4 writing classes

In Chapters 2 and 3 we used objects and classes for the services they provide. We also explored several basic programming statements. We are now ready to design more complex software by creating our own classes to define objects that perform whatever services we define. This chapter explores the details of class definitions, including the structure and semantics of methods and the scope and encapsulation of data.

chapter objectives

- Define classes that act like blueprints for new objects, made of variables and methods.
- Explain encapsulation and Java modifiers.
- Explore the details of method declarations.
- Review method invocation and parameter passing.
- Explain and use method overloading.
- Learn to divide complicated methods into simpler, supporting methods.
- Describe relationships between objects.
- Create graphics-based objects.

4.0 objects revisited

In Chapters 2 and 3 we created objects from classes in the Java standard class library. We didn't need to know the details of how the classes did their jobs; we simply trusted them to do so. That is one of the advantages of abstraction. Now we are ready to write our own classes.

First, let's review the concept of an object and explore it in more detail. Think about objects in the world around you. How would you describe them? Let's use a ball as an example. A ball has a diameter, color, and elasticity. We say the properties that describe an object, called *attributes*, define the object's *state of being*. We also describe a ball by what it does, such as the fact that it can be thrown, bounced, or rolled. These are the object's *behavior.*

All objects have a state and a set of behaviors. So do software objects. The values of an object's variables describe the object's state, and the methods define the object's behaviors.

key concept

Each object has a state and a set of behaviors. The values of an object's variables define its state and the methods define its behaviors.

Consider a computer game that uses a ball. The ball could be represented as an object. It could have variables to store its size and location, and methods that draw it on the screen and calculate how it moves when thrown, bounced, or rolled. The variables and methods defined in the ball object are the state and behavior of the ball in the computerized ball game.

Each object has its own state. Each ball object has a particular location, for instance, which is different from the location of all other balls. Behaviors, though, tend to apply to all objects of a particular type. For instance, any ball can be thrown, bounced, or rolled. The act of rolling a ball is generally the same for all balls.

The state of an object and that object's behaviors work together. How high a ball bounces depends on its elasticity. A basketball will bounce higher than, say, a golf ball. The action is the same, but the result depends on that particular object's state. An object's behavior often changes its state. For example, when a ball is rolled, its location changes.

Any object can be described in terms of its state and behavior. Let's take another example. In software that is used to manage a school, a student could be represented as an object. The collection of all such objects represents the entire student body at the school. Each student has a state. That is, each student object would contain the variables that store information about a particular student, such as name, address, courses taken, grades, and grade point average. A student object also has behaviors. For example, the class of the student object may contain a method to add a new course.

Although software objects often represent physical things, like balls and students, they don't have to. For example, an error message can be an object, with its state being the text of the message and behaviors, including printing the error message. A common mistake made by new programmers is to limit the possibilities to physical things.

classes

An object is defined by a class. A class is the model, pattern, or blueprint from which an object is created. Consider the blueprint created by an architect when designing a house. The blueprint defines the important characteristics of the house—its walls, windows, doors, electrical outlets, and so on. Once the blueprint is created, several houses can be built using it, as shown in Figure 4.1.

The houses built from the blueprint are different. They are in different locations, they have different addresses, contain different furniture, and different people live in them. Yet in many ways they are the "same" house. The layout of the rooms, number of windows and doors, and so on are the same in each. To create a different house, we would need a different blueprint.

A class is a blueprint of an object. However, a class is not an object any more than a blueprint is a house. In general, no space to store data values is reserved in a class. To create space to store data values, we must instantiate

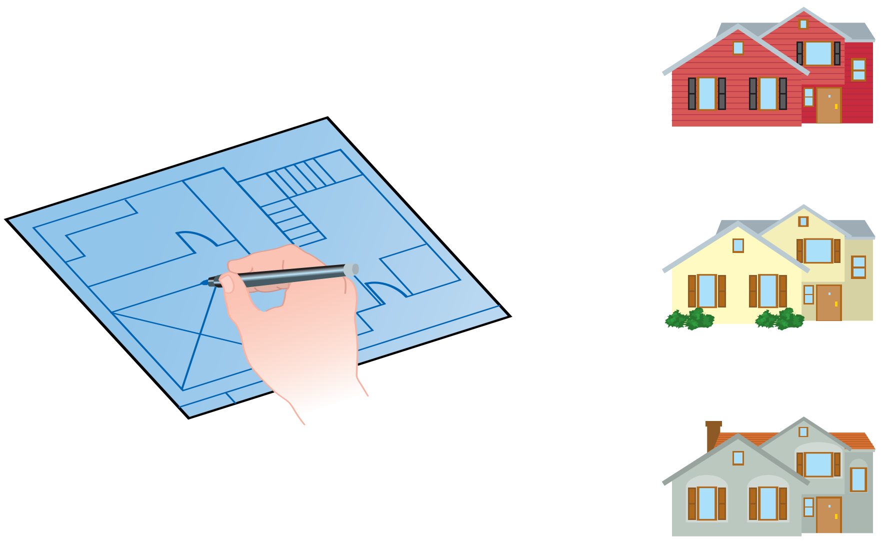


figure 4.1 A house blueprint and three houses created from it

key concept

A class is a blueprint of an object; it has no memory space for data. Each object has its own data space, thus its own state.

one or more objects from the class. (We discuss the exception to this rule in the next chapter.) Each object is an instance of a class. Each object has space for its own data, which is why each object can have its own state.

4.1 anatomy of a class

A class contains the declarations of the data that will be stored in each instantiated object and the declarations of the methods that can be invoked using an object. These are called the *members* of the class, as shown in Figure 4.2.

Consider the `CountFlips` program shown in Listing 4.1. It uses an object that represents a coin that can be flipped to get a random "heads" or "tails." The `CountFlips` program simulates the flipping of a coin 1,000 times to see how often it comes up heads or tails. The `myCoin` object is instantiated from a class called `Coin`.

Listing 4.2 shows the `Coin` class used by the `CountFlips` program. A class, and therefore any object created from it, is made up of data values (variables and constants) and methods. In the `Coin` class, we have two integer constants, `HEADS` and `TAILS`, and one integer variable, `face`. The rest of the `Coin` class is made up of the `Coin` constructor and three regular methods: `flip`, `isHeads`, and `toString`.

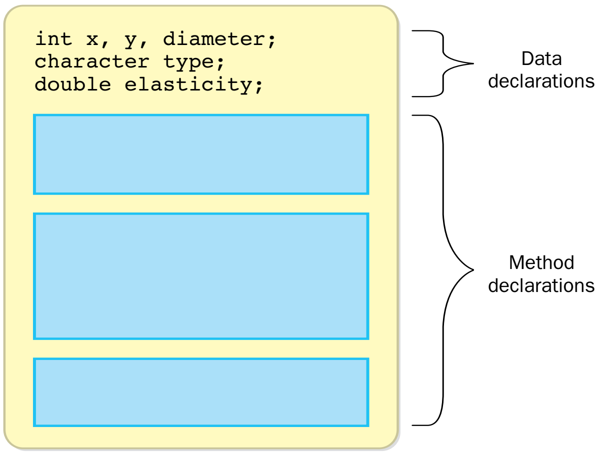


figure 4.2 The members of a class: data and method declarations

listing 4.1

```java
//********************************************************************
//  CountFlips.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of a programmer-defined class.
//********************************************************************

public class CountFlips
{
   //-----------------------------------------------------------------
   //  Flips a coin multiple times and counts the number of heads
   //  and tails that result.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int NUM_FLIPS = 1000;
      int heads = 0, tails = 0;

      Coin myCoin = new Coin();  // instantiate the Coin object

      for (int count=1; count <= NUM_FLIPS; count++)
      {
         myCoin.flip();

         if (myCoin.isHeads())
            heads++;
         else
            tails++;
      }

      System.out.println ("The number flips: " + NUM_FLIPS);
      System.out.println ("The number of heads: " + heads);
      System.out.println ("The number of tails: " + tails);
   }
}
```

output

```
The number flips: 1000
The number of heads: 486
The number of tails: 514
```

listing
4.2

```
//********************************************************************
//  Coin.java       Author: Lewis/Loftus/Cocking
//
//  Represents a coin with two sides that can be flipped.
//********************************************************************

import java.util.Random;

public class Coin
{
   private final int HEADS = 0;
   private final int TAILS = 1;

   private int face;

   //-----------------------------------------------------------------
   //  Sets up the coin by flipping it initially.
   //-----------------------------------------------------------------
   public Coin ()
   {
      flip();
   }

   //-----------------------------------------------------------------
   //  Flips the coin by randomly choosing a face value.
   //-----------------------------------------------------------------
   public void flip ()
   {
      face = (int) (Math.random() * 2);
   }

   //-----------------------------------------------------------------
   //  Returns true if the current face of the coin is heads.
   //-----------------------------------------------------------------
   public boolean isHeads ()
   {
      return (face == HEADS);
   }

   //-----------------------------------------------------------------
   //  Returns the current face of the coin as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String faceName;
```

listing 4.2 continued

```
      if (face == HEADS)
         faceName = "Heads";
      else
         faceName = "Tails";

      return faceName;
   }
}
```

Remember from Chapter 2 that constructors are special methods that have the same name as the class. The `Coin` constructor gets called when the `new` operator is used to create a new instance of the `Coin` class. The rest of the methods in the `Coin` class define the various services provided by `Coin` objects.

Note that a header block of documentation is used to explain the purpose of each method in the class. This practice is not only important for anyone trying to understand the software, it also separates the code so that it's easy to see where one method ends and the next begins. The definitions of these methods have many parts, and we'll look at them in later sections of this chapter.

Figure 4.3 lists the services defined in the `Coin` class. The `Coin` class looks like other classes that we've used in previous examples. The only important difference is that the `Coin` class is not part of the Java standard class library. We wrote it ourselves.

`Coin ()`
   Constructor: sets up a new `Coin` object with a random initial face.

`void flip ()`
   Flips the coin.

`boolean isHeads ()`
   Returns true if the current face of the coin shows heads.

`String toString ()`
   Returns a string describing the current face of the coin.

figure 4.3 Some methods of the `Coin` class

For most of the examples in this book, we store each class in its own file. Java lets you put several classes in one file. If a file contains several classes, only one of those classes can be declared using the reserved word `public`. Also, the name of the public class must match the name of the file. For instance, class `Coin` is stored in a file called `Coin.java`.

instance data

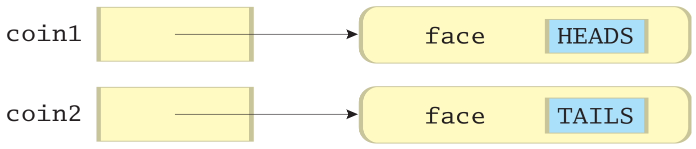
In the `Coin` class, the constants `HEADS` and `TAILS`, and the variable `face` are declared inside the class, but not inside any method. The location at which a variable is declared defines its *scope,* which is the area in a program where that variable can be referenced. Because they are declared at the class level (not within a method), these variables and constants can be referenced in any method of the class.

key concept

The scope of a variable, which determines where it can be referenced, depends on where it is declared.

Attributes such as the variable `face` are also called *instance data* because memory space is created for each instance of the class that is created. Each `Coin` object, for example, has its own `face` variable with its own data space. Therefore at any point in time, two `Coin` objects can have their own states: one can be showing heads and the other can be showing tails, for example.

We can depict this situation as follows:



The `coin1` and `coin2` reference variables point to (that is, contain the address of) their respective `Coin` objects. Each object contains a `face` variable with its own memory space. Thus each object can store different values for its instance data.

The program `FlipRace` shown in Listing 4.3 declares two `Coin` objects. They are used in a race to see which coin will flip first to three heads in a row.

The output of the `FlipRace` program shows the results of each coin flip on each turn. The object reference variables, `coin1` and `coin2`, are used in the `println` statement. When an object is used as an operand of the string concatenation operator (+), that object's `toString` method is automatically called to get a string representation of the object. The `toString` method is also called if an object is sent to a `print` or `println` method by itself. If no `toString` method is defined for a class, a default version returns a string

listing 4.3

```
//********************************************************************
//  FlipRace.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the existence of separate data space in multiple
//  instantiations of a programmer-defined class.
//********************************************************************

public class FlipRace
{
   //-----------------------------------------------------------------
   //  Flips two coins until one of them comes up heads three times
   //  in a row.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int GOAL = 3;
      int count1 = 0, count2 = 0;

      // Create two separate coin objects
      Coin coin1 = new Coin();
      Coin coin2 = new Coin();

      while (count1 < GOAL && count2 < GOAL)
      {
         coin1.flip();
         coin2.flip();

         // Print the flip results (uses Coin's toString method)
         System.out.print ("Coin 1: " + coin1);
         System.out.println ("   Coin 2: " + coin2);

         // Increment or reset the counters
         if (coin1.isHeads())
            count1++;
         else
            count1 = 0;
         if (coin2.isHeads())
            count2++;
         else
            count2 = 0;
      }

      // Determine the winner
      if (count1 < GOAL)
         System.out.println ("Coin 2 Wins!");
      else
         if (count2 < GOAL)
            System.out.println ("Coin 1 Wins!");
```

listing 4.3 continued

```
         else
            System.out.println ("It's a TIE!");
   }
}
```

output

```
Coin 1: Heads   Coin 2: Tails
Coin 1: Heads   Coin 2: Tails
Coin 1: Tails   Coin 2: Heads
Coin 1: Tails   Coin 2: Heads
Coin 1: Heads   Coin 2: Tails
Coin 1: Tails   Coin 2: Heads
Coin 1: Heads   Coin 2: Tails
Coin 1: Heads   Coin 2: Heads
Coin 1: Heads   Coin 2: Tails
Coin 1 Wins!
```

that contains the name of the class, together with other information. It is usually a good idea to define a specific `toString` method for a class.

We have now used the `Coin` class to create objects in two separate programs (`CountFlips` and `FlipRace`). This is no different from using the `String` class in whatever program we need it. When designing a class, it is always good to try to give the class behaviors that can be used in other programs, not just the program you are creating at the moment.

Java automatically initializes any variables declared at the class level. For example, all variables of numeric types such as `int` and `double` are initialized to zero. However, it is good practice to initialize variables explicitly (usually in a constructor) so that anyone reading the code will clearly understand what you are doing.

encapsulation and visibility modifiers

We can think about an object in one of two ways, depending on what we are trying to do. First, when we are designing and implementing an object, we need to think about how an object works. That is, we have to design the class—we have to define the variables that will be held in the object and write the methods that make the object useful.

However, when we are designing a solution to a larger problem, we have to think about how the objects in the program work with each other. At that level, we have to think only about the services that an object provides, not the details of how those services are provided. As we discussed in Chapter 2, an object provides a level of abstraction that lets us focus on the big picture when we need to.

This abstraction works only if we are careful to respect its boundaries. An object should be *self-governing*, which means that the variables contained in an object should be changed only within the object. Only the methods within an object should have access to the variables in that object. For example, the methods of the `Coin` class should be the only methods that can change the value of the `face` variable. We should make it difficult or impossible for code outside of a class to "reach in" and change the value of a variable that is declared inside the class.

In Chapter 2 we mentioned that the *object-oriented* term for this is *encapsulation*. An object should be encapsulated from the rest of the system. It should work with other parts of a program only through the object's own methods. These methods define the *interface* between that object and the program that uses it.

key concept

Objects should be encapsulated. The rest of a program should work with an object only through a well-defined interface.

Figure 4.4 shows how encapsulation works. The code that uses an object, called the *client* of an object, should not be able to get to variables directly. The client should interact with the object's methods, and those methods then interact with the data encapsulated in the object. For example, the `main` method in the `CountFlips` program calls the `flip` and `isHeads` methods of the `myCoin` object. The `main` method should not (and in fact cannot) get to the `face` variable directly.

In Java, we create object encapsulation using *modifiers*. A modifier is a Java reserved word that names special characteristics of a programming language construct. We've already seen one modifier, `final`, which we use to declare a constant. Java has several modifiers that can be used in different

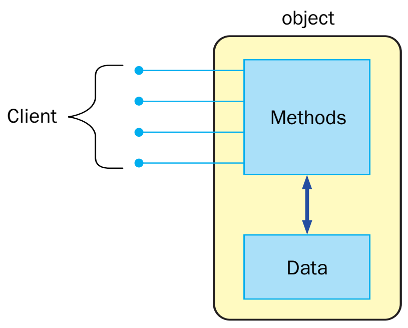


figure 4.4 A client interacting with the methods of an object

ways. Some modifiers can be used together. We discuss various Java modifiers at appropriate points throughout this book.

Some Java modifiers are called *visibility modifiers*. They control whether client code can "see" what's "inside" an object. The reserved words `public` and `private` are visibility modifiers that can be applied to the variables and methods of a class. If a member of a class has *public visibility,* it can be "seen" (read and perhaps changed) from outside the object. If a member of a class has *private visibility,* it can be used anywhere inside but not outside the class definition.

Public variables let code outside the class reach in and read or change the value of the data. Therefore instance data should be defined with private visibility. Data that is declared as `private` can be read or changed only by the methods of the class, which makes the objects created from that class self-governing. Whether a method has public or private visibility depends on the purpose of that method. Methods that provide services to the client of the class must have public visibility so that they can be invoked by the client. These methods are sometimes called *service methods*. A `private` method cannot be invoked from outside the class. The only purpose of a `private` method is to help the other methods of the class do their job. Therefore they are sometimes called *support methods*. We discuss an example that makes use of several support methods later in this chapter.

The table in Figure 4.5 summarizes the effects of public and private visibility on both variables and methods.

key concept
Instance variables should be declared with private visibility.

Note that a client can still read or change `private` data by invoking service methods that change the data. For example, although the `main` method of the `FlipRace` class cannot directly get to the `face` variable, it can invoke the `flip` service method, which sets the value

	`public`	`private`
Variables	Does not protect data	Protects data
Methods	Provides services to clients	Supports other methods in the class

figure 4.5 The effects of public and private visibility

of `face`. A class must provide service methods for valid client operations. The code of those methods must be carefully designed to allow only appropriate access and valid changes.

Giving constants public visibility is generally considered acceptable because, although their values can be accessed directly, they cannot be changed. That is because constants are declared using the `final` modifier. Keep in mind that encapsulation means that data values should not be able to be *changed* directly by another part of the code. Because constants, by definition, cannot be changed, they don't need to be protected. If we had thought it important to provide external access to the values of the constants `HEADS` and `TAILS` in the `Coin` class, we could have declared them with public visibility.

4.2 anatomy of a method

We've seen that a class is made up of data declarations and method declarations. Let's look at method declarations in more detail.

As we stated in Chapter 1, a method is a group of programming language statements that is given a name. Every method in a Java program is part of a particular class. A *method declaration* defines the code that is executed when the method is invoked.

When a method is called, the statements of that method are executed. When that method is done, control returns to the location where the call was made and execution continues. A method that is called might be part of the same object (defined in the same class) as the method that called it, or it might be part of a different object. If the called method is part of the same object, only the method name is needed to call it. If it is part of a different object, it is invoked through that object's name, as we've seen many times. Figure 4.6 shows what this process looks like.

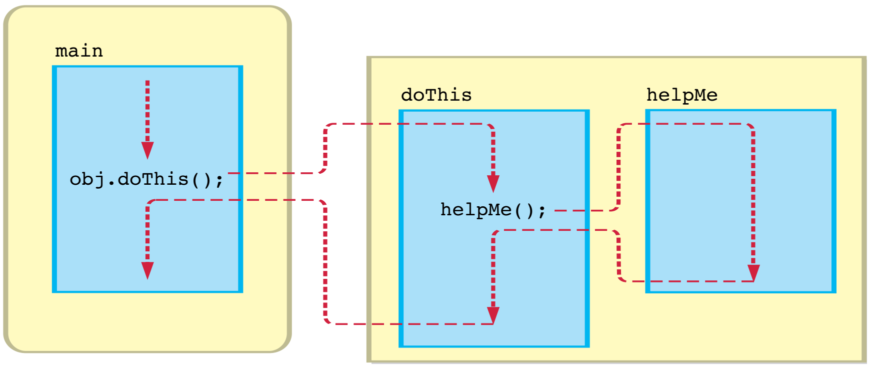


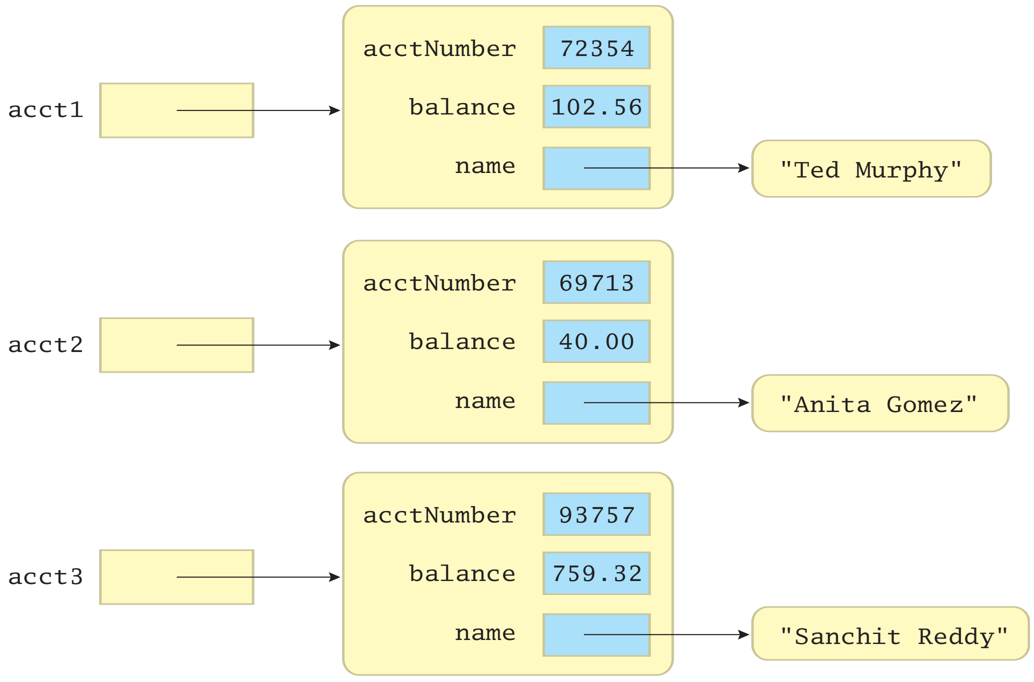
figure 4.6 The flow of control following method invocations

We've defined the `main` method of a program many times in our examples. The `main` method follows the same syntax as all methods. The header of a method includes the type of the return value, the method name, and a list of parameters that the method accepts. The statements that make up the body of the method are defined in a block inside braces.

Let's look at another example. The `Banking` class shown in Listing 4.4 contains a `main` method that creates and then calls methods on a few `Account` objects. The `Banking` program doesn't really do anything useful except show how to interact with `Account` objects. Such programs are often called *driver programs* because all they do is drive other, more interesting parts of a program. They are often used for testing.

The `Account` class represents a basic bank account and is shown in Listing 4.5. Its data values include the account number, the balance, and the name of the account's owner. The interest rate is stored as a constant.

The status of the three `Account` objects just after they were created in the `Banking` program could be depicted as follows:



listing
4.4

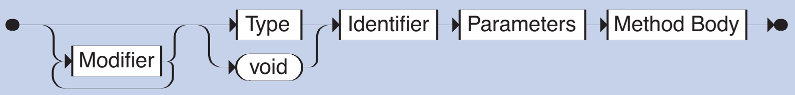
```java
//********************************************************************
//  Banking.java       Author: Lewis/Loftus/Cocking
//
//  Driver to exercise the use of multiple Account objects.
//********************************************************************

public class Banking
{
   //-----------------------------------------------------------------
   //  Creates some bank accounts and requests various services.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
      Account acct2 = new Account ("Anita Gomez", 69713, 40.00);
      Account acct3 = new Account ("Sanchit Reddy", 93757, 759.32);

      acct1.deposit (25.85);

      double gomezBalance = acct2.deposit (500.00);
      System.out.println ("Gomez balance after deposit: " +
                          gomezBalance);

      System.out.println ("Gomez balance after withdrawal: " +
                          acct2.withdraw (430.75, 1.50));

      acct3.withdraw (800.00, 0.0);  // exceeds balance

      acct1.addInterest();
      acct2.addInterest();
      acct3.addInterest();

      System.out.println ();
      System.out.println (acct1);
      System.out.println (acct2);
      System.out.println (acct3);
   }
}
```

listing 4.4 continued

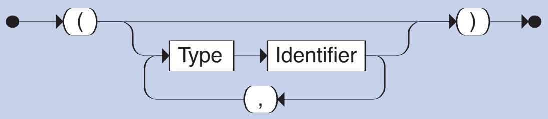
output

```
Gomez balance after deposit: 540.0
Gomez balance after withdrawal: 107.75

Error: Insufficient funds.
Account: 93757
Requested: $800.00
Available: $759.32

72354   Ted Murphy      $132.90
69713   Anita Gomez      $111.52
93757   Sanchit Reddy   $785.90
```

Method Declaration



Parameters



A method is defined by optional modifiers, followed by a return Type, followed by an Identifier that determines the method name, followed by a list of Parameters, followed by the Method Body. The return Type indicates the type of value that will be returned by the method, which may be void. The Method Body is a block of statements that executes when the method is called.

Example:

```
public void instructions (int count)
{
   System.out.println ("Follow all instructions.");
   System.out.println ("Use no more than " + count +
                       " turns.");
}
```

listing
4.5

```
//********************************************************************
//  Account.java       Author: Lewis/Loftus/Cocking
//
//  Represents a bank account with basic services such as deposit
//  and withdraw.
//********************************************************************

import java.text.NumberFormat;

public class Account
{
   private NumberFormat fmt = NumberFormat.getCurrencyInstance();

   private final double RATE = 0.035;  // interest rate of 3.5%

   private int acctNumber;
   private double balance;
   private String name;

   //-----------------------------------------------------------------
   //  Sets up the account by defining its owner, account number,
   //  and initial balance.
   //-----------------------------------------------------------------
   public Account (String owner, int account, double initial)
   {
      name = owner;
      acctNumber = account;
      balance = initial;
   }

   //-----------------------------------------------------------------
   //  Validates the transaction, then deposits the specified amount
   //  into the account. Returns the new balance.
   //-----------------------------------------------------------------
   public double deposit (double amount)
   {
      if (amount < 0)  // deposit value is negative
      {
         System.out.println ();
         System.out.println ("Error: Deposit amount is invalid.");
         System.out.println (acctNumber + "  " + fmt.format(amount));
      }
      else
         balance = balance + amount;

      return balance;
   }
```

listing 4.5 continued

```
   //-----------------------------------------------------------------
   //  Validates the transaction, then withdraws the specified amount
   //  from the account. Returns the new balance.
   //-----------------------------------------------------------------
   public double withdraw (double amount, double fee)
   {
      amount += fee;

      if (amount < 0)  // withdraw value is negative
      {
         System.out.println ();
         System.out.println ("Error: Withdraw amount is invalid.");
         System.out.println ("Account: " + acctNumber);
         System.out.println ("Requested: " + fmt.format(amount));
      }
      else
         if (amount > balance)  // withdraw value exceeds balance
         {
            System.out.println ();
            System.out.println ("Error: Insufficient funds.");
            System.out.println ("Account: " + acctNumber);
            System.out.println ("Requested: " + fmt.format(amount));
            System.out.println ("Available: " + fmt.format(balance));
         }
         else
            balance = balance - amount;

      return balance;
   }

   //-----------------------------------------------------------------
   //  Adds interest to the account and returns the new balance.
   //-----------------------------------------------------------------
   public double addInterest ()
   {
      balance += (balance * RATE);
      return balance;
   }

   //-----------------------------------------------------------------
   //  Returns the current balance of the account.
   //-----------------------------------------------------------------
   public double getBalance ()
   {
      return balance;
   }
```

listing 4.5 continued

```
   //-----------------------------------------------------------------
   //  Returns the account number.
   //-----------------------------------------------------------------
   public int getAccountNumber ()
   {
      return acctNumber;
   }

   //-----------------------------------------------------------------
   //  Returns a one-line description of the account as a string.
   //-----------------------------------------------------------------
   public String toString ()
   {
      return (acctNumber + "\t" + name + "\t" + fmt.format(balance));
   }
}
```

The methods of the `Account` class do things like make deposits and withdrawals. The program also makes sure that the data are valid, such as preventing the withdrawal of a negative amount. We explore the methods of the `Account` class in detail in the following sections.

the return statement

The return type in the method header can be a primitive type, class name, or the reserved word `void`. When a method does not return any value, `void` is used as the return type, as is always done with the `main` method.

key concept: A return value must match the return type in the method header.

A method that returns a value must have a *return statement*. After a `return` statement is executed, control is immediately returned to the statement in the calling method, and processing continues there. A `return` statement is the reserved word `return` followed by an expression that dictates the value to be returned. The expression must match the return type in the method header.

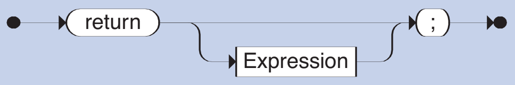
For example, the `return` statement in a method that returns a `double` could look like this:

```
return sum/2.0;
```

If the variable `sum` had the value 14 when the `return` statement was reached, then the value returned would be 7.0.

A method that does not return a value does not usually need a `return` statement because it automatically returns to the calling method when it is

Return Statement



A `return` statement is the `return` reserved word followed by an optional Expression. After the statement executes, control is immediately returned to the calling method, returning the value defined by Expression.

Examples:

```
return;

return (distance * 4);
```

done. A method with a `void` return type may, however, have a `return` statement without an expression.

It is usually not good practice to use more than one `return` statement in a method. In general, the `return` statement should be the last line of the method body.

Many of the methods of the `Account` class in Listing 4.5 return a `double` that represents the balance of the account. Constructors do not have a return type at all (not even `void`), so they cannot have a `return` statement. We discuss constructors in more detail in a later section.

Note that a return value can be ignored. In the `main` method of the `Banking` class, sometimes the value that is returned by a method is used in some way, and sometimes the value returned is simply ignored.

parameters

Remember from Chapter 2 that a parameter is a value that is passed into a method when it is invoked. The *parameter list* in the header of a method lists the types of the values that are passed and their names.

The parameters in the header of the method declaration are called *formal parameters*. The values passed into a method are called *actual parameters*. The parameter list is always in parentheses after the method name. If there are no parameters, the parentheses are empty.

key concept

When a method is called, the actual parameters are copied into the formal parameters. The types of the corresponding parameters must match.

The formal parameters are identifiers that act as variables inside the method. Their initial values come from the actual parameters in the invocation. When a method is called, the value in each actual parameter is copied and stored in the matching formal parameter. Actual parameters can be literals, variables, or full expressions. If an expression is used as an actual parameter, it is fully evaluated before the method call and the result is passed as the parameter.

The parameter lists in the invocation and the method declaration must match up. That is, the value of the first actual parameter is copied into the first formal parameter, the second actual parameter into the second formal parameter, and so on, as shown in Figure 4.7. The types of the actual parameters must match the types of the formal parameters.

The `deposit` method of the `Account` class in Listing 4.5, for example, takes one formal parameter called `amount`, of type `double`, representing the amount to be deposited into the account. Each time the method is invoked in the `main` method of the `Banking` class, one literal value of type `double` is passed as an actual parameter. For example, in the first call to `deposit`, the actual parameter is `25.85`. When the program is running and the statement `acct1.deposit(25.85)` is invoked, control transfers to the `deposit` method and the parameter `amount` takes on the value `25.85`. In the case of the `withdraw` method, two parameters of type `double` are expected. The types and number of parameters must match or you will get an error message.

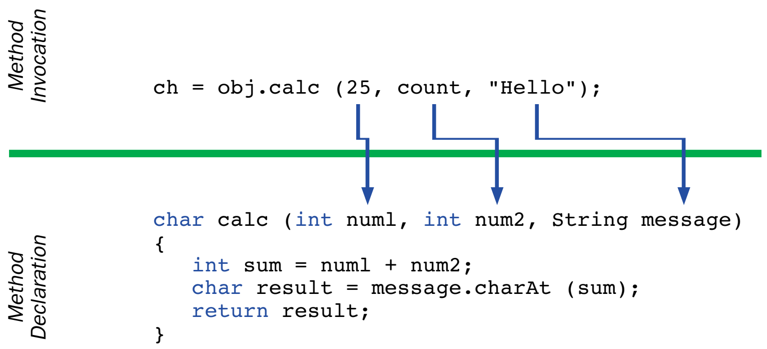
Constructors can also take parameters, as we discuss in the next section. We discuss parameter passing in more detail in Chapter 5.

preconditions and postconditions

Preconditions and postconditions allow us to reason about methods, helping to make sure they are correct. A *precondition* is a condition that should be true when a method is called. A *postcondition* is a condition that should be true when a method finishes executing,

key concept

A precondition is a condition that should be true when a method is called. A postcondition is a condition that should be true when a method finishes executing.



```
ch = obj.calc (25, count, "Hello");
```

```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);
   return result;
}
```

figure 4.7 Passing parameters from the method invocation to the declaration assuming that the precondition was true. If a method's precondition is violated, the postcondition does not apply.

For example, the `deposit` method requires that `amount` be greater than or equal to 0, otherwise an error is printed and the deposit is not performed. If the precondition is true, then when `deposit` finishes executing, the balance will be the original balance plus `amount`. This is the postcondition.

Pre- and postconditions can be stated in comments above the method declaration. For the `deposit` method, we could include the following comments:

```
//  Precondition: amount >= 0
//  Postcontition: balance = balance + amount
```

Preconditions and postconditions are both a kind of *assertion*, which is a logical statement that can be true or false, and represents an assumption being made about a program. For example, consider the following test of whether a number is odd or even:

```
if (num % 2 == 1)
   System.out.println ("odd");
else // num % 2 == 0
   System.out.println ("even");
```

The comment after the `else`, `// num % 2 == 0`, is an assertion. The programmer is assuming that if the remainder of `num` divided by `2` is not `1`, then it must be `0`.

Including assertions in comments clarifies the programmer's assumptions about the program, but comments don't affect the execution of a program. The Java language includes an assertion statement that does affect the execution of a program, allowing assumptions to be tested while a program is running:

```
if (num % 2 == 1)
   System.out.println ("odd");
else // num % 2 == 0
{
   assert num % 2 == 0;
   System.out.println ("even");
}
```

If the expression after the `assert` keyword is false, the program will end with an `AssertionError`. Assertions help ensure that a program is correct by uncovering false assumptions the programmer may have. In the code segment above, the programmer is assuming that `num % 2` will always be `0` or `1`, but actually, if `num` is negative, `num % 2` could be `-1`!

Since assertions are normally used only during testing, they are turned off by default in Java. Consult the documentation for your development environment to find out how to turn assertions on (otherwise any `assert` statements are ignored).

Note that while preconditions, postconditions, and the concept of assertions are required for the AP* exam, the Java assertion statement is not, and we do not use it in this book.

constructors

As we stated in Chapter 2, a constructor is like a method that is called when an object is instantiated. When we define a class, we usually define a constructor to help us set up the class. In particular, we often use a constructor to initialize the variables associated with each object.

A constructor is different from a regular method in two ways. First, the name of a constructor is the same name as the class. Therefore the name of the constructor in the `Coin` class is `Coin`, and the name of the constructor of the `Account` class is `Account`. Second, a constructor cannot return a value and does not have a return type specified in the method header. A constructor cannot be called like other methods. It is called only when an object is first created.

key concept

A constructor cannot have any return type, even `void`.

A common mistake made by programmers is to put a `void` return type on a constructor. As far as the compiler is concerned, putting any return type on a constructor, even `void`, turns it into a regular method that happens to have the same name as the class. That means it cannot be invoked as a constructor. This leads to error messages that are sometimes difficult to figure out.

A constructor is generally used to initialize the newly instantiated object. For instance, the constructor of the `Coin` class calls the `flip` method to determine the initial face value of the coin. The constructor of the `Account` class sets the values of the instance variables to the values passed in as parameters to the constructor.

We don't have to define a constructor for every class. Each class has a *default constructor* that takes no parameters and is used if we don't provide our own. This default constructor generally has no effect on the newly created object.

local data

key concept

A variable declared in a method is local to that method and cannot be used outside of it.

As we described earlier in this chapter, the scope of a variable (or constant) is the part of a program where a valid reference to that variable can be made. A variable can be declared inside a method, making it *local data* as opposed to instance data. Recall that instance data is declared in a class but not inside any particular method. Local data has scope limited to only the method where it is declared. The `faceName` variable declared in the `toString` method of the `Coin` class is local data. Any reference to `faceName` in any other method of the `Coin` class would cause an error message. A local variable simply does not exist outside of the method in which it is declared. Instance data, declared at the class level, has a scope of the entire class; any method of the class can refer to it.

Because local data and instance data have different levels of scope, it's possible to declare a local variable inside a method using the same name as an instance variable declared at the class level. In the method that name will reference the local version of the variable. This naming practice could confuse anyone reading the code, so it should be avoided.

The formal parameter names in a method header serve as local data for that method. They don't exist until the method is called, and they stop existing when the method ends. For example, although `amount` is the name of the formal parameter in both the `deposit` and `withdraw` method of the `Account` class, each is a separate piece of local data that doesn't exist until the method is invoked.

accessors and mutators

key concept Most objects contain accessor and mutator methods to allow the client to manage data in a controlled manner.

Recall that instance data is generally declared with private visibility. Because of this, a class usually provides services to access and modify data values. An *accessor method* provides read-only access to a particular value. Likewise, a *mutator method*, sometimes called a modifier method, changes a particular value.

Generally, accessor method names have the form `getX`, where `X` is the value to which it provides access. Likewise, mutator method names have the form `setX`, where `X` is the value they are setting. Therefore these types of methods are sometimes referred to as "getters" and "setters."

Some methods may provide accessor and/or mutator capabilities as a side effect of their primary purpose. For example, the `flip` method of the `Coin` class changes the `face` of the coin, and returns that new value as well. Note that the code of the `flip` method is careful to keep the face of the coin in the valid range (0 or 1). Service methods must be carefully designed to permit only appropriate access and valid changes.

In the next section we will see an example of accessor and mutator methods.

4.3 method overloading

When a method is invoked, control transfers to the code that defines the method. After the method finishes, control returns to the location of the call, and processing continues.

Often the method name is enough 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 several methods. This is called *method overloading*. It is useful when you need to perform similar operations on different types of data.

The compiler must still be able to match up each invocation with a specific method declaration. If the method name for two or more methods is the same, then additional information is used to tell which version is being invoked. In Java, a method name can be used for several methods as long as the number of parameters, the types of those parameters, and/or the order of the types of parameters is different. 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 its definition.

key concept

You can tell the versions of an overloaded method apart by their signature, which is the number, type, and order of the parameters.

The compiler must be able to examine a method invocation, including the parameter list, to determine which method is being called. If you try to create two method names with the same signature, you will get an error message.

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

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

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

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

```
System.out.println ("The total number of students is: ");
System.out.println (count);
```

The first line calls the `println` that accepts a string. The second line calls the version that accepts an integer.

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

```
System.out.println ("The total number of students is: " +
                    count);
```

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 called.

Constructors are a good candidates for overloading. Several versions of a constructor give us several ways to set up an object. For example, the `SnakeEyes` program shown in Listing 4.6 instantiates two `Die` objects and initializes them using different constructors.

The purpose of the program is to roll the dice and count the number of times both dice show a 1 on the same throw (snake eyes). In this case, however, one die has 6 sides and the other has 20 sides. Each `Die` object is initialized using different constructors of the `Die` class. Listing 4.7 shows the `Die` class.

Both `Die` constructors have the same name, but one takes no parameters and the other takes an integer as a parameter. The compiler can examine the invocation and determine which version of the method is intended.

The `getFaceValue` method is an accessor method since it provides read-only access to `faceValue`. A mutator method for `faceValue` could look like this:

```
public void setFaceValue (int Value)
{
     facevalue = value;
}
```

However, this is not a good design because a client could use the method to change the face value to an invalid value. Writing a better mutator method is left as an exercise.

listing 4.6

```
//********************************************************************
//  SnakeEyes.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of a class with overloaded constructors.
//********************************************************************

public class SnakeEyes
{
   //-----------------------------------------------------------------
   //  Creates two die objects, then rolls both dice a set number of
   //  times, counting the number of snake eyes that occur.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int ROLLS = 500;
      int snakeEyes = 0, num1, num2;

      Die die1 = new Die();     // creates a six-sided die
      Die die2 = new Die(20);   // creates a twenty-sided die

      for (int roll = 1; roll <= ROLLS; roll++)
      {
         num1 = die1.roll();
         num2 = die2.roll();

         if (num1 == 1 && num2 == 1)  // check for snake eyes
            snakeEyes++;
      }

      System.out.println ("Number of rolls: " + ROLLS);
      System.out.println ("Number of snake eyes: " + snakeEyes);
      System.out.println ("Ratio: " + (double)snakeEyes/ROLLS);
   }
}
```

output

```
Number of rolls: 500
Number of snake eyes: 6
Ratio: 0.012
```

listing 4.7

```java
//********************************************************************
//  Die.java       Author: Lewis/Loftus/Cocking
//
//  Represents one die (singular of dice) with faces showing values
//  between 1 and the number of faces on the die.
//********************************************************************

import java.util.Random;

public class Die
{
   private final int MIN_FACES = 4;

   private static Random generator = new Random();
   private int numFaces;   // number of sides on the die
   private int faceValue;  // current value showing on the die

   //-----------------------------------------------------------------
   //  Defaults to a six-sided die. Initial face value is 1.
   //-----------------------------------------------------------------
   public Die ()
   {
      numFaces = 6;
      faceValue = 1;
   }

   //-----------------------------------------------------------------
   //  Explicitly sets the size of the die. Defaults to a size of
   //  six if the parameter is invalid.  Initial face value is 1.
   //-----------------------------------------------------------------
   public Die (int faces)
   {
      if (faces < MIN_FACES)
         numFaces = 6;
      else
         numFaces = faces;

      faceValue = 1;
   }

   //-----------------------------------------------------------------
   //  Rolls the die and returns the result.
   //-----------------------------------------------------------------
   public int roll ()
   {
      faceValue = generator.nextInt(numFaces) + 1;
```

listing 4.7 continued

```
      return faceValue;
   }

   //-----------------------------------------------------------------
   //  Returns the current die value.
   //-----------------------------------------------------------------
   public int getFaceValue ()
   {
      return faceValue;
   }
}
```

4.4 method decomposition

Sometimes we want to do something so complicated we need more than one method. One thing we can do is to break a method into several simpler 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 changed by moving the first letter of the word to the end and adding "ay." 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" added on the end, turning the word *enough* into *enoughyay*. Pairs of letters, called blends, such as "ch" and "st" are moved to the end together, so *grapefruit* becomes *apefruitgray*.

The `PigLatin` program shown in Listing 4.8 reads one or more sentences, translating each into Pig Latin.

The real workhorse behind the `PigLatin` program is the `PigLatinTranslator` class, shown in Listing 4.9. An object of type `PigLatinTranslator` provides a 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.

listing 4.8

```
//********************************************************************
//  PigLatin.java       Author: Lewis/Loftus/Cocking
//
//  Driver to exercise the PigLatinTranslator class.
//********************************************************************

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 = "y";
      Scanner scan = new Scanner (System.in);
      PigLatinTranslator translator = new PigLatinTranslator();

      while (another.equalsIgnoreCase("y"))
      {
         System.out.println ();
         System.out.println ("Enter a sentence (no punctuation):");
         sentence = scan.nextLine();

         System.out.println ();
         result = translator.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();
      }
   }
}
```

listing 4.8 continued

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 4.9

```java
//********************************************************************
//  PigLatinTranslator.java       Author: Lewis/Loftus/Cocking
//
//  Represents a translation system from English to Pig Latin.
//  Demonstrates method decomposition and the use of StringTokenizer.
//********************************************************************

import java.util.Scanner;

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

      sentence = sentence.toLowerCase();
      Scanner scan = new Scanner (sentence);
```

listing 4.9 **continued**

```
      while (scan.hasNext())
      {
         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 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 boolean beginsWithVowel (String word)
   {
      String vowels = "aeiou";

      char letter = word.charAt(0);

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

listing 4.9 continued

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

Translating an entire sentence into Pig Latin is not easy. One big method would be very long and difficult to follow. A better solution is to break the `translate` method into simpler methods and use several other support methods to help.

The `translate` method uses a `Scanner` object to separate the string into words. Recall that one role of the `Scanner` class (discussed in Chapter 2) is to separate a string into smaller elements called tokens. In this case, the tokens are separated by space characters so we can use the default white space 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 complicated, so the `translateWord` method uses two other private methods, `beginsWithVowel` and `beginsWithBlend`.

The `beginsWithVowel` method returns a `boolean` value that tells `translateWord` whether the word begins with a vowel. Instead of checking each vowel separately, the code for this method declares a string of all the vowels, and then invokes the `String` method `indexOf` to see whether the first character of the word is in the vowel string. If the 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 has one large expression that makes several calls to the `startsWith` method of the `String` class. If any of these calls returns true, because the word begins with a blend, then the `beginsWithBlend` method returns true as well.

Note that the `translateWord`, `beginsWithVowel`, and `beginsWithBlend` methods are all declared with private visibility. They do not deal directly with clients outside the class. Instead, they only help the `translate` method do its job. Because they have private visibility, they cannot be called from outside this class. If the `main` method of the `PigLatin` class attempted to call the `translateWord` method, for instance, the compiler would issue an error message.

Whenever a method becomes large or complicated, we should consider breaking it into simpler 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, breaking up methods must happen after objects have been broken up.

key concept

A complicated method can be broken into several simpler methods and helped by support methods.

4.5 object relationships

Classes, and their objects, can have particular types of relationships, or associations, to each other. This section looks at associations between objects of the same class. We then explore aggregation, in which one object is made up of other objects, creating a "has-a" relationship.

Inheritance, which we introduced in Chapter 2, is another important relationship between classes. It creates a generalization, or an "is-a" relationship, between classes. We examine inheritance in Chapter 7.

association

Two classes have a general association if they are "aware" of each other. Objects of those classes may use each other. This is sometimes called a "use" *relationship*.

An association can be described in general terms, such as the fact that an `Author` object writes a `Book` object. Because association relationships are so general, they are very useful. We introduce uses of general associations throughout the book.

association between objects of the same class

Some associations occur between two objects of the same class. That is, a method of one object takes as a parameter another object of the same class. The operation performed often involves the internal data of both objects.

key concept A method invoked through one object may take as a parameter another object of the same class.

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

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

The `String` object executing the method (`str1`) attaches 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 `RationalNumbers` program shown in Listing 4.10 works like this. Recall that a rational number is a value that can be represented as a fraction, such as 2/2 (which equals 1), or 35/5 (which equals 7), or 1/2.

listing 4.10

```
//********************************************************************
//  RationalNumbers.java       Author: Lewis/Loftus/Cocking
//
//  Driver to exercise the use of multiple Rational objects.
//********************************************************************

public class RationalNumbers
{
   //-----------------------------------------------------------------
   //  Creates some rational number objects and performs various
   //  operations on them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Rational r1 = new Rational (6, 8);
      Rational r2 = new Rational (1, 3);
      Rational r3, r4, r5, r6, r7;

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

listing 4.10 continued

```
      if (r1.equals(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 `RationalNumbers` program creates two objects representing rational numbers and then does things to them to produce new rational numbers.

The `Rational` class is shown in Listing 4.11. Each object of type `Rational` represents one rational number. The `Rational` class contains operations on rational numbers, such as addition and subtraction.

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

Note that some of the methods in the `Rational` class are `private` because we don't want them executed directly from outside a `Rational` object. They are there only to support the other services of the object.

listing 4.11

```
//********************************************************************
//  Rational.java       Author: Lewis/Loftus/Cocking
//
//  Represents one rational number with a numerator and denominator.
//********************************************************************

public class Rational
{
   private int numerator, denominator;

   //-----------------------------------------------------------------
   //  Sets up the rational number by ensuring a nonzero denominator
   //  and making only the numerator signed.
   //-----------------------------------------------------------------
   public Rational (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 ()
   {
      return denominator;
   }
```

listing 4.11 continued

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

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

      return new Rational (sum, commonDenominator);
   }

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

      return new Rational (difference, commonDenominator);
   }

   //-----------------------------------------------------------------
   //  Multiplies this rational number by the one passed as a
   //  parameter.
   //-----------------------------------------------------------------
   public Rational multiply (Rational op2)
   {
      int numer = numerator * op2.getNumerator();
      int denom = denominator * op2.getDenominator();
```

listing 4.11 continued

```
      return new Rational (numer, denom);
   }

   //-----------------------------------------------------------------
   //  Divides this rational number by the one passed as a parameter
   //  by multiplying by the reciprocal of the second rational.
   //-----------------------------------------------------------------
   public Rational divide (Rational 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 equals (Rational 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;
   }

   //-----------------------------------------------------------------
   //  Reduces this rational number by dividing both the numerator
   //  and the denominator by their greatest common divisor.
   //-----------------------------------------------------------------
```

listing 4.11 continued

```
   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;
   }
}
```

aggregation

key concept

An aggregate object is made up, in part, 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 lots of other parts. Each of these other parts are separate objects. Therefore we can say that a car is an *aggregation*—it is made up of other objects. Aggregation is sometimes described as a *has-a relationship*. For instance, a car *has a* chassis.

In the software world, an *aggregate object* is any object that has other objects as instance data. For example, an `Account` object contains, among other things, a `String` object that is the name of the account owner. That makes each `Account` object an aggregate object.

Let's consider another example. The program `StudentBody` shown in Listing 4.12 creates two `Student` objects. Each `Student` object is made up of two `Address` objects, one for the student's college address and another for the student's home address. The `main` method just creates these objects

listing 4.12

```java
//********************************************************************
//  StudentBody.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of an aggregate class.
//********************************************************************

public class StudentBody
{
   //-----------------------------------------------------------------
   //  Creates some Address and Student objects and prints them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Address school = new Address ("800 Lancaster Ave.", "Villanova",
                                    "PA", 19085);

      Address jHome = new Address ("21 Jump Street", "Lynchburg",
                                   "VA", 24551);
      Student john = new Student ("John", "Gomez", jHome, school);

      Address mHome = new Address ("123 Main Street", "Euclid", "OH",
                                   44132);
      Student marsha = new Student ("Marsha", "Jones", mHome, school);

      System.out.println (john);
      System.out.println ();
      System.out.println (marsha);
   }
}
```

output

```
John Gomez
Home Address:
21 Jump Street
Lynchburg, VA  24551
School Address:
800 Lancaster Ave.
Villanova, PA  19085

Marsha Jones
Home Address:
123 Main Street
Euclid, OH  44132
School Address:
800 Lancaster Ave.
Villanova, PA  19085
```

and prints them out. Note that we once again pass objects to the `println` method, relying on the automatic call to the `toString` method to create something suitable for printing.

The `Student` class shown in Listing 4.13 represents a single college student. This class would have to be huge if it were to represent all aspects of a student. We're keeping it simple for now so that the object aggregation is clear. The instance data of the `Student` class includes two references to

listing 4.13

```java
//********************************************************************
//  Student.java       Author: Lewis/Loftus/Cocking
//
//  Represents a college student.
//********************************************************************

public class Student
{
   private String firstName, lastName;
   private Address homeAddress, schoolAddress;

   //-----------------------------------------------------------------
   //  Sets up this Student object with the specified initial values.
   //-----------------------------------------------------------------
   public Student (String first, String last, Address home,
                   Address school)
   {
      firstName = first;
      lastName = last;
      homeAddress = home;
      schoolAddress = school;
   }

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

      result = firstName + " " + lastName + "\n";
      result += "Home Address:\n" + homeAddress + "\n";
      result += "School Address:\n" + schoolAddress;

      return result;
   }
}
```

Address objects: homeAddress and schoolAddress. We refer to those objects in the toString method when we create a string representation of the student. Because we are concatenating an Address object to another string, the toString method in Address is automatically called.

The Address class is shown in Listing 4.14. It contains only the parts of a street address. Note that nothing about the Address class indicates that it is part of a Student object. The Address class is general so it could be used in any situation in which a street address is needed.

listing
4.14

```java
//********************************************************************
//  Address.java       Author: Lewis/Loftus/Cocking
//
//  Represents a street address.
//********************************************************************

public class Address
{
   private String streetAddress, city, state;
   private int zipCode;

   //-----------------------------------------------------------------
   //  Sets up this Address object with the specified data.
   //-----------------------------------------------------------------
   public Address (String street, String town, String st, int zip)
   {
      streetAddress = street;
      city = town;
      state = st;
      zipCode = zip;
   }

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

      result = streetAddress + "\n";
      result += city + ", " + state + "  " + zipCode;

      return result;
   }
}
```

AP* case study

To work with the AP* Case Study section for this chapter, go to www.aw.com/cssupport and look under author: Lewis/Loftus/Cocking.

4.6 applet methods

key concept: Several methods of the `Applet` class are designed to facilitate their execution in a Web browser.

In Chapter 3 we used the `paint` method to draw the contents of an applet on a screen. An applet has several other methods, too. Figure 4.8 lists several applet methods.

The `init` method is executed once when the applet is first loaded, such as when the browser or appletviewer first views the applet. So the `init` method is the place to initialize the applet's environment and permanent data.

The `start` and `stop` methods of an applet are called when the applet becomes active or inactive, respectively. For example, after we use a browser

```
public void init ()
```
Initializes the applet. Called just after the applet is loaded.

```
public void start ()
```
Starts the applet. Called just after the applet is made active.

```
public void stop ()
```
Stops the applet. Called just after the applet is made inactive.

```
public void destroy ()
```
Destroys the applet. Called when the browser is exited.

```
public URL getCodeBase ()
```
Returns the URL at which this applet's bytecode is located.

```
public URL getDocumentBase ()
```
Returns the URL at which the HTML document containing this applet is located.

```
public AudioClip getAudioClip (URL url, String name)
```
Retrieves an audio clip from the specified URL.

```
public Image getImage (URL url, String name)
```
Retrieves an image from the specified URL.

figure 4.8 Some methods of the `Applet` class to load an applet, the applet's `start` method is called. When we leave that page the applet becomes inactive and the `stop` method is called. If we return to the page, the applet becomes active again and the `start` method is called again. Note that the `init` method is called once when the applet is loaded, but `start` may be called several times as the page is revisited. It is good practice to use `start` and `stop` if an applet actively uses CPU time, such as when it is showing an animation, so that CPU time is not wasted.

Reloading the Web page in the browser does not necessarily reload the applet. To force the applet to reload, most browsers need the user to press a key combination. For example, in Netscape Navigator, the user can hold down the shift key while pressing the reload button to reload the Web page and reload (and reinitialize) all applets linked to that page.

The `getCodeBase` and `getDocumentBase` methods determine where the applet's bytecode or HTML document resides. An applet could use the URL to get more resources, such as an image or audio clip, using the applet methods `getImage` or `getAudioClip`.

We use these applet methods throughout this book.

4.7 graphical objects

Often an object has a graphical representation. Consider the `LineUp` applet shown in Listing 4.15. It creates several `StickFigure` objects, of different colors and heights. The `StickFigure` objects are instantiated in the `init` method of the applet, so they are created only once, when the applet is first loaded.

The `paint` method asks that the stick figures redraw themselves whenever the method is called. The `paint` method is called whenever something happens that might change the graphic representation of the applet itself. For instance, when the window that the applet is displayed in is moved, `paint` redraws the applet contents.

The `StickFigure` class is shown in Listing 4.16. Like any other object, a `StickFigure` object contains data that defines its state, such as the position, color, and height of the figure. The `draw` method contains the individual commands that draw the figure itself.

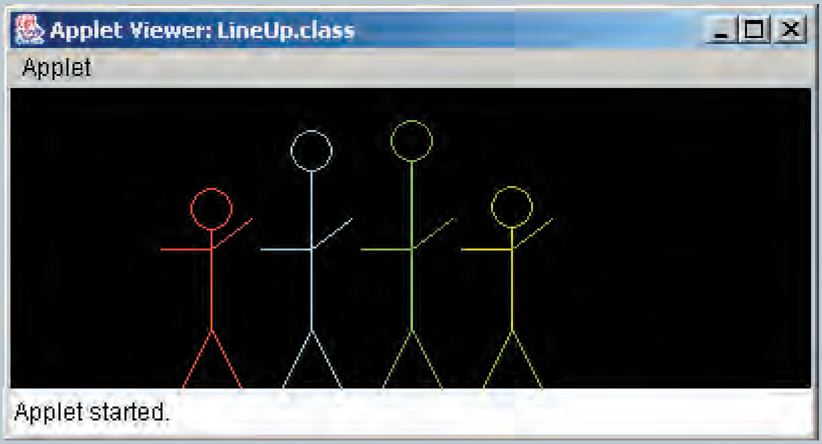
listing 4.15

```java
//********************************************************************
//  LineUp.java       Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of a graphical object.
//********************************************************************

import java.util.Random;
import java.applet.Applet;
import java.awt.*;

public class LineUp extends Applet
{
   private final int APPLET_WIDTH = 400;
   private final int APPLET_HEIGHT = 150;
   private final int HEIGHT_MIN = 100;
   private final int VARIANCE = 40;

   private StickFigure figure1, figure2, figure3, figure4;

   //-----------------------------------------------------------------
   //  Creates several stick figures with varying characteristics.
   //-----------------------------------------------------------------
   public void init ()
   {
      int h1, h2, h3, h4;  // heights of stick figures
      Random generator = new Random();

      h1 = HEIGHT_MIN + generator.nextInt(VARIANCE);
      h2 = HEIGHT_MIN + generator.nextInt(VARIANCE);
      h3 = HEIGHT_MIN + generator.nextInt(VARIANCE);
      h4 = HEIGHT_MIN + generator.nextInt(VARIANCE);

      figure1 = new StickFigure (100, 150, Color.red, h1);
      figure2 = new StickFigure (150, 150, Color.cyan, h2);
      figure3 = new StickFigure (200, 150, Color.green, h3);
      figure4 = new StickFigure (250, 150, Color.yellow, h4);

      setBackground (Color.black);
      setSize (APPLET_WIDTH, APPLET_HEIGHT);
   }
```

listing 4.15 continued

```
   //-----------------------------------------------------------------
   //  Paints the stick figures on the applet.
   //-----------------------------------------------------------------
   public void paint (Graphics page)
   {
      figure1.draw (page);
      figure2.draw (page);
      figure3.draw (page);
      figure4.draw (page);
   }
}
```

display



listing
4.16

```java
//********************************************************************
//  StickFigure.java       Author: Lewis/Loftus/Cocking
//
//  Represents a graphical stick figure.
//********************************************************************

import java.awt.*;

public class StickFigure
{
   private int baseX;     // center of figure
   private int baseY;     // floor (bottom of feet)
   private Color color;   // color of stick figure
   private int height;    // height of stick figure

   //-----------------------------------------------------------------
   //  Sets up the stick figure's primary attributes.
   //-----------------------------------------------------------------
   public StickFigure (int center, int bottom, Color shade, int size)
   {
      baseX = center;
      baseY = bottom;
      color = shade;
      height = size;
   }

   //-----------------------------------------------------------------
   //  Draws this figure relative to baseX, baseY, and height.
   //-----------------------------------------------------------------
   public void draw (Graphics page)
   {
      int top = baseY - height;  // top of head

      page.setColor (color);

      page.drawOval (baseX-10, top, 20, 20);  // head

      page.drawLine (baseX, top+20, baseX, baseY-30);  // trunk

      page.drawLine (baseX, baseY-30, baseX-15, baseY);  // legs
      page.drawLine (baseX, baseY-30, baseX+15, baseY);

      page.drawLine (baseX, baseY-70, baseX-25, baseY-70);  // arms
      page.drawLine (baseX, baseY-70, baseX+20, baseY-85);
   }
}
```

summary of key concepts

- Each object has a state and a set of behaviors. The values of an object's variables define its state. The methods to which an object responds define its behaviors.
- A class is a blueprint of an object; it saves no memory space for data. Each object has its own data space, thus its own state.
- The scope of a variable determines where it can be referenced and depends on where it is declared.
- Objects should be encapsulated. The rest of a program should interact with an object only through its public methods.
- Instance variables should be declared with private visibility to protect their data.
- A method must return a value that matches the return type in the method header.
- When a method is called, the actual parameters are copied into the formal parameters. The types of parameters must match.
- A constructor cannot have any return type, even `void`.
- A precondition is a condition that should be true when a method is called. A postcondition is a condition that should be true when a method finishes executing.
- A variable declared in a method cannot be used outside of it.
- Most objects contain accessor and mutator methods to allow the client to manage data in a controlled manner.
- The versions of an overloaded method can be told apart by their signatures. The number, type, and order of their parameters must be different.
- A complicated method can be broken up into simpler methods that act as private support methods.
- A method called through one object may take another object of the same class as a parameter.
- An aggregate object is made up of other objects, forming a has-a relationship.
- Several methods of the `Applet` class are designed to work in a Web browser.

self-review questions

4.1 What is the difference between an object and a class?

4.2 What is the scope of a variable?

4.3 Objects should be self-governing. Explain.

4.4 What is a modifier?

4.5 Describe each of the following:

a. public method

b. private method

c. public variable

d. private variable

4.6 What does the `return` statement do?

4.7 Explain the difference between an actual parameter and a formal parameter.

4.8 What are constructors used for? How are they defined?

4.9 How can you tell overloaded methods apart?

4.10 What can you do to avoid long, complex methods?

4.11 Explain how a class can have an association with itself.

4.12 What is an aggregate object?

4.13 What do the `start` and `stop` methods of an applet do?

multiple choice

4.1 Object is to class as

a. circle is to square

b. house is to blueprint

c. blueprint is to house

d. bicycle is to car

e. car is to bicycle

4.2 When a Coin object is passed to the println method,

a. a compile error occurs

b. a runtime error occurs

c. the toString method is called on the object to get the string to print

d. a default string that includes the class name is generated

e. the Coin is flipped and the result printed

4.3 Which of the following should be used for services that an object provides to client code?

a. private variables

b. public variables

c. private methods

d. public methods

e. none of the above

4.4 The values passed to a method when it is called are

a. formal parameters

b. actual parameters

c. primitive values

d. objects

e. return values

4.5 Consider the following code:

```
int cube(int x)
{
   x = 3;
   return x * x * x;
}
```

What is returned by the call cube(2)?

a. 2

b. 3

c. 8

d. 9

e. 27

4.6 Method overloading refers to

a. a method with 10 or more parameters

b. a method with a very large body

c. a method that performs too many tasks and should be divided into support methods

d. more than one method with the same name

e. more than one method with the same numbers and types of parameters

4.7 Consider the following code:

```
int sum(int n)
{
   int total = 0;
   for (int i=1; i <= n; i++)
      _______________________
   return total;
}
```

What statement should go in the body of the for loop so that the sum of the first n integers is returned?

a. `total += i;`

b. `total += 1;`

c. `total += n;`

d. `total = n + 1;`

e. `total += n - 1;`

4.8 What will be printed when the method `printStuff` is called?

```
int calc(double a, double b)
{
   int num = a * b;
   b = a;
   num = a + b;
   return num;
}
void printStuff()
{
   double x = 5.1, y = 6.2;
   System.out.println(calc(x,0) + calc(0,y));
}
```

a. `5.16.2`

b. `11.3`

c. `10.20`

d. `0`

e. There will be a compile error because a `double` cannot be assigned to an `int` without a cast.

4.9 The keyword `void` is placed in front of a method name when it is declared to indicate that

a. the method does not return a value

b. the method is a constructor

c. the method is overloaded

d. the method should be called only within its class

e. the method returns a value of an unknown type

4.10 Consider the following method:

```
double doIt(double x)
{
   while (x > 0)
      x -= 3;
}
```

What is it missing?

a. a declaration of the variable `x`

b. a return statement

c. a body

d. a name

e. a parameter

true/false

4.1 Instance variables that are declared public violate the principle of encapsulation.

4.2 Every method must have a return statement.

4.3 A constructor must have the same name as its class.

4.4 A variable declared in one method may be used in any other method in the same class.

4.5 Overloaded methods have a signature, which includes the number, type, and order of parameters.

4.6 An object may be made up of other objects.

4.7 Only one object may be created from a particular class.

4.8 Methods that provide services to clients should be made private.

4.9 Constructors should always return void.

4.10 Parameters to methods may only be primitive types.

short answer

4.1 Write a method header for a method named `translate` that takes an integer parameter and returns a `double`.

4.2 Write a method header for a method named `find` that takes a `String` and a `double` as parameters and returns an integer.

4.3 Write a method header for a method named `printAnswer` that takes three `double`s as parameters and doesn't return anything.

4.4 Write the body of the method for the following header. The method should return a welcome message that includes the user's name and visitor number. For example, if the parameters were "Joe" and 5, the returned string would be `"Welcome Joe! You are visitor number 5."`

```
String welcomeMessage (String name, int
visitorNum)
```

4.5 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.

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

4.7 Write a method called `sum100` that adds up all the numbers from 1 to 100, inclusive and returns the answer.

4.8 Write a method called `maxOfTwo` that accepts two integer parameters from the user and returns the larger of the two.

4.9 Write a method called `sumRange` that accepts two integer parameters that represent a range such as 50 to 75. 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).

4.10 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 false if it is less than the second.

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

4.12 Write a method called `evenlyDivisible` that accepts two integer parameters and returns true if the first parameter can be evenly divided by the second, or vice versa, and false if it can't be. Return false if either parameter is zero.

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

4.14 Overload the `average` method of Exercise 4.13 so that the method returns the average of three integers.

4.15 Overload the `average` method of Exercise 4.13 to take four integer parameters and return their average.

4.16 Write a method called `multiConcat` that takes a `String` and an integer as parameters. Return a `String` made up of the string parameter concatenated with itself `count` times, where `count` is the integer. 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.

4.17 Overload the `multiConcat` method from Exercise 4.16 so 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"`.

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

4.19 Write a method called `floatEquals` that accepts three floating point values as parameters. The method should return true if the first two parameters are no further apart from each other than the third parameter. For example, `floatEquals (2.453, 2.459, 0.01)` should return true because `2.453` and `2.459` are `0.006` apart from each other and `0.006` is less than `0.01`. *Hint*: See the discussion in Chapter 3 on comparing floating point values for equality.

4.20 Write a method called `reverse` that accepts a `String` parameter and returns a string made up of the characters of the parameter in reverse order. There is a method in the `String` class that performs this operation, but for the sake of this exercise, you should write your own.

4.21 Write a mutator method for `faceValue` in the `Die` class in Listing 4.7. The method should only allow `faceValue` to take on a valid value.

4.22 Write a method called `isIsosceles` that accepts the lengths of the sides of a triangle as its parameters. The method returns true if the triangle is isosceles but not equilateral (meaning that exactly two of the sides have an equal length), and false if all three sides are equal or if none of the sides are equal.

4.23 Write a method called `randomInRange` that accepts two integer parameters representing a range such as 30 to 50. The method should return a random integer in the specified range (inclusive). Return zero if the first parameter is greater than the second.

4.24 Write a method called `randomColor` that creates and returns a random `Color` object. Recall that a `Color` object has three values between 0 and 255, representing the contributions of red, green, and blue (its RGB value).

4.25 Write a method called `drawCircle` that draws a circle based on these parameters: a `Graphics` object through which to draw the circle, two integer values for the (x, y) coordinates of the center of the circle, another integer for the circle's radius, and a `Color` object for the circle's color. The method does not return anything.

4.26 Overload the `drawCircle` method of Exercise 4.24 so that if the `Color` parameter is not provided, the circle's color will be black.

programming projects

4.1 Change the `Account` class so that funds can be moved from one account to another. Think of this as withdrawing money from one account and depositing it into another. Change the `main` method of the `Banking` class to show this new service.

4.2 Change the `Account` class so that it also lets a user open an account with just a name and an account number, and a starting balance of zero. Change the `main` method of the `Banking` class to show this new capability.

4.3 Write an application that rolls a die and displays the result. Let the user pick the number of sides on the die. Use the `Die` class to represent the die in your program.

4.4 Design and implement a class called `PairOfDice`, with two six-sided `Die` objects. Create a driver class called `BoxCars` with a `main` method that rolls a `PairOfDice` object 1000 times, counting the number of box cars (two sixes) that occur.

4.5 Using the `PairOfDice` class from Programming Project 4.4, design and implement a class to play a game called Pig. In this game, the user competes against the computer. On each turn, the player rolls a pair of dice and adds up his or her points. Whoever reaches 100 points first, wins. If a player rolls a 1, he or she loses all points for that round and the dice go to the other player. If a player rolls two 1s in one turn, the player loses all points earned so far in the game and loses control of the dice. The player may voluntarily turn over the dice after each roll. So the player must decide to either roll again (be a pig) and risk losing points, or give up the dice, possibly letting the other player win. Set up the computer player so that it always gives up the dice after getting 20 or more points in a round.

4.6 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 20 random cards.

4.7 Write an application that lets the user add, subtract, multiply, or divide two fractions. Use the `Rational` class in your implementation.

4.8 Change the `Student` class (Listing 4.13) so that each student object also contains the scores for three tests. Provide a constructor that sets all instance values based on parameter values. Overload the constructor so that each test score starts out at zero. Provide a method called `setTestScore` that accepts two parameters: the test number (1 through 3) and the score. Also provide a method called `getTestScore` that accepts the test number and returns the score. Provide a method called `average` that computes and returns the average test score for this student. Modify the `toString` method so that the test scores and average are included in the description of the student. Modify the driver class `main` method to exercise the new `Student` methods.

4.9 Design and implement a class called `Building` that represents a drawing of a building. The parameters to the constructor should be the building's width and height. Each building should be colored black and have a few random windows colored yellow. Create an applet that draws a random skyline of buildings.

4.10 Create a class called `Crayon` that represents one crayon of a particular color and length (height). Design and implement an applet that draws a box of crayons.

AP*-style multiple choice

Questions 4.1–4.3 refer to the following class.

```
public class Point

{
   private int myX;
   private int myY;

   public Point()
   {
      myX = 0;
      myY = 0;
   }

   public Point(int x, int y)
   {
      myX = x;
      myY = y;
   }

   public int getX()
   {
      return myX;
   }

   public int getY()
   {
      return myY;
   }
}
```

4.1 Which of the following statements creates a point with coordinates (0,0)?

I. `p = new Point();`

II. `p = Point(0,0);`

III. `p = (0,0);`

(A) I only

(B) II only

(C) III only

(D) I and II only

(E) II and III only

4.2 Which of the following statements is true regarding the `Point` class?

(A) The class won't compile because there are two methods named `Point`.

(B) Variables `myX` and `myY` can be changed from outside the class.

(C) `Point` objects are immutable.

(D) It's impossible to create `Point` objects with coordinates other than (0,0).

(E) Giving `myX` and `myY` private visibility was a poor design decision.

4.3 Suppose we want to add to the `Point` class a method with the following signature.

```
// Sets the x coordinate of the point to the given value
public void setX(int x)
```

Which statement should be in the body of the method?

(A) `x = myX;`

(B) `myX = x;`

(C) `myX = 0;`

(D) `x = 0;`

(E) `myX = myY;`

4.4 Consider a method that will calculate and return the sales tax on an item. Which method signature would be most appropriate?

(A) `int computeTax(double price)`

(B) `void computeTax(double price)`

(C) `int computeTax()`

(D) `double computeTax(double price)`

(E) `void computeTax()`

Questions 4.5–4.6 refer to the following method.

```
int doSomething(int k, int j)
{
   while (k > j)
      k--;
   if (k < j)
      j = k;
   else
      return j;
   return k;
}
```

4.5 What best characterizes the return value of this method?

(A) Returns `k - j`

(B) Returns `j - k`

(C) Returns the smaller of `j` and `k`

(D) Returns the larger of `j` and `k`

(E) Returns `j` if `k` and `j` are equal, `k` otherwise

4.6 Consider the following code segment.

```
int p = 5;
int q = 6;
doSomething(p, q);
System.out.println(p + " " + q);
```

What is printed?

(A) `5 6`

(B) `5 5`

(C) `6 6`

(D) `p q`

(E) There is a compile error because the return value of `doSomething` is not stored in a variable.

AP*-style free response

4.1 Consider the following incomplete declaration of a `Code` class which represents a code consisting of letters and digits. The actual code is stored internally as a `String` variable, `myCode`. Portions of the code may be hidden by changing the corresponding letter or digit to an X. Hidden portions may later be recovered.

```
public class Code
{
   private String myCode;
   // additional instance variables

   public Code(String code)
   {
      myCode = code;
      // possibly additional statements
   }

   public String getCode()
   {
      return myCode;
   }

   // precondition: p1 >= 0, p1 < p2,
   //               p2 <= myCode.length()
   // Replaces the characters in the code starting at
   // position p1 until position p2-1 inclusive with an X
   public void hide(int p1, int p2)
   {
      // to be implemented
   }

   // precondition: p1 >= 0, p1 < p2,
   //               p2 <= myCode.length()
   // Restores to their original values the characters in
   // the code starting at position p1 until position
   // p2-1 inclusive
   public void recover(int p1, int p2)
   {
      // to be implemented
   }
}
```

The methods `hide` and `recover` work as described in the comments. Note that if `hide` is called for a portion of the code that is already hidden, it has no effect and if `recover` is called for a portion of the code that is already "clear," it has no effect.

Suppose the following code is created:

```
Code code = new Code("ABCdef123ghi456jklMNO");
```

The following sequence of method calls results in the instance variable `myCode` having the indicated values.

	Value of `myCode`
`code.hide(2,7);`	`ABXXXXX23ghi456jklMNO`
`code.recover(5,9);`	`ABXXXf123ghi456jklMNO`
`code.hide(3,14);`	`ABXXXXXXXXXXXX6jklMNO`
`code.hide(6,10);`	`ABXXXXXXXXXXXX6jklMNO`
`code.recover(5,6);`	`ABXXXfXXXXXXXX6jklMNO`
`code.recover(0,14);`	`ABCdef123ghi456jklMNO`

a. Thinking ahead to implementing the `hide` and `recover` methods, declare in this part any additional instance variables you will need and any additional code you may need in the constructor.

```
// instance variables
String myCode;



Code(String code)
{
   myCode = code;



}
```

b. Implement the `hide` method.

c. Implement the `recover` method.

answers to self-review questions

4.1 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 saves no memory space for variables. Each object has its own data space and therefore its own state.

4.2 The scope of a variable is where in a program 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.

4.3 A self-governing object controls the values of its own data. Encapsulated objects, which don't allow an external client to reach in and change the data, are self-governing.

4.4 A modifier is a Java reserved word that can be used in the definition of a variable or method and that defines certain characteristics. For example, if a variable has private visibility, it cannot be directly accessed from outside the object.

4.5 The modifiers affect the methods and variables in the following ways:

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

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

c. A public variable is a variable that can be directly read and changed by a client. This violates the principle of encapsulation and should be avoided.

d. A private variable can be read and changed only from within the class. Variables almost always are declared with private visibility.

4.6 An explicit `return` statement spells out the value that is returned from a method. The type of the return value must match the return type in the method definition.

4.7 An actual parameter is a value sent to a method when the method is invoked. A formal parameter is the matching 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.

4.8 Constructors are special methods 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.

4.9 Overloaded methods have a unique signature, which includes the number, order, and type of the parameters. The return type is not part of the signature.

4.10 Dividing a complex method into several support methods simplifies the design of the program.

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

4.12 An aggregate object has other objects as instance data. That is, an aggregate object is made up of other objects.

4.13 The `Applet start` method is called automatically every time the applet becomes active, such as when a browser opens the page it is on. The `stop` method is called automatically when the applet becomes inactive.