*, *b*, and *c* are all zero, then every value of *x* is a real root.

(2) If *a* and *b* are zero, but *c* is nonzero, then there are no real roots.

(3) If *a* is zero, and *b* is nonzero, then the only real root is $x = -c/b$.

(4) If *a* is nonzero and $b^2 < 4ac$, then there are no real roots.

(5) If *a* is nonzero and $b^2 = 4ac$, then there is one real root $x = -b/2a$.

(6) If *a* is nonzero, and $b^2 > 4ac$, then there are two real roots:

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

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

Write a new method that returns the number of real roots of a quadratic expression. This answer could be 0, or 1, or 2, or infinity. In the case of an infinite number of real roots, have the method return 3. (Yes, we know that 3 is not infinity, but for this purpose it is close enough!) Write two other methods that calculate and return the real roots of a quadratic expression. The precondition for both methods is that the expression has at least one real root. If there are two real roots, then one of the methods returns the smaller of the two roots, and the other method returns the larger of the two roots. If every value of *x* is a real root, then both methods should return zero.

10 Specify, design, and implement a class that can be used to simulate a lunar lander, which is a small spaceship that transports astronauts from lunar orbit to the surface of the moon. When a lunar lander is constructed, the following items should be initialized as follows:

(1) Current fuel flow rate as a fraction of the maximum fuel flow (initially zero)

(2) Vertical speed of the lander (initially zero meters/sec)

(3) Altitude of the lander (specified as a parameter of the constructor)

(4) Amount of fuel (specified as a parameter of the constructor)

(5) Mass of the lander when it has no fuel (specified as a parameter of the constructor)

(6) Maximum fuel consumption rate (specified as a parameter of the constructor)

(7) Maximum thrust of the lander's engine (specified as a parameter of the constructor)

Don't worry about other properties (such as horizontal speed).

The lander has accessor methods that allow a program to retrieve the current values of any of the preceding seven items. There are only two modification methods, described below.

The first modification method changes the current fuel flow rate to a new value ranging from 0.0 to 1.0. This value is expressed as a fraction of the maximum fuel flow.

The second modification method simulates the passage of a small amount of time. This time, called *t*, is expressed in seconds and will typically be a small value such as 0.1 seconds. The method will update the first four values in the preceding list, to reflect the passage of *t* seconds. To implement this method, you will require a few physics formulas listed below. These formulas are only approximate, because some of the lander's values are changing during the simulated time period. But if the time span is kept short, these formulas will suffice.

Fuel flow rate: Normally, the fuel flow rate does not change during the passage of a small amount of time. But there is one exception: If the fuel flow rate is greater than zero, and the amount of fuel left is zero, then you should reset the fuel flow rate to zero (because there is no fuel to flow).

Velocity change: During *t* seconds, the velocity of the lander changes by approximately this amount (measured in meters/sec):

$$t \times \left(\frac{f}{m} - 1.62 \right)$$

The value *m* is the total mass of the lander, measured in kilograms (i.e., the mass of a lander with no fuel, plus the mass of any remaining fuel). The value *f* is the thrust of the lander's engine, measured in newtons. You can calculate *f* as the current fuel flow rate times the maximum thrust of the lander. The number –1.62 is the downward acceleration from gravity on the moon.

Altitude change: During *t* seconds, the altitude of the lander changes by $t \times v$ meters, where *v* is the vertical velocity of the lander (measured in meters/sec, with negative values downward).

Change in remaining fuel: During *t* seconds, the amount of remaining fuel is reduced by $t \times r \times c$ kilograms. The value of *r* is the current fuel flow rate, and *c* is the maximum fuel consumption (measured in kilograms per second).

We suggest that you calculate the changes to the four items in the order just listed. After all the changes have been made, there are two further adjustments. First, if the altitude has dropped below zero, then reset both altitude and velocity to zero (indicating that the ship has landed). Second, if the total amount of remaining fuel drops below zero, then reset this amount to zero (indicating that we have run out of fuel).

11 In this project you will design and implement a class that can generate a sequence of **pseudorandom** integers, which is a sequence that appears random in many ways. The approach uses the **linear congruence method**, explained below. The linear congruence method starts with a number called the **seed**. In addition to the seed, three other numbers are used in the linear congruence method, called the **multiplier**, the **increment**, and the **modulus**. The formula for generating a sequence of pseudorandom numbers is quite simple. The first number is:

```
(multiplier * seed + increment) % modulus
```

This formula uses the Java `%` operator, which computes the remainder from an integer division.

Each time a new random number is computed, the value of the seed is changed to that new number. For example, we could implement a pseudorandom number generator with `multiplier = 40`, `increment = 3641`, and `modulus = 729`. If we choose the seed to be `1`, then the sequence of numbers will proceed as shown here:

First number
= `(multiplier * seed + increment) % modulus`
= `(40 * 1 + 3641) % 729`
= `36`
and `36` becomes the new seed.

Next number
= `(multiplier * seed + increment) % modulus`
= `(40 * 36 + 3641) % 729`
= `707`
and `707` becomes the new seed.

These particular values for multiplier, increment, and modulus happen to be good choices. The pattern generated will not repeat until 729 different numbers have been produced. Other choices for the constants might not be so good.

For this project, design and implement a class that can generate a pseudorandom sequence in the manner described. The initial seed, multiplier, increment, and modulus should all be parameters of the constructor. There should also be a method to permit the seed to be changed, and a method to generate and return the next number in the pseudorandom sequence.

12 Add a new method to the random number class of the previous project. The new method generates the next pseudorandom number but does not return the number directly. Instead, the method returns this number divided by the modulus. (You will have to cast the modulus to a double number before carrying out the division; otherwise, the division will be an integer division, throwing away the remainder.)

The return value from this new member function is a pseudorandom double number in the range `[0..1)`. (The square bracket, '`[`', indicates that the range does include `0`, but the rounded parenthesis, '`)`', indicates that the range goes up to `1`, without actually including `1`.)

13 Run some experiments to determine the distribution of numbers returned by the new pseudorandom method from the previous project. Recall that this method returns a double number in the range `[0..1)`. Divide this range into ten intervals, and call the method one million times, producing a table such as shown here:

Range	Number of Occurrences
[0.0..0.1)	99889
[0.1..0.2)	100309
[0.2..0.3)	100070
[0.3..0.4)	99940
[0.4..0.5)	99584
[0.5..0.6)	100028
[0.6..0.7)	99669
[0.7..0.8)	100100
[0.8..0.9)	100107
[0.9..1.0)	100304

Run your experiment for different values of the multiplier, increment, and modulus. With good choices of the constants, you will end up with about 10% of the numbers in each interval. A pseudorandom number generator with this equal-interval behavior is called **uniformly distributed.**

14 This project is a continuation of the previous project. Many applications require pseudorandom number sequences that are *not* uniformly distributed. For example, a program that simulates the birth of babies can use random numbers for the birth weights of the newborns. But these birth weights should have a **Gaussian distribution.** In a Gaussian distribution, numbers form a bell-shaped curve in which values are more likely to fall in intervals near the center of the overall distribution. The exact probabilities of falling in a particular interval can be computed by knowing two numbers: (1) a number called the *variance*, which indicates how widely spread the distribution appears, and (2) the center of the overall distribution, called the *median*. For this kind of distribution, the median is equal to the arithmetic average (the *mean*) and equal to the most frequent value (the *mode*).

Generating a pseudorandom number sequence with an exact Gaussian distribution can be difficult, but there is a good way to approximate a Gaussian distribution using uniformly distributed random numbers in the range `[0..1)`. The approach is to generate three pseudorandom numbers r_1, r_2, and r_3, each of which is in the range `[0..1)`. These numbers are then combined to produce the next number in the Gaussian sequence. The formula to combine the numbers is:

Next number in the Gaussian sequence

$$= \text{median} + (2 \times (r_1 + r_2 + r_3) - 3) \times \text{variance}$$

Add a new method to the random number class, which can be used to produce a sequence of pseudorandom numbers with a Gaussian distribution.