The way in which one object gains access to an object of another class is an important design decision. It occurs when one class instantiates the objects of another, but the access can also be accomplished by passing one object to another as a method parameter.

In general, we want to minimize the number of dependencies among classes. The less dependent our classes are on each other, the less impact changes and errors will have on the system.

Dependencies among Objects of the Same Class

In some cases, a class depends on itself. That is, an object of one class interacts with another object of the same class. To accomplish this, a method of the class may accept as a parameter an object of the same class.

The `concat` method of the `String` class is an example of this situation. The method is executed through one `String` object and is passed another `String` object as a parameter. Here is an example:

```
str3 = str1.concat(str2);
```

The `String` object executing the method (`str1`) appends its characters to those of the `String` passed as a parameter (`str2`). A new `String` object is returned as a result and stored as `str3`.

The `RationalTester` program shown in Listing 5.10 on the next page demonstrates a similar situation. A rational number is a value that can be represented as a ratio of two integers (a fraction). The `RationalTester` program creates two objects representing rational numbers and then performs various operations on them to produce new rational numbers.

LISTING 5.10

```
//********************************************************************
//  RationalTester.java                 Java Foundations
//
//  Driver to exercise the use of multiple Rational objects.
//********************************************************************

public class RationalTester
{
   //-----------------------------------------------------------------
   //  Creates some rational number objects and performs various
   //  operations on them.
   //-----------------------------------------------------------------
```


LISTING 5.10 *continued*

```
   public static void main(String[] args)
   {
      RationalNumber r1 = new RationalNumber(6, 8);
      RationalNumber r2 = new RationalNumber(1, 3);
      RationalNumber r3, r4, r5, r6, r7;

      System.out.println("First rational number: " + r1);
      System.out.println("Second rational number: " + r2);

      if (r1.isLike(r2))
         System.out.println("r1 and r2 are equal.");
      else
         System.out.println("r1 and r2 are NOT equal.");

      r3 = r1.reciprocal();
      System.out.println("The reciprocal of r1 is: " + r3);

      r4 = r1.add(r2);
      r5 = r1.subtract(r2);
      r6 = r1.multiply(r2);
      r7 = r1.divide(r2);

      System.out.println("r1 + r2: " + r4);
      System.out.println("r1 - r2: " + r5);
      System.out.println("r1 * r2: " + r6);
      System.out.println("r1 / r2: " + r7);
   }
}
```

OUTPUT

```
First rational number: 3/4
Second rational number: 1/3
r1 and r2 are NOT equal.
The reciprocal of r1 is: 4/3
r1 + r2: 13/12
r1 - r2: 5/12
r1 * r2: 1/4
r1 / r2: 9/4
```

The `RationalNumber` class is shown in Listing 5.11 on the next page. As you examine this class, keep in mind that each object created from the `RationalNumber` class represents a single rational number. The `RationalNumber` class contains various operations on rational numbers, such as addition and subtraction.

The methods of the `RationalNumber` class, such as `add`, `subtract`, `multiply`, and `divide`, use the `RationalNumber` object that is executing the method as the first (left) operand and use the `RationalNumber` object passed as a parameter as the second (right) operand.

The `isLike` method of the `RationalNumber` class is used to determine whether two rational numbers are essentially equal. It's tempting, therefore, to call that method `equals`, similar to the method used to compare `String` objects (discussed in Chapter 4). However, in Chapter 8 we will discuss how the `equals` method is somewhat special due to inheritance, and we will note that it should be implemented in a particular way. Thus, to avoid confusion, we call this method `isLike` for now.

Note that some of the methods in the `RationalNumber` class, including `reduce` and `gcd`, are declared with private visibility. These methods are private because we don't want them executed directly from outside a `RationalNumber` object. They exist only to support the other services of the object.

Aggregation

KEY CONCEPT
An aggregate object is composed of other objects, forming a has-a relationship.

Some objects are made up of other objects. A car, for instance, is made up of its engine, its chassis, its wheels, and several other parts. Each of these other parts could be considered a separate object. Therefore, we can say that a car is an *aggregation*—it is composed, at least in part, of other objects. Aggregation is sometimes described as a *has-a relationship*. For instance, a car "has-a" chassis.

In the software world, we define an *aggregate object* as any object that contains references to other objects as instance data. For example, an `Account` object contains, among other things, a `String` object that represents the name of the account owner. We sometimes forget that strings are objects, but technically that makes each `Account` object an aggregate object.

Aggregation is a special type of dependency. That is, a class that is defined in part by another class is dependent on that class. The methods of the aggregate object generally invoke the methods of the objects of which it is composed.

LISTING 5.11

```
//********************************************************************
//  RationalNumber.java       Java Foundations
//
//  Represents one rational number with a numerator and denominator.
//********************************************************************

public class RationalNumber
{
   private int numerator, denominator;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the rational number by ensuring a nonzero
   //  denominator and making only the numerator signed.
   //-----------------------------------------------------------------
   public RationalNumber(int numer, int denom)
   {
      if (denom == 0)
         denom = 1;

      // Make the numerator "store" the sign
      if (denom < 0)
      {
         numer = numer * -1;
         denom = denom * -1;
      }

      numerator = numer;
      denominator = denom;

      reduce();
   }

   //-----------------------------------------------------------------
   //  Returns the numerator of this rational number.
   //-----------------------------------------------------------------
   public int getNumerator()
   {
      return numerator;
   }

   //-----------------------------------------------------------------
   //  Returns the denominator of this rational number.
   //-----------------------------------------------------------------
   public int getDenominator()
```

LISTING 5.11 *continued*

```
   {
      return denominator;
   }

   //-----------------------------------------------------------------
   //  Returns the reciprocal of this rational number.
   //-----------------------------------------------------------------
   public RationalNumber reciprocal()
   {
      return new RationalNumber(denominator, numerator);
   }

   //-----------------------------------------------------------------
   //  Adds this rational number to the one passed as a parameter.
   //  A common denominator is found by multiplying the individual
   //  denominators.
   //-----------------------------------------------------------------
   public RationalNumber add(RationalNumber op2)
   {
      int commonDenominator = denominator * op2.getDenominator();
      int numerator1 = numerator * op2.getDenominator();
      int numerator2 = op2.getNumerator() * denominator;
      int sum = numerator1 + numerator2;

      return new RationalNumber(sum, commonDenominator);
   }

   //-----------------------------------------------------------------
   //  Subtracts the rational number passed as a parameter from this
   //  rational number.
   //-----------------------------------------------------------------
   public RationalNumber subtract(RationalNumber op2)
   {
      int commonDenominator = denominator * op2.getDenominator();
      int numerator1 = numerator * op2.getDenominator();
      int numerator2 = op2.getNumerator() * denominator;
      int difference = numerator1 - numerator2;

      return new RationalNumber(difference, commonDenominator);
   }

   //-----------------------------------------------------------------
   //  Multiplies this rational number by the one passed as a
   //  parameter.
   //-----------------------------------------------------------------
```


LISTING 5.11 *continued*

```
   public RationalNumber multiply(RationalNumber op2)
   {
      int numer = numerator * op2.getNumerator();
      int denom = denominator * op2.getDenominator();

      return new RationalNumber(numer, denom);
   }

   //-----------------------------------------------------------------
   //  Divides this rational number by the one passed as a parameter
   //  by multiplying by the reciprocal of the second rational.
   //-----------------------------------------------------------------
   public RationalNumber divide(RationalNumber op2)
   {
      return multiply(op2.reciprocal());
   }

   //-----------------------------------------------------------------
   //  Determines if this rational number is equal to the one passed
   //  as a parameter. Assumes they are both reduced.
   //-----------------------------------------------------------------
   public boolean isLike(RationalNumber op2)
   {
      return (numerator == op2.getNumerator() &&
              denominator == op2.getDenominator());
   }

   //-----------------------------------------------------------------
   //  Returns this rational number as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result;

      if (numerator == 0)
         result = "0";
      else
         if (denominator == 1)
         result = numerator + "";
      else
         result = numerator + "/" + denominator;

      return result;
   }
```

LISTING 5.11 *continued*

```
//-----------------------------------------------------------------
//  Reduces this rational number by dividing both the numerator
//  and the denominator by their greatest common divisor.
//-----------------------------------------------------------------
private void reduce()
{
   if (numerator != 0)
   {
      int common = gcd(Math.abs(numerator), denominator);

      numerator = numerator / common;
      denominator = denominator / common;
   }
}

//-----------------------------------------------------------------
//  Computes and returns the greatest common divisor of the two
//  positive parameters. Uses Euclid's algorithm.
//-----------------------------------------------------------------
private int gcd(int num1, int num2)
{
   while (num1 != num2)
      if (num1 > num2)
         num1 = num1 - num2;
   else
      num2 = num2 - num1;

      return num1;
   }
}
```

The more complex an object, the more likely it is that it will need to be represented as an aggregate object. In UML, aggregation is represented by a connection between two classes, with an open diamond at the end near the class that is the aggregate. Figure 5.10 shows a UML class diagram that contains an aggregation relationship.

Note that in previous UML diagram examples, strings are not represented as separate classes with aggregation relationships, even though technically they could be. Strings are so fundamental to programming that often they are represented as though they were a primitive type in a UML diagram.

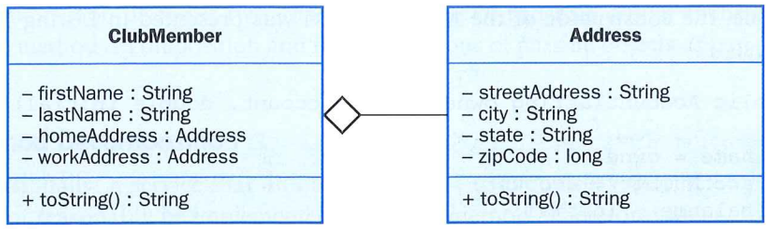

FIGURE 5.10 A UML class diagram showing an aggregation relationship

The this Reference

Before we leave the topic of relationships among classes, we should examine another special reference used in Java programs called the `this` reference. The word `this` is a reserved word in Java. It allows an object to refer to itself. As we have discussed, a nonstatic method is invoked through (or by) a particular object or class. Inside that method, the `this` reference can be used to refer to the currently executing object.

For example, in a class called `ChessPiece` there could be a method called `move`, which could contain

```
if (this.position == piece2.position)
          result =false;
```

In this situation, the `this` reference is being used to clarify which position is being referenced. The `this` reference refers to the object through which the method was invoked. So when the following line is used to invoke the method, the `this` reference refers to `bishop1`:

```
bishop1.move();
```

However, when another object is used to invoke the method, the `this` reference refers to it. Therefore, when the following invocation is used, the `this` reference in the `move` method refers to `bishop2`:

```
bishop2.move();
```

Often, the `this` reference is used to distinguish the parameters of a constructor from their corresponding instance variables with the same names. For

example, the constructor of the `Account` class was presented in Listing 5.7 as follows:

```
public Account(String owner, long account, double initial)
{
   name = owner;
   acctNumber = account;
   balance = initial;
}
```

When writing this constructor, we deliberately came up with different names for the parameters to distinguish them from the instance variables `name`, `acctNumber`, and `balance`. This distinction is arbitrary. The constructor could have been written as follows using the `this` reference:

```
public Account(String name, long acctNumber, double balance)
{
   this.name = name;
   this.acctNumber = acctNumber;
   this.balance = balance;
}
```

In this version of the constructor, the `this` reference specifically refers to the instance variables of the object. The variables on the right-hand side of the assignment statements refer to the formal parameters. This approach eliminates the need to come up with different yet equivalent names. This situation sometimes occurs in other methods, but it comes up often in constructors.

5.7 Method Design

Once you have identified classes and assigned basic responsibilities, the design of each method will determine how exactly the class will define its behaviors. Some methods are straightforward and require little thought. Others are more interesting and require careful planning.

An *algorithm* is a step-by-step process for solving a problem. A recipe is an example of an algorithm. Travel directions are another example of an algorithm. Every method implements an algorithm that determines how that method accomplishes its goals.

An algorithm is often described using *pseudocode*, which is a mixture of code statements and English phrases. Pseudocode provides enough structure to show how the code will operate, without getting bogged down in the syntactic details of a particular programming language or becoming prematurely constrained by the characteristics of particular programming constructs.

This section discusses two important aspects of program design at the method level: method decomposition and the implications of passing objects as parameters.

Method Decomposition

KEY CONCEPT
The best way to make use of a complex service provided by an object may be to decompose the method and use several private support methods to help with the task.

Occasionally, a service that an object provides is so complex that it cannot reasonably be implemented using one method. Therefore, we sometimes need to decompose a method into multiple methods to create a more understandable design. As an example, let's examine a program that translates English sentences into "Pig Latin."

Pig Latin is a made-up language in which each word of a sentence is modified, in general, by moving the initial sound of the word to the end and adding an "ay" sound. For example, the word *happy* would be written and pronounced *appyhay*, and the word *birthday* would become *irthdaybay*. Words that begin with vowels simply have a "yay" sound added on the end, turning the word *enough* into *enoughyay*. Consonant blends such as "ch" and "st" at the beginning of a word are moved to the end together before adding the "ay" sound. Therefore, the word *grapefruit* becomes *apefruitgray*.

The `PigLatin` program shown in Listing 5.12 reads one or more sentences, translating each into Pig Latin.

The workhorse behind the `PigLatin` program is the `PigLatinTranslator` class, shown in Listing 5.13 on page 215. The `PigLatinTranslator` class provides one fundamental service, a static method called `translate`, which accepts a string and translates it into Pig Latin. Note that the `PigLatinTranslator` class does not contain a constructor because none is needed.

The act of translating an entire sentence into Pig Latin is not trivial. If written in one big method, it would be very long and difficult to follow. A better solution, as implemented in the `PigLatinTranslator` class, is to decompose the `translate` method and use several private support methods to help with the task.

The `translate` method uses a `Scanner` object to separate the string into words. Recall that one role of the `Scanner` class (discussed in Chapter 3) is to separate a string into smaller elements called tokens. In this case, the tokens are separated by space characters so that we can use the default whitespace delimiters. The `PigLatin` program assumes that no punctuation is included in the input.

The `translate` method passes each word to the private support method `translateWord`. Even the job of translating one word is somewhat involved, so the `translateWord` method makes use of two other private methods, `beginsWithVowel` and `beginsWithBlend`.

The `beginsWithVowel` method returns a `boolean` value that indicates whether the word passed as a parameter begins with a vowel. Note that instead of checking each vowel separately, the code for this method declares a string that contains all the vowels and then invokes the `String` method `indexOf` to determine whether

LISTING 5.12

```
//********************************************************************
//  PigLatin.java           Java Foundations
//
//  Demonstrates the concept of method decomposition.
//********************************************************************

import java.util.Scanner;

public class PigLatin
{
   //-----------------------------------------------------------------
   //  Reads sentences and translates them into Pig Latin.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      String sentence, result, another;

      Scanner scan = new Scanner(System.in);

      do
      {
         System.out.println();
         System.out.println("Enter a sentence (no punctuation):");
         sentence = scan.nextLine();

         System.out.println();
         result = PigLatinTranslator.translate (sentence);
         System.out.println("That sentence in Pig Latin is:");
         System.out.println(result);

         System.out.println();
         System.out.print("Translate another sentence (y/n)? ");
         another = scan.nextLine();
      }
      while (another.equalsIgnoreCase("y"));
   }
}
```

OUTPUT

```
Enter a sentence (no punctuation):
Do you speak Pig Latin

That sentence in Pig Latin is:
oday ouyay eakspay igpay atinlay

Translate another sentence (y/n)? y

Enter a sentence (no punctuation):
Play it again Sam

That sentence in Pig Latin is:
ayplay ityay againyay amsay

Translate another sentence (y/n)? n
```


LISTING 5.13

```
//********************************************************************
//  PigLatinTranslator.java      Java Foundations
//
//  Represents a translator from English to Pig Latin. Demonstrates
//  method decomposition.
//********************************************************************

import java.util.Scanner;

public class PigLatinTranslator
{
   //-----------------------------------------------------------------
   //  Translates a sentence of words into Pig Latin.
   //-----------------------------------------------------------------
   public static String translate(String sentence)
   {
      String result = "";

      sentence = sentence.toLowerCase();

      Scanner scan = new Scanner(sentence);

      while (scan.hasNext())
```

LISTING 5.13 *continued*

```
      {
         result += translateWord(scan.next());
         result += " ";
      }

      return result;
   }

   //-----------------------------------------------------------------
   //  Translates one word into Pig Latin. If the word begins with a
   //  vowel, the suffix "yay" is appended to the word. Otherwise,
   //  the first letter or two are moved to the end of the word,
   //  and "ay" is appended.
   //-----------------------------------------------------------------
   private static String translateWord(String word)
   {
      String result = "";

      if (beginsWithVowel(word))
         result = word + "yay";
      else
         if (beginsWithBlend(word))
            result = word.substring(2) + word.substring(0,2) + "ay";
         else
            result = word.substring(1) + word.charAt(0) + "ay";

      return result;
   }

   //-----------------------------------------------------------------
   //  Determines if the specified word begins with a vowel.
   //-----------------------------------------------------------------
   private static boolean beginsWithVowel(String word)
   {
      String vowels = "aeiou";

      char letter = word.charAt(0);

      return (vowels.indexOf(letter) != -1);
   }

   //-----------------------------------------------------------------
   //  Determines if the specified word begins with a particular
   //  two-character consonant blend.
   //-----------------------------------------------------------------
   private static boolean beginsWithBlend(String word)
```


LISTING 5.13 *continued*

```
   {
      return ( word.startsWith("bl") || word.startsWith("sc") ||
               word.startsWith("br") || word.startsWith("sh") ||
               word.startsWith("ch") || word.startsWith("sk") ||
               word.startsWith("cl") || word.startsWith("sl") ||
               word.startsWith("cr") || word.startsWith("sn") ||
               word.startsWith("dr") || word.startsWith("sm") ||
               word.startsWith("dw") || word.startsWith("sp") ||
               word.startsWith("fl") || word.startsWith("sq") ||
               word.startsWith("fr") || word.startsWith("st") ||
               word.startsWith("gl") || word.startsWith("sw") ||
               word.startsWith("gr") || word.startsWith("th") ||
               word.startsWith("kl") || word.startsWith("tr") ||
               word.startsWith("ph") || word.startsWith("tw") ||
               word.startsWith("pl") || word.startsWith("wh") ||
               word.startsWith("pr") || word.startsWith("wr") );
   }
}
```

the first character of the word is in the vowel string. If the specified character cannot be found, the `indexOf` method returns a value of `-1`.

The `beginsWithBlend` method also returns a `boolean` value. The body of the method contains only a `return` statement with one large expression that makes several calls to the `startsWith` method of the `String` class. If any of these calls returns true, then the `beginsWithBlend` method returns true as well.

Note that the `translateWord`, `beginsWithVowel`, and `beginsWithBlend` methods are all declared with private visibility. They are not intended to provide services directly to clients outside the class. Instead, they exist to help the `translate` method, which is the only true service method in this class, to do its job. Declaring them with private visibility means that they cannot be invoked from outside this class. For instance, if the `main` method of the `PigLatin` class attempted to invoke the `translateWord` method, the compiler would issue an error message.

Figure 5.11 shows a UML class diagram for the `PigLatin` program. Note the notation showing the visibility of various methods.

Whenever a method becomes large or complex, we should consider decomposing it into multiple methods to create a more understandable class design. First, however, we must consider how other classes and objects can be defined to create better overall system design. In an object-oriented design, method decomposition must be subordinate to object decomposition.

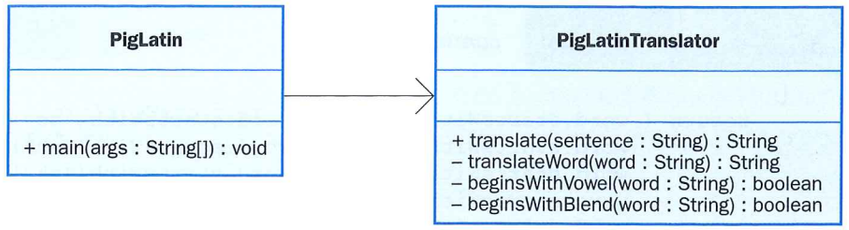

FIGURE 5.11 A UML class diagram for the `PigLatin` program

Method Parameters Revisited

Another important issue related to method design involves the way parameters are passed into a method. In Java, all parameters are passed *by value*. That is, the current value of the actual parameter (in the invocation) is copied into the formal parameter in the method header. We mentioned this issue previously in this chapter; let's examine it now in more detail.

Essentially, parameter passing is like an assignment statement, assigning to the formal parameter a copy of the value stored in the actual parameter. This issue must be considered when making changes to a formal parameter inside a method. The formal parameter is a separate copy of the value that is passed in, so any changes made to it have no effect on the actual parameter. After control returns to the calling method, the actual parameter will have the same value it had before the method was called.

However, when we pass an object to a method, we are actually passing a reference to that object. The value that gets copied is the address of the object. Therefore, the formal parameter and the actual parameter become aliases of each other. If we change the state of the object through the formal parameter reference inside the method, we are changing the object referenced by the actual parameter, because they refer to the same object. On the other hand, if we change the formal parameter reference itself (to make it point to a new object, for instance), we have not changed the fact that the actual parameter still refers to the original object.

KEY CONCEPT
When an object is passed to a method, the actual and formal parameters become aliases.

The program in Listing 5.14 illustrates the nuances of parameter passing. Carefully trace the processing of this program, and note the values that are output. The `ParameterTester` class contains a `main` method that calls the `changeValues` method in a `ParameterModifier` object. Two of the parameters to `changeValues` are `Num` objects, each of which simply stores an integer value. The other parameter is a primitive integer value.

Listing 5.15 on page 220 shows the `ParameterModifier` class, and Listing 5.16 on page 221 shows the `Num` class. Inside the `changeValues` method, a modification is made to each of the three formal parameters: The integer parameter is set to a different value, the value stored in the first `Num` parameter is changed

LISTING 5.14

```
//********************************************************************
//  ParameterTester.java       Java Foundations
//
//  Demonstrates the effects of passing various types of parameters.
//********************************************************************

public class ParameterTester
{
   //-----------------------------------------------------------------
   //  Sets up three variables (one primitive and two objects) to
   //  serve as actual parameters to the changeValues method. Prints
   //  their values before and after calling the method.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      ParameterModifier modifier = new ParameterModifier();

      int a1 = 111;
      Num a2 = new Num(222);
      Num a3 = new Num(333);

      System.out.println("Before calling changeValues:");
      System.out.println("a1\ta2\ta3");
      System.out.println(a1 + "\t" + a2 + "\t" + a3 + "\n");

      modifier.changeValues(a1, a2, a3);

      System.out.println("After calling changeValues:");
      System.out.println("a1\ta2\ta3");
      System.out.println(a1 + "\t" + a2 + "\t" + a3 + "\n");
   }
}
```

OUTPUT

```
Before calling changeValues:
a1          a2          a3
111         222         333

Before changing the values:
f1          f2          f3
111         222         333
```

LISTING 5.14 *continued*

```
After changing the values:
f1          f2          f3
999         888         777

After calling changeValues:
a1          a2          a3
111         888         333
```

LISTING 5.15

```
//********************************************************************
//  ParameterModifier.java       Java Foundations
//
//  Demonstrates the effects of changing parameter values.
//********************************************************************

public class ParameterModifier
{
   //-----------------------------------------------------------------
   //  Modifies the parameters, printing their values before and
   //  after making the changes.
   //-----------------------------------------------------------------
   public void changeValues(int f1, Num f2, Num f3)
   {
      System.out.println("Before changing the values:");
      System.out.println("f1\tf2\tf3");
      System.out.println(f1 + "\t" + f2 + "\t" + f3 + "\n");

      f1 = 999;
      f2.setValue(888);
      f3 = new Num(777);

      System.out.println("After changing the values:");
      System.out.println("f1\tf2\tf3");
      System.out.println(f1 + "\t" + f2 + "\t" + f3 + "\n");
   }
}
```


LISTING 5.16

```
//********************************************************************
//  Num.java       Java Foundations
//
//  Represents a single integer as an object.
//********************************************************************

public class Num
{
   private int value;

   //-----------------------------------------------------------------
   //  Sets up the new Num object, storing an initial value.
   //-----------------------------------------------------------------
   public Num(int update)
   {
      value = update;
   }

   //-----------------------------------------------------------------
   //  Sets the stored value to the newly specified value.
   //-----------------------------------------------------------------
   public void setValue(int update)
   {
      value = update;
   }

   //-----------------------------------------------------------------
   //  Returns the stored integer value as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      return value + "";
   }
}
```

using its `setValue` method, and a new `Num` object is created and assigned to the second `Num` parameter. These changes are reflected in the output printed at the end of the `changeValues` method.

However, note the final values that are printed after returning from the method. The primitive integer was not changed from its original value, because the change was made to a copy inside the method. Likewise, the last parameter

still refers to its original object with its original value. This is because the new `Num` object created in the method was referred to only by the formal parameter. When the method returned, that formal parameter was destroyed, and the `Num` object it referred to was marked for garbage collection. The only change that is "permanent" is the change made to the state of the second parameter. Figure 5.12 shows the step-by-step processing of this program.

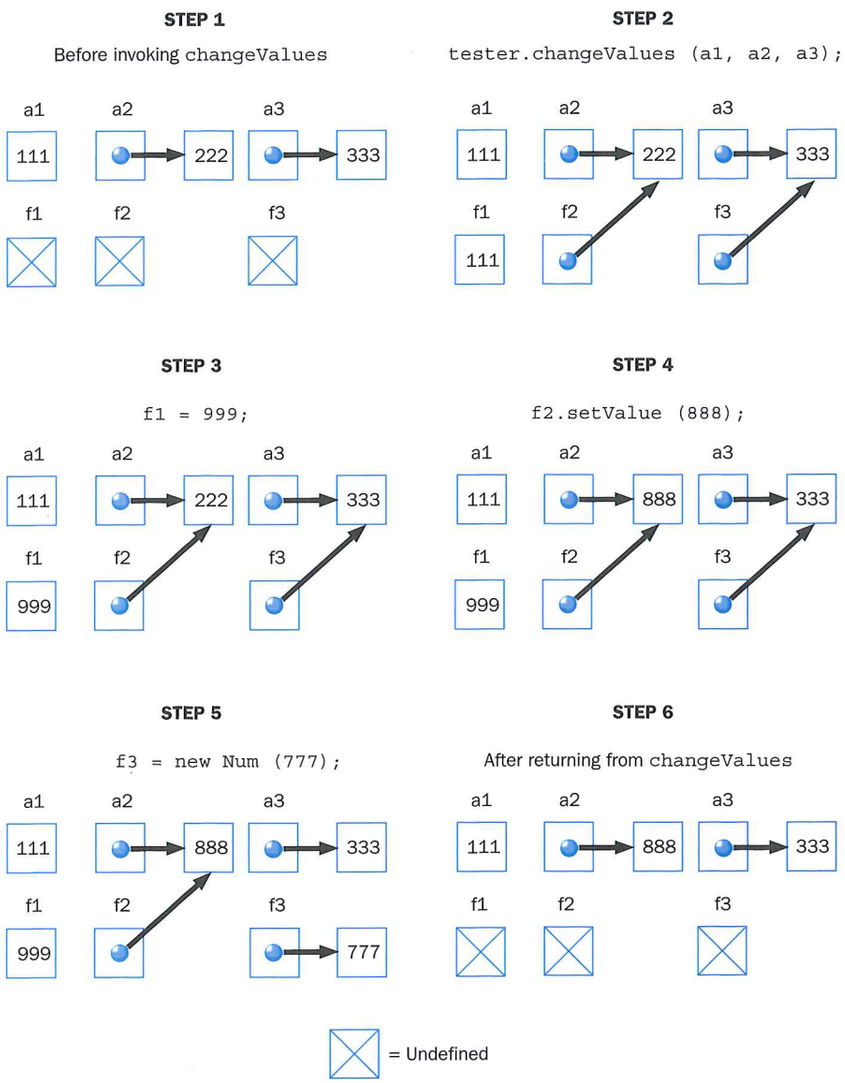

FIGURE 5.12 Tracing the parameters in the `ParameterTesting` program

5.8 Method Overloading

As we've discussed, when a method is invoked, control transfers to the code that defines the method. After the method has been executed, control returns to the location of the call, and processing continues.

Often the method name is sufficient to indicate which method is being called by a specific invocation. But in Java, as in other object-oriented languages, you can use the same method name with different parameter lists for multiple methods. This technique is called *method overloading*. It is useful when you need to perform similar methods on different types of data.

The compiler must still be able to associate each invocation with a specific method declaration. If the method name for two or more methods is the same, additional information is used to uniquely identify the version that is being invoked. In Java, a method name can be used for multiple methods as long as the number of parameters, the types of those parameters, and/or the order of the types of parameters is distinct.

KEY CONCEPT
The versions of an overloaded method are distinguished by the number, type, and order of their parameters.

For example, we could declare a method called `sum` as follows:

```
public int sum(int num1, int num2)
{
   return num1 + num2;
}
```

Then we could declare another method called `sum`, within the same class, as follows:

```
public int sum(int num1, int num2, int num3)
{
   return num1 + num2 + num3;
}
```

Now, when an invocation is made, the compiler looks at the number of parameters to determine which version of the `sum` method to call. For instance, the following invocation will call the second version of the `sum` method:

```
sum(25, 69, 13);
```

A method's name, along with the number, type, and order of its parameters, is called the method's *signature*. The compiler uses the complete method signature to *bind* a method invocation to the appropriate definition.

The compiler must be able to examine a method invocation to determine which specific method is being invoked. If you attempt to specify two method names with the same signature, the compiler will issue an appropriate error message and will not create an executable program. There can be no ambiguity.

Note that the return type of a method is not part of the method signature. That is, two overloaded methods cannot differ only by their return type. This is because the value returned by a method can be ignored by the invocation. The compiler would not be able to tell which version of an overloaded method was being referenced in such situations.

The `println` method is an example of a method that is overloaded several times, each accepting a single type. Here is a partial list of its various signatures:

- `println(String s)`
- `println(int i)`
- `println(double d)`
- `println(char c)`
- `println(boolean b)`

The following two lines of code actually invoke different methods that have the same name:

```
System.out.println("Number of students: ");
System.out.println(count);
```

The first line invokes the version of `println` that accepts a string. The second line, assuming that `count` is an integer variable, invokes the version of `println` that accepts an integer.

We often use a `println` statement that prints several distinct types, such as

```
System.out.println("Number of students: " + count);
```

Remember, in this case the plus sign is the string concatenation operator. First, the value in the variable `count` is converted to a string representation, then the two strings are concatenated into one longer string, and finally the definition of `println` that accepts a single string is invoked.

Constructors can be overloaded, and they often are. By providing multiple versions of a constructor, we provide multiple ways to set up an object.

5.9 Testing

As our programs become larger and more complex, it becomes more difficult to ensure their accuracy and reliability. Accordingly, before we continue with further programming details, let's explore the processes involved in testing a program.

The term *testing* can be applied in many ways to software development. Testing certainly includes its traditional definition: the act of running a completed program with various inputs to discover problems. But it also includes any evaluation that is performed by human or machine to assess the quality of the evolving system. These evaluations should occur long before a single line of code is written.

The goal of testing is to find errors. By finding errors and fixing them, we improve the quality of our program. It's likely that later on, someone else will find any errors that remain hidden during development. The earlier the errors are found, the easier and cheaper they are to fix. Taking the time to uncover problems as early as possible is almost always worth the effort.

Running a program with specific input and producing the correct results establishes only that the program works for that particular input. As more and more test cases execute without revealing errors, our confidence in the program rises, but we can never really be sure that all errors have been eliminated. There could always be another error still undiscovered. Because of that, it is important to thoroughly test a program in as many ways as possible and with well-designed test cases.

KEY CONCEPT
Testing a program can never guarantee the absence of errors.

It is possible to prove that a program is correct, but that technique is enormously complex for large systems, and errors can be made in the proof itself. Therefore, we generally rely on testing to determine the quality of a program.

After determining that an error exists, we determine the cause of the error and fix it. After a problem is fixed, we should run the previously administered tests again to make sure that while fixing the problem, we didn't create another. This technique is called *regression testing*.

Reviews

One technique used to evaluate design or code is called a *review*, which is a meeting in which several people carefully examine a design document or section of code. Presenting our design or code to others causes us to think more carefully about it and permits others to share their suggestions with us. The participants discuss its merits and problems and create a list of issues that must be addressed. The goal of a review is to identify problems, not to solve them, which usually takes much more time.

A design review should determine whether the requirements are addressed. It should also assess the way the system is decomposed into classes and objects. A code review should determine how faithfully the design satisfies the requirements and how faithfully the implementation represents the design. It should identify any specific problems that would cause the design or the implementation to fail in its responsibilities.

Sometimes a review is called a *walkthrough*, because its goal is to step carefully through a document and evaluate each section.

Defect Testing

Because the goal of testing is to find errors, it is often referred to as *defect testing*. With that goal in mind, a good test is one that uncovers any deficiencies in a program. This might seem strange, because we ultimately don't want to have problems in our system. But keep in mind that errors almost certainly exist. Our testing efforts should make every attempt to find them. We want to increase the reliability of our program by finding and fixing the errors that exist, rather than letting users discover them.

KEY CONCEPT
A good test is one that uncovers an error.

A *test case* is a set of inputs, user actions, or other initial conditions, and the expected output. A test case should be appropriately documented so that it can be repeated later as needed. Developers often create a complete *test suite*, which is a set of test cases that covers various aspects of the system.

KEY CONCEPT
It is not feasible to exhaustively test a program for all possible input and user actions.

Because programs operate on a large number of possible inputs, it is not feasible to create test cases for all possible input or user actions. Nor is it usually necessary to test every single situation. Two specific test cases may be so similar that they actually do not test unique aspects of the program. To perform both such tests would be a waste of effort. We'd rather execute a test case that stresses the program in some new way. Therefore, we want to choose our test cases carefully. To that end, let's examine two approaches to defect testing: black-box testing and white-box testing.

As the name implies, *black-box testing* treats the thing being tested as a black box. In black-box testing, test cases are developed without regard to the internal workings. Black-box tests are based on inputs and outputs. An entire program can be tested using a black-box technique, in which case the inputs are the user-provided information and user actions such as button pushes. A test case is successful only if the input produces the expected output. A single class can also be tested using a black-box technique, which focuses on the system interface of the class (its public methods). Certain parameters are passed in, producing certain results. Black-box test cases are often derived directly from the requirements of the system or from the stated purpose of a method.

The input data for a black-box test case are often selected by defining *equivalence categories*. An equivalence category is a collection of inputs that are expected to produce similar outputs. Generally, if a method will work for one value in the equivalence category, we have every reason to believe it will work for the others. For example, the input to a method that computes the square root of an integer can be divided into two equivalence categories: nonnegative integers and negative integers. If it works appropriately for one nonnegative value, it is likely to work for all nonnegative values. Likewise, if it works appropriately for one negative value, it is likely to work for all negative values.

Equivalence categories have defined boundaries. Because all values of an equivalence category essentially test the same features of a program, only one test case inside the equivalence boundary is needed. However, because programming often produces "off by one" errors, the values on and around the boundary should be tested exhaustively. For an integer boundary, a good test suite would include at least the exact value of the boundary, the boundary minus 1, and the boundary plus 1. Test cases that use these cases, plus at least one from within the general field of the category, should be defined.

Let's look at an example. Consider a method whose purpose is to validate that a particular integer value is in the range 0 to 99, inclusive. There are three equivalence categories in this case: values below 0, values in the range of 0 to 99, and values above 99. Black-box testing dictates that we use test values that surround and fall on the boundaries, as well as some general values from the equivalence categories. Therefore, a set of black-box test cases for this situation might be –500, –1, 0, 1, 50, 98, 99, 100, and 500.

White-box testing, also known as *glass-box testing*, exercises the internal structure and implementation of a method. A white-box test case is based on the logic of the code. The goal is to ensure that every path through a program is executed at least once. A white-box test maps the possible paths through the code and ensures that the test cases cause every path to be executed. This type of testing is often called *statement coverage*.

Paths through code are controlled by various control flow statements that use conditional expressions, such as `if` statements. In order to have every path through the program executed at least once, the input data values for the test cases need to control the values for the conditional expressions. The input data of one or more test cases should cause the condition of an `if` statement to evaluate to `true` in at least one case and to `false` in at least one case. Covering both true and false values in an `if` statement guarantees that both paths through the `if` statement will be executed. Similar situations can be created for loops and other constructs.

In both black-box and white-box testing, the expected output for each test should be established before running the test. It's too easy to be persuaded that the results of a test are appropriate if you haven't first carefully determined what the results should be.

Unit Testing

Another type of testing is known as *unit testing*. This approach creates a test case for each module of code (method) that has been authored. The goal of unit testing is to ensure correctness of the methods (units), one method at a time. Generally, we collect our unit tests together, execute each test, and observe all of

the results. We can also use these tests repeatedly as the source code changes, to observe the effect of our code changes on the results of the test (regression testing, discussed above).

Integration Testing

During *integration testing*, modules that were individually tested during unit testing are now tested as a collection. This form of testing looks at the larger picture and determines whether there are bugs present when modules are brought together and integrated to work together. As with unit testing, we can use regression testing as the software changes to determine how our results may have changed as integration occurred and as problems in the modules or integration approaches were modified. Typically, the goal of integration testing is to examine the correctness of large components of a system.

System Testing

System testing seeks to test the entire software system and how its implementation adheres to its requirements. You may be familiar with public alpha or beta testing of applications or operating systems. Alpha and beta tests are system tests applied before the formal release and availability of a software product. Software development companies partake in these types of public tests to increase the number of users testing a product or to expand the hardware base that the product is tested on.

Test-Driven Development

Ideally, developers should be writing test cases concurrently with the development of the source code that their applications use. In fact, many developers have adopted the practice of writing their test cases first and then implementing only enough source code for the test case to pass. This notion is known professionally as *test-driven development*.

KEY CONCEPT
In test-driven development, test cases are developed for code before the code is written.

The test-driven approach requires that developers periodically (during development and implementation) test their code using the implemented test cases. If you were to look at the test-driven approach as a sequence of steps, you would generally find the following activities:

1. Create a test case that tests a specific method that has yet to be completed.
2. Execute all of the test cases present, and verify that all test cases pass except for the most recently implemented test case.
3. Develop the method that the test case targets so that the test case will pass without errors.
4. Re-execute all of the test cases, and verify that every test case passes, including the most recently implemented test case.
5. Clean up the code to eliminate any redundant portions introduced by the development of the most recent method. This step is known as *refactoring* the code.
6. Repeat the process starting with Step 1.

The test-driven approach is becoming increasingly popular in professional settings. Without a doubt, it requires an adjustment for many developers to stop and write test cases before writing methods that provide functionality to systems under development.

5.10 Debugging

Finally, we come to one of the most important concepts that you will ever master as a programmer—the art of *debugging* your programs. Debugging is the act of locating and correcting run-time and logic errors in your programs. We can locate errors in our programs in a number of different ways. You may notice a run-time error (the program terminating abnormally) when you execute your program and certain situations arise that you did not consider or design for. Also, you may notice a logic error in your program as it executes when your anticipated results do not match the actual results you obtain.

KEY CONCEPT
Debugging is the act of locating and correcting run-time and logic errors in your programs.

Ideally, through rigorous testing, we hope to discover all possible errors in our programs. Typically, however, a few errors slip through into the final program. Once you recognize that your program contains an error, you will want to locate the portion of code from which the error is arising. For example, if you determine that your program is terminating abnormally because of a divide-by-zero problem, you'll probably want to locate the exact line where it is happening. You may also wish to observe the values of the variables involved in the division. The same may be true if you have a logic error (rather than a divide-by-zero problem) in that division operation.

Regardless of your motivation, it is often very helpful to obtain detailed information about the values of variables, states of objects, and other inner workings of your program. This is where a debugger comes in. A *debugger* is a software application that allows us to observe these inner workings as the program executes. However, before we discuss the debugger, we should first talk about simple debugging.

KEY CONCEPT
A debugger is a software program that permits developers to observe the execution of a program.

Simple Debugging with `print` Statements

One of the most simplistic approaches to debugging involves the use of printing. That is, scattered throughout a program can be `print` and `println` statements that output various information either to the screen or to an output file. Generally, this type of approach will provide information on the value or state of a specific variable or object. Periodic printing of an object's string representation is considered a useful approach to observing the state of an object over time.

Other types of useful information can be printed as well. For example, sometimes we wish to know "how far our program got before it died." Programmers facing this challenge often print a series of "It got here." statements to output to monitor the exact path of execution of the program.

Consider the case of calling a method. It may be useful for us to print the value of each parameter after the method starts to observe how the method was called. This is particularly helpful when we are debugging recursive methods, discussed in the next chapter. We can also print the value of a variable prior to its being returned as the method ends.

Debugging Concepts

Debugging through printing can take us only so far. Most of the time, this style of debugging can be effectively used to identify what is happening during execution, or the value of a variable at a certain point in the program. However, a more powerful approach is to use a debugger in which our program will execute. The debugger can be used to control our program's execution and provide additional functionality to the developer that we simply can't get with simple debugging through `print` statements.

A debugger allows us to do the following:

- Set one or more *breakpoints* in our program. In the debugger, we can examine the source code and set special flags or triggers on one or more lines of code. When execution of the program comes across a statement that has been flagged, execution stops.
- Print the value of a variable or object. Once we have reached a breakpoint and execution has stopped, the debugger allows us to display the value of a variable or examine the state of an object. Generally, these types of displays are to the screen and only within the confines of the debugger application.
- Step into or over a method. If we set a breakpoint at a statement that is a call to a method, when execution reaches this breakpoint and the program stops, the developer can choose to enter into the method and continue debugging or to step over the method, bypassing the display of the execution of the statements contained in the method. In stepping over the method, we should note that the method is still executed, but we have chosen not to delve into the method. Consider the call to the printing of a string of output to the screen. We probably don't need to step into the `println` method of the `System.out` object. It's likely to have been fully debugged already (and we can't change its behavior anyway).
- Execute the next single statement. After reaching a breakpoint, the developer can choose to execute the next single statement (also known as a *step*). By executing a single step, we can literally control the execution of our program one statement at a time. Developers often perform stepping to be sure they understand the flow of execution and to give themselves an opportunity to display the value of a variable following each step, if desired.
- Continue execution. Once a program has stopped due to a breakpoint, or is waiting for the developers to decide whether they will step into, step over, or single step, the developers can also continue execution. Continuing execution will result in the program running each statement without pausing until the program ends, it encounters another breakpoint, or a run-time error occurs.

Debuggers also offer a pile of additional features to assist in the debugging task. However, for the purposes of this discussion, we can limit ourselves to the set of activities listed above. Any debugger that is worth using has these operations, at the very least.

Summary of Key Concepts

- The nouns in a problem description may indicate some of the classes and objects needed in a program.
- The heart of object-oriented programming is defining classes that represent objects with well-defined state and behavior.
- The scope of a variable, which determines where it can be referenced, depends on where it is declared.
- A UML class diagram helps us visualize the contents of, and the relationships among, the classes of a program.
- An object should be encapsulated in order to safeguard its data from inappropriate access.
- Instance variables should be declared with private visibility to promote encapsulation.
- Most objects contain accessor and mutator methods to allow the client to manage data in a controlled manner.
- The way a class represents an object's state should be independent of how that object is used.
- The value returned from a method must be consistent with the return type specified in the method header.
- When a method is called, the actual parameters are copied into the formal parameters.
- A variable declared in a method is local to that method and cannot be used outside of it.
- A constructor cannot have any return type, even `void`.
- A static variable is shared among all instances of a class.
- An aggregate object is composed of other objects, forming a has-a relationship.
- A complex service provided by an object can be decomposed to make use of private support methods.
- When an object is passed to a method, the actual and formal parameters become aliases.
- The versions of an overloaded method are distinguished by the number, type, and order of their parameters.
- Testing a program can never guarantee the absence of errors.

- A good test is one that uncovers an error.
- It is not feasible to exhaustively test a program for all possible input and user actions.
- In test-driven development, test cases are developed for code before the code is written.
- Debugging is the act of locating and correcting run-time and logic errors in your programs.
- A debugger is a software program that permits developers to observe the execution of a program.

Summary of Terms

accessor method A method that provides access to the attributes of an object but does not modify it.

actual parameter A value that is passed into a method when it is invoked. Also called an argument.

aggregation A relationship among object in which one object is made up of other objects.

behavior The set of operations defined by the public methods of an object.

black-box testing Testing a program with attention to the inputs and outputs of the code.

client A part of a software system that uses an object.

debugging The act of locating and correcting run-time and logical errors in a program.

defect testing Executing a program with specific inputs in order to find errors.

encapsulation The characteristic of an object that keeps its data protected from external modification.

formal parameter A parameter name in the header of a method definition.

instance data Data defined at the class level and created in memory for every object.

integration testing Testing modules as they are incorporated with each other, focusing on the communication between them.

interface The set of public methods that define the operations that an object makes available to other objects.

local data Data that are declared within a method.

method overloading The ability to declare multiple methods with the same name in a class as long as the method signatures are distinct.

method signature The method's name, along with the number, type, and order of the method's parameters.

modfier A Java reserved word that is used to specify particular characteristics of a variable, method, or class.

mutator method A method that changes the attributes of an object.

private visibility Restricting access to an object member to the methods within that object.

public visiblity The ability to be referenced from outside an object.

`return` statement A statement that causes a method to terminate and possibly return a value to the calling method.

review A meeting in which several people examine a design document or section of code in order to discover problems.

scope The area of a program in which a variable can be referenced.

service method A public method that provides a service to the clients of an object.

state The current values of the attributes (instance variables) of an object.

static method A method that is invoked through the class name and cannot refer to instance data.

static variable A variable that is shared among all instances of a class. Also called a class variable.

support methods A method with private visibility, used to support another method in its task.

system testing Testing an entire software system for its overall functionality.

testing The process of evaluating a program to discover defects.

test suite A set of tests that covers various aspects of a software system and can be repeated when needed.

unit testing Creating specific tests for small units of code (usually a method).

Unified Modeling Language (UML) A popular notation for representing designs of an object-oriented program.

visibility modifier One of the three modifiers (public, private, and protected) that determine what other parts of a software system can access a variable or method.

white-box testing Testing a program with attention to the logic of the code.

Self-Review Questions

SR 5.1 What is an attribute?

SR 5.2 What is an operation?

SR 5.3 What is the difference between an object and a class?

SR 5.4 What is the scope of a variable?

SR 5.5 What are UML diagrams designed to do?

SR 5.6 Objects should be self-governing. Explain.

SR 5.7 What is a modifier?

SR 5.8 Why might a constant be given public visibility?

SR 5.9 Describe each of the following:

a. public method

b. private method

c. public variable

d. private variable

SR 5.10 What is the interface to an object?

SR 5.11 Why is a method invoked through (or on) a particular object? What is the exception to that rule?

SR 5.12 What does it mean for a method to return a value?

SR 5.13 What does the `return` statement do?

SR 5.14 Is a `return` statement required?

SR 5.15 Explain the difference between an actual parameter and a formal parameter.

SR 5.16 What are constructors used for? How are they defined?

SR 5.17 What is the difference between a static variable and an instance variable?

SR 5.18 What kinds of variables can the `main` method of any program reference? Why?

SR 5.19 Describe a dependency relationship between two classes.

SR 5.20 How are overloaded methods distinguished from each other?

SR 5.21 What is method decomposition?

SR 5.22 Explain how a class can have an association with itself.

SR 5.23 What is an aggregate object?

SR 5.24 What does the `this` reference refer to?

SR 5.25 How are objects passed as parameters?

SR 5.26 What is a defect test?

SR 5.27 What is a debugger?

Exercises

EX 5.1 For each of the following pairs, indicate which member of the pair represents a class and which represents an object of that class.

a. Superhero, Superman

b. Justin, Person

c. Rover, Pet

d. Magazine, *Time*

e. Christmas, Holiday

EX 5.2 List some attributes and operations that might be defined for a class called `PictureFrame` that represents a picture frame.

EX 5.3 List some attributes and operations that might be defined for a class called `Meeting` that represents a business meeting.

EX 5.4 List some attributes and operations that might be defined for a class called `Course` that represents a college course (not a particular offering of a course, just the course in general).

EX 5.5 Rewrite the `for` loop body from the `SnakeEyes` program so that the variables `num1` and `num2` are not used.

EX 5.6 Write a method called `lyrics` that prints the lyrics of a song when invoked. The method should accept no parameters and return no value.

EX 5.7 Write a method called `cube` that accepts one integer parameter and returns that value raised to the third power.

EX 5.8 Write a method called `random100` that returns a random integer in the range of 1 to 100 (inclusive).

EX 5.9 Write a method called `randomInRange` that accepts two integer parameters representing a range. The method should return a random integer in the specified range (inclusive). Assume that the first parameter is greater than the second.

EX 5.10 Write a method called `powersOfTwo` that prints the first 10 powers of 2 (starting with 2). The method takes no parameters and doesn't return anything.

EX 5.11 Write a method called `alarm` that prints the string "`Alarm!`" multiple times on separate lines. The method should accept an integer parameter that specifies how many times the string is printed. Print an error message if the parameter is less than 1.

EX 5.12 Write a method called `sum100` that returns the sum of the integers from 1 to 100, inclusive.

EX 5.13 Write a method called `maxOfTwo` that accepts two integer parameters and returns the larger of the two.

EX 5.14 Write a method called `sumRange` that accepts two integer parameters that represent a range. Issue an error message and return zero if the second parameter is less than the first. Otherwise, the method should return the sum of the integers in that range (inclusive).

EX 5.15 Write a method called `larger` that accepts two floating point parameters (of type `double`) and returns true if the first parameter is greater than the second, and returns false otherwise.

EX 5.16 Write a method called `countA` that accepts a `String` parameter and returns the number of times the character `'A'` is found in the string.

EX 5.17 Write a method called `evenlyDivisible` that accepts two integer parameters and returns true if the first parameter is evenly divisible by the second, or vice versa, and returns false otherwise. Return false if either parameter is zero.

EX 5.18 Write a method called `isAlpha` that accepts a character parameter and returns true if that character is either an uppercase or a lowercase alphabetic letter.

EX 5.19 Write a method called `floatEquals` that accepts three floating point values as parameters. The method should return true if the first two parameters are equal within the tolerance of the third parameter.

EX 5.20 Write a method called `reverse` that accepts a `String` parameter and returns a string that contains the characters of the parameter in reverse order. *Note:* There is a method in the `String` class that performs this operation, but for the sake of this exercise, you are expected to write your own.

EX 5.21 Write a method called `isIsosceles` that accepts three integer parameters that represent the lengths of the sides of a triangle. The method returns true if the triangle is isosceles but not equilateral (meaning that exactly two of the sides have the same length), and returns false otherwise.

EX 5.22 Write a method called `average` that accepts two integer parameters and returns their average as a floating point value.

EX 5.23 Overload the `average` method of Exercise 5.22 such that if three integers are provided as parameters, the method returns the average of all three.

EX 5.24 Overload the `average` method of Exercise 5.22 to accept four integer parameters and return their average.

EX 5.25 Write a method called `multiConcat` that takes a `String` and an integer as parameters. Return a `String` that consists of the string parameter concatenated with itself `count` times, where `count` is the integer parameter. For example, if the parameter values are `"hi"` and `4`, the return value is `"hihihihi"`. Return the original string if the integer parameter is less than 2.

EX 5.26 Overload the `multiConcat` method from Exercise 5.25 such that if the integer parameter is not provided, the method returns the string concatenated with itself. For example, if the parameter is `"test"`, the return value is `"testtest"`.

EX 5.27 Discuss the manner in which Java passes parameters to a method. Is this technique consistent between primitive types and objects? Explain.

EX 5.28 Explain why a static method cannot refer to an instance variable.

EX 5.29 Can a class implement two interfaces that contain the same method signature? Explain.

EX 5.30 Draw a UML class diagram for the `CountFlips` program.

EX 5.31 Draw a UML class diagram for the `FlipRace` program.

EX 5.32 Draw a UML class diagram for the `Transactions` program.

Programming Projects

PP 5.1 Revise the `Coin` class such that its state is represented internally using a `boolean` variable. Test the new versions of the class as part of the `CountFlips` and `FlipRace` programs.

PP 5.2 Repeat Programming Project 5.1, representing the state of the coin using a character string.

PP 5.3 Repeat Programming Project 5.1, representing the state of the coin using an enumerated type.

PP 5.4 Design and implement a class called `Sphere` that contains instance data that represent the sphere's diameter. Define the `Sphere` constructor to accept and initialize the diameter, and include getter and setter methods for the diameter. Include methods that calculate and return the volume and surface area of the sphere (see Programming Project 3.5 for the formulas). Include a `toString` method that returns a one-line description of the sphere. Create a driver class called `MultiSphere`, whose `main` method instantiates and updates several `Sphere` objects.

PP 5.5 Design and implement a class called `Dog` that contains instance data that represent the dog's name and age. Define the `Dog` constructor to accept and initialize instance data. Include getter and setter methods for the name and age. Include a method to compute and return the age of the dog in "person years" (seven times the dog's age). Include a `toString` method that returns a one-line description of the dog. Create a driver class called `Kennel`, whose `main` method instantiates and updates several `Dog` objects.

PP 5.6 Design and implement a class called `Box` that contains instance data that represent the height, width, and depth of the box. Also include a `boolean` variable called `full` as instance data that represent whether the box is full or not. Define the `Box` constructor to accept and initialize the height, width, and depth of the box. Each newly created `Box` is empty (the constructor should initialize `full` to false). Include getter and setter methods for all instance data. Include a `toString` method that returns a one-line description of the box. Create a driver class called `BoxTest`, whose `main` method instantiates and updates several `Box` objects.

PP 5.7 Design and implement a class called `Book` that contains instance data for the title, author, publisher, and copyright date. Define the `Book` constructor to accept and initialize these data. Include setter and getter methods for all instance data. Include a `toString` method that returns a nicely formatted, multiline description of the book. Create a driver class called `Bookshelf`, whose `main` method instantiates and updates several `Book` objects.

PP 5.8 Design and implement a class called `Flight` that represents an airline flight. It should contain instance data that represent the

airline name, the flight number, and the flight's origin and destination cities. Define the `Flight` constructor to accept and initialize all instance data. Include getter and setter methods for all instance data. Include a `toString` method that returns a one-line description of the flight. Create a driver class called `FlightTest`, whose `main` method instantiates and updates several `Flight` objects.

PP 5.9 Design and implement a class called `Bulb` that represents a light bulb that can be turned on and off. Create a driver class called `Lights`, whose `main` method instantiates and turns on some `Bulb` objects.

PP 5.10 Using the `Die` class defined in this chapter, design and implement a class called `PairOfDice`, composed of two `Die` objects. Include methods to set and get the individual die values, a method to roll the dice, and a method that returns the current sum of the two die values. Rewrite the `SnakeEyes` program using a `PairOfDice` object.

PP 5.11 Using the `PairOfDice` class from Programming Project 5.10, design and implement a class to play a game called Pig. In this game, the user competes against the computer. On each turn, the current player rolls a pair of dice and accumulates points. The goal is to reach 100 points before your opponent does. If, on any turn, the player rolls a 1, all points accumulated for that round are forfeited, and control of the dice moves to the other player. If the player rolls two 1's in one turn, the player loses all points accumulated thus far in the game and loses control of the dice. The player may voluntarily turn over the dice after each roll. Therefore, the player must decide either to roll again (be a pig) and risk losing points or to relinquish control of the dice, possibly allowing the other player to win. Implement the computer player such that it always relinquishes the dice after accumulating 20 or more points in any given round.

PP 5.12 Modify the `Account` class from this chapter so that it also permits an account to be opened with just a name and an account number, assuming an initial balance of zero. Modify the `main` method of the `Transactions` class to demonstrate this new capability.

PP 5.13 Design and implement a class called `Card` that represents a standard playing card. Each card has a suit and a face value. Create a program that deals five random cards.

Answers to Self-Review Questions

SRA 5.1 An attribute is a data value that is stored in an object and defines a particular characteristic of that object. For example, one attribute of a `Student` object might be that student's current grade point average. Collectively, the values of an object's attributes determine that object's current state.

SRA 5.2 An operation is a function that can be done to or done by an object. For example, one operation of a `Student` object might be to compute that student's current grade point average. Collectively, an object's operations are referred to as the object's behaviors.

SRA 5.3 A class is the blueprint of an object. It defines the variables and methods that will be a part of every object that is instantiated from it. But a class reserves no memory space for variables. Each object has its own data space and therefore its own state.

SRA 5.4 The scope of a variable is the area within a program in which the variable can be referenced. An instance variable, declared at the class level, can be referenced in any method of the class. Local variables (including the formal parameters) declared within a particular method can be referenced only in that method.

SRA 5.5 A UML diagram helps us visualize the classes used in a program as well as the relationships among them. UML diagrams are tools that help us capture the design of a program prior to writing it.

SRA 5.6 A self-governing object is one that controls the values of its own data. An encapsulated object, which doesn't allow an external client to reach in and change its data, is self-governing.

SRA 5.7 A modifier is a Java reserved word that can be used in the definition of a variable or method and that specifically defines certain characteristics of its use. For example, if a variable is declared with the modifier `private`, the variable cannot be directly accessed outside the object in which it is defined.

SRA 5.8 A constant might be declared with public visibility because that would not violate encapsulation. Because the value of a constant cannot be changed, it is not generally a problem for another object to access it directly.

SRA 5.9 The modifiers affect the methods and variables in the following ways:

a. A public method is called a service method for an object because it defines a service that the object provides.

b. A private method is called a support method because it cannot be invoked from outside the object and is used to support the activities of other methods in the class.

c. A public variable is a variable that can be directly accessed and modified by a client. This explicitly violates the principle of encapsulation and therefore should be avoided.

d. A private variable is a variable that can be accessed and modified only from within the class. Variables almost always are declared with private visibility.

SRA 5.10 An object's interface is the set of public operations (methods) defined on it. That is, the interface establishes the set of services the object will perform for the rest of the system.

SRA 5.11 Although a method is defined in a class, it is invoked through a particular object to indicate which object of that class is being affected. For example, the `Student` class may define the operation that computes the grade point average of a student, but the operation is invoked through a particular `Student` object to compute the GPA for that student. The exception to this rule is the invocation of a static method, which is executed through the class name and does not affect any particular object.

SRA 5.12 An invoked method may return a value, which means it computes a value and provides that value to the calling method. The calling method usually uses the invocation, and thus its return value, as part of a larger expression.

SRA 5.13 An explicit `return` statement is used to specify the value that is returned from a method. The type of the return value must match the return type specified in the method definition.

SRA 5.14 A `return` statement is required in methods that have a return type other than `void`. A method that does not return a value could use a `return` statement without an expression, but it is not necessary. Only one `return` statement should be used in a method.

SRA 5.15 An actual parameter is a value sent to a method when it is invoked. A formal parameter is the corresponding variable in the header of the method declaration; it takes on the value of the actual parameter so that it can be used inside the method.

SRA 5.16 Constructors are special methods in an object that are used to initialize the object when it is instantiated. A constructor has the same name as its class, and it does not return a value.

SRA 5.17 Memory space for an instance variable is created for each object that is instantiated from a class. A static variable is shared among all objects of a class.

SRA 5.18 The `main` method of any program is static and can refer only to static or local variables. Therefore, a `main` method cannot refer to instance variables declared at the class level.

SRA 5.19 A dependency relationship between two classes occurs when one class relies on the functionality of the other. It is often referred to as a "uses" relationship.

SRA 5.20 Overloaded methods are distinguished by having a unique signature, which includes the number, order, and type of the parameters. The return type is not part of the signature.

SRA 5.21 Method decomposition is the process of dividing a complex method into several support methods to get the job done. This simplifies and facilitates the design of the program.

SRA 5.22 A method executed through an object might take as a parameter another object created from the same class. For example, the `concat` method of the `String` class is executed through one `String` object and takes another `String` object as a parameter.

SRA 5.23 An aggregate object is an object that has other objects as instance data. That is, an aggregate object is one that is made up of other objects.

SRA 5.24 The `this` reference always refers to the currently executing object. A nonstatic method of a class is written generically for all objects of the class, but it is invoked through a particular object. The `this` reference, therefore, refers to the object through which that method is currently being executed.

SRA 5.25 Objects are passed to methods by copying the reference to the object (its address). Therefore, the actual and formal parameters of a method become aliases of each other.

SRA 5.26 Defect testing is the act of testing to locate errors in a program.

SRA 5.27 A debugger is a software application that allows us to observe and manipulate the inner workings of a program as it executes.