4 writing classes

In Chapters 2 and 3 we used objects and classes for the various services they provide. We also explored several fundamental programming statements. With that experience as a foundation, 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 serve as blueprints for new objects, composed of variables and methods.
- Explain the advantages of encapsulation and the use of Java modifiers to accomplish it.
- Explore the details of method declarations.
- Revisit the concepts of method invocation and parameter passing.
- Explain and use method overloading to create versatile classes.
- Demonstrate the usefulness of method decomposition.
- Describe various relationships between objects.
- Create graphics-based objects.

4.0 objects revisited

Throughout Chapters 2 and 3 we created objects from classes in the Java standard class library in order to use the particular services they provide. We didn't need to know the details of how the classes did their jobs; we simply trusted them to do so. That, as we have discussed previously, is one of the advantages of abstraction. Now, however, we are ready to turn our attention to writing our own classes.

First, let's revisit 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 particular characteristics such as its diameter, color, and elasticity. Formally, 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 activities define the object's *behavior*.

All objects have a state and a set of behaviors. We can represent these characteristics in software objects as well. The values of an object's variables describe the object's state, and the methods that can be invoked using the object 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 to which an object responds 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 establish the state and behavior that are relevant to the ball's use in the computerized ball game.

Each object has its own state. Each ball object has a particular location, for instance, which typically is different from the location of all other balls.. Behaviors, though, tend to apply to all objects of a particular type. For instance, in general, 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. The action is the same, but the specific result depends on that particular object's state. An object's behavior often modifies 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 consider another example. In software that is used to manage a university, a student could be represented as an object. The collection of all such objects represents the entire student body at the university. 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, major, 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 tangible items, 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 the process of issuing (printing) the error. A common mistake made by new programmers to the world of object-orientation is to limit the possibilities to tangible entities.

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 depicted in Fig. 4.1.

In one sense, 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 and other crucial characteristics are the same in each. To create a different house, we would need a different blueprint.

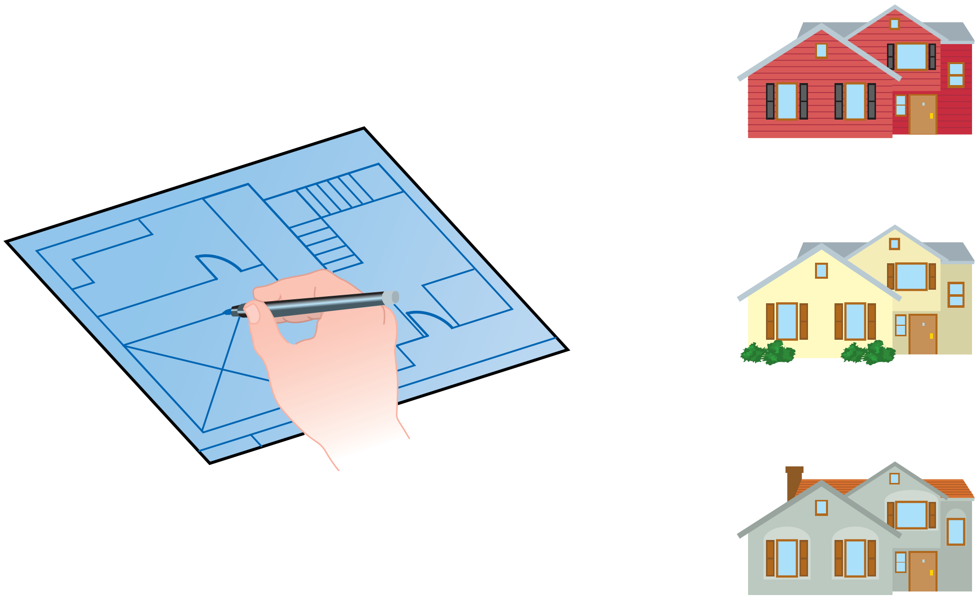


figure 4.1 A house blueprint and three houses created from it

key concept

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

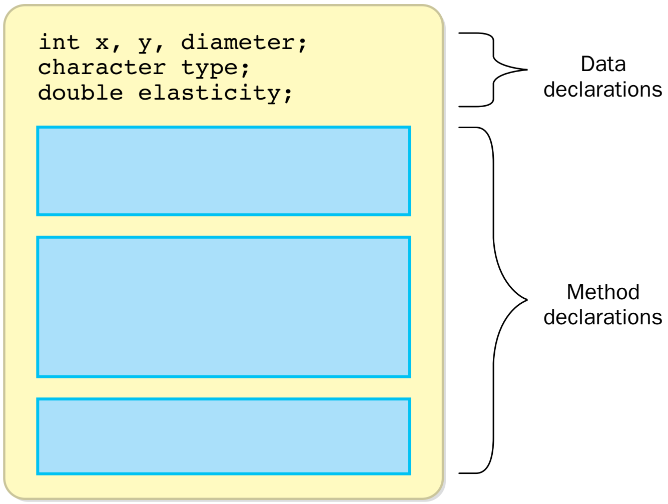
A class is a blueprint of an object. However, a class is not an object any more than a blueprint is a house. In general, no space to store data values is reserved in a class. To allocate space to store data values, we must instantiate one or more objects from the class. (We discuss the exception to this 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 instatntiated object and the declarations of the methods that can be invoked using an object. Collectively these are called the *members* of the class, as shown in Fig. 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 result of "heads" or "tails." The `CountFlips` program simulates the flipping of a coin 500 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 composed of data values (variables



```
int x, y, diameter;
character type;
double elasticity;
```

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

listing 4.1

```
//********************************************************************
//  CountFlips.java       Author: Lewis/Loftus
//
//  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
```

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 composed of the `Coin` constructor and three regular methods: `flip`, `isHeads`, and `toString`.

You will recall from Chapter 2 that constructors are special methods that have the same name as the class. The `Coin` constructor gets called when the `new` opera-

listing 4.2

```
//********************************************************************
//  Coin.java       Author: Lewis/Loftus
//
//  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.
   //-----------------------------------------------------------------
```

listing 4.2 continued

```
   public boolean isHeads ()
   {
      return (face == HEADS);
   }

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

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

      return faceName;
   }
}
```

tor 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 crucial for anyone trying to understand the software, it also separates the code visually so that it's easy to jump visually from one method to the next while reading the code. The definitions of these methods have various parts, and we'll dissect them in later sections of this chapter.

Figure 4.3 lists the services defined in the `Coin` class. From this point of view, it looks no different from any other class that we've used in previous examples. The only important difference is that the `Coin` class was not provided for us by the Java standard class library. We wrote it ourselves.

For the examples in this book, we generally store each class in its own file. Java allows multiple classes to be stored in one file. If a file contains multiple classes, only one of those classes can be declared using the reserved word `public`. Furthermore, the name of the public class must correspond to the name of the file. For instance, class `Coin` is stored in a file called `Coin.java`.

```
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

instance data

Note that 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 within a program in which that variable can be referenced. By being 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 instance.

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 particular class, a default version is called that returns a string 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 same class, `Coin`, 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

listing 4.3

```java
//**********************************************************************
//  FlipRace.java       Author: Lewis/Loftus
//
//  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
         count1 = (coin1.isHeads()) ? count1+1 : 0;
         count2 = (coin2.isHeads()) ? count2+1 : 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!
```

to look to the future to try to give the class behaviors that may be beneficial in other programs, not just fit the specific purpose for which you are creating it 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, despite the fact that the language performs this automatic initialization, it is good practice to initialize variables explicitly (usually in a constructor) so that anyone reading the code will clearly understand the intent.

UML diagrams

Throughout this book, we use *UML diagrams* to visualize relationships among classes and objects. UML stands for the *Unified Modeling Language*. Several types of UML diagrams exist, each designed to show specific aspects of object-oriented program design.

A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes, and methods. Figure 4.4 depicts an example showing classes of the `FlipRace` program. Depending on the goal of the diagram, the attribute and/or method sections can be left out of any class.

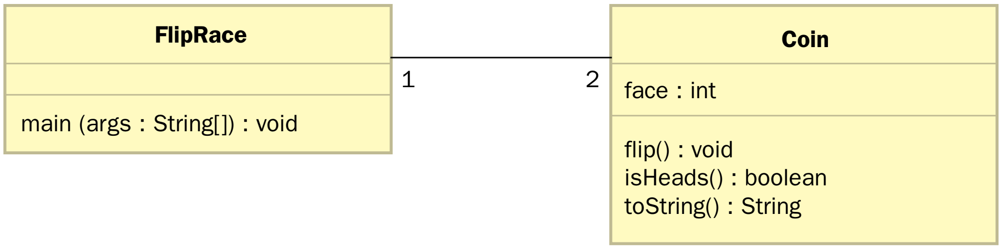


figure 4.4 A UML class diagram showing the classes involved in the FlipRace program

The line connecting the FlipRace and Coin classes in Fig. 4.4 indicates that a relationship exists between the classes. This simple line represents a basic *association*, meaning that the classes are generally aware of each other within the program; that one may refer to and make use of the other. An association can show *multiplicity*, as this one does by annotating the connection with numeric values. In this case, it indicates that FlipRace is associated with exactly two Coin objects.

UML diagrams always show the type of an attribute, parameter, and the return value of a method after the attribute name, parameter name, or method header (separated by a colon). This may seem somewhat backward given that types in Java are generally shown before the entity of that type. We must keep in mind that UML is not designed specifically for Java programmers. It is intended to be language independent. UML has become the most popular notation in the world for the design of object-oriented software.

A UML *object diagram* consists of one or more instantiated objects. An object diagram is a snapshot of the objects at a given point in the executing program. For example, Fig. 4.5 shows the two Coin objects of the FlipRace program.

The notation for an object is similar to that for a class. However, the contents of the first section are underlined and often include the name of a specific object in addition to the class name. Another important difference between the notation of a class and an object is that the attributes shown in the second section of an object are shown with their current value. Because objects of the same class respond to the same methods, the third section is often left out.

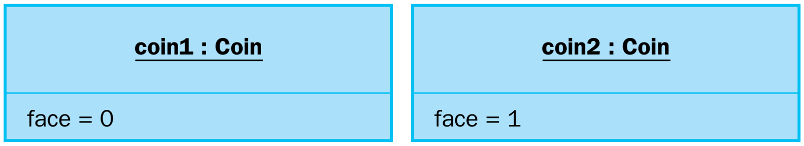


figure 4.5 A UML object diagram showing the `Coin` objects of the `FlipRace` program

We should keep in mind that UML notation is not intended to describe a program after it is written. It's primarily a language-independent mechanism for visualizing and capturing the design of a program before it is written.

As we develop larger programs consisting of multiple classes and objects, UML diagrams will help us visualize them. UML diagrams have additional notations that represent various other program entities and relationships. We will explore new aspects of UML diagrams as the situation dictates.

key concept

A UML diagram is a software design tool that helps us visualize the classes and objects of a program and the relationships among them.

encapsulation and visibility modifiers

We can think about an object in one of two ways. The view we take depends on what we are trying to accomplish at the moment. First, when we are designing and implementing an object, we need to think about the details of 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 in terms of how the objects in the program interact. 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 allows us to focus on the larger 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 modified 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 solely responsible for changing the value of the face variable. We should make it difficult, if not 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 characteristic is encapsulation. An object should be encapsulated from the rest of the system. It should interact with other parts of a program only through the specific set of methods that define the services that that object provides. 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 interact with an object only through a well-defined interface.

Encapsulation is depicted graphically in Fig. 4.6. The code that uses an object, sometimes called the *client* of an object, should not be allowed to access variables directly. The client should interact with the object's methods, and those methods then interact with the data encapsulated within 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) access the face variable directly.

In Java, we accomplish object encapsulation using *modifiers*. A modifier is a Java reserved word that is used to specify particular 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 various ways. Some modifiers can be used together, but some combinations are invalid. We discuss various Java modifiers at appropriate points throughout this book, and all of them are summarized in Appendix F.

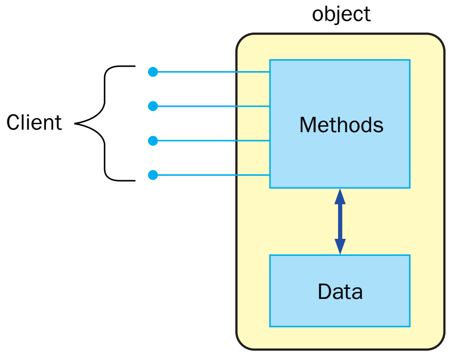


figure 4.6 A client interacting with the methods of an object

Some Java modifiers are called *visibility modifiers* because they control access to the members of a class. 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 directly referenced from outside of the object. If a member of a class has *private visibility*, it can be used anywhere inside the class definition but cannot be referenced externally. A third visibility modifier, `protected`, is relevant only in the context of inheritance. We discuss it in Chapter 7.

Public variables violate encapsulation. They allow code external to the class in which the data is defined to reach in and access or modify the value of the data. Therefore instance data should be defined with private visibility. Data that is declared as `private` can be accessed only by the methods of the class, which makes the objects created from that class self-governing. The visibility we apply to a method depends on the purpose of that method. Methods that provide services to the client of the class must be declared with public visibility so that they can be invoked by the client. These methods are sometimes referred to as *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 referred to as *support methods*. We discuss an example that makes use of several support methods later in this chapter.

key concept

Instance variables should be declared with private visibility to promote encapsulation.

The table in Fig. 4.7 summarizes the effects of public and private visibility on both variables and methods.

	public	**private**
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

figure 4.7 The effects of public and private visibility

Note that a client can still access or modify private data by invoking service methods that change the data. For example, although the main method of the FlipRace class cannot directly access the face variable, it can invoke the flip service method, which sets the value of face. A class must provide service methods for valid client operations. The code of those methods must be carefully designed to permit 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 because they were 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, the encapsulation issue is largely moot. 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 without violating the principle of encapsulation.

UML diagrams reflect the visibility of a class member with special notations. A member with public visibility is preceded by a plus sign (+), and a member with private visibility is preceded by a minus sign (-). We'll see these notations used in the next example.

4.2 anatomy of a method

We've seen that a class is composed of data declarations and method declarations. Let's examine 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* specifies the code that is executed when the method is invoked.

When a method is called, the flow of control transfers to that method. One by one, 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 invoked 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 invoke it. If it is part of a different object, it is invoked through that object's name, as we've seen many times. Figure 4.8 presents this process.

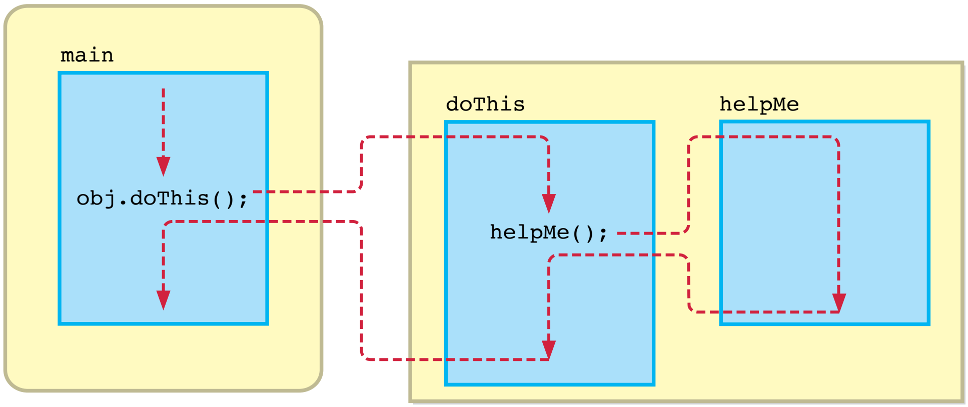
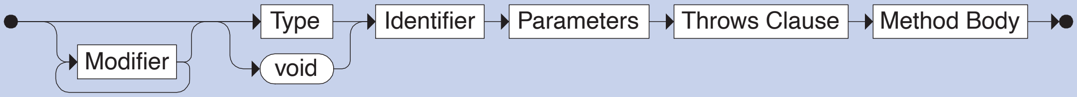
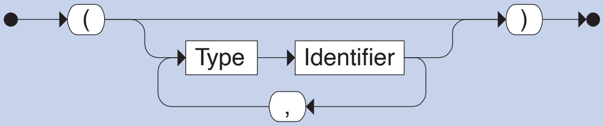


figure 4.8 The flow of control following method invocations

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 invoked. The Throws Clause is optional and indicates the exceptions that may be thrown by this method.

Example:

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

We've defined the main method of a program many times in previous examples. Its definition 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 delimited by braces.

Let's look at another example as we explore the details of method declarations. The Banking class shown in Listing 4.4 contains a main method that creates a few Account objects and invokes their services. The Banking program doesn't really do anything useful except demonstrate how to interact with Account objects. Such programs are often called *driver programs* because all they do is drive the use of other, more interesting parts of our program. They are often used for testing purposes.

The Account class represents a basic bank account and is shown in Listing 4.5. It contains data values important to the management of a bank account: the account number, the balance, and the name of the account's owner. The interest rate is stored as a constant.

Figure 4.9 shows an object diagram for the Banking program. Note the use of the minus signs in front of the attributes to indicate that they have private visibility.

The methods of the Account class perform various services on a bank account, such as making deposits and withdrawals. Checks are made to ensure that the data used for the services are valid, such as preventing the withdrawal of a negative amount (which would essentially be a deposit). We explore the methods of the Account class in detail in the following sections.

the return statement

The return type specified 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.

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

key concept

A method must return a value consistent with the return type specified in the method header.

listing 4.4

```
//********************************************************************
//  Banking.java       Author: Lewis/Loftus
//
//  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 ("Jane Smith", 69713, 40.00);
      Account acct3 = new Account ("Edward Demsey", 93757, 759.32);

      acct1.deposit (25.85);

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

      System.out.println ("Smith 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

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

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

72354   Ted Murphy      $132.90
69713   Jane Smith      $111.52
93757   Edward Demsey   $785.90
```

listing 4.5

```java
//********************************************************************
//  Account.java       Author: Lewis/Loftus
//
//  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 long acctNumber;
   private double balance;
   private String name;

   //-----------------------------------------------------------------
   //  Sets up the account by defining its owner, account number,
   //  and initial balance.
   //-----------------------------------------------------------------
   public Account (String owner, long account, double initial)
```

listing 4.5 continued

```
{
   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;
}

//-----------------------------------------------------------------
//  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.");
```

listing 4.5 continued

```
            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;
   }

   //-----------------------------------------------------------------
   //  Returns the account number.
   //-----------------------------------------------------------------
   public long 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));
   }
}
```

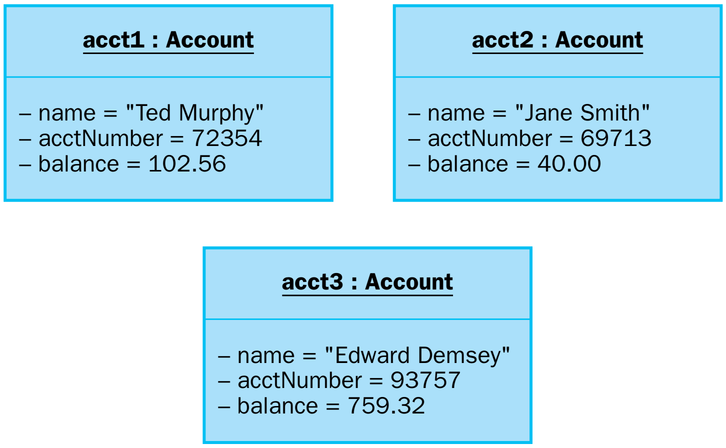
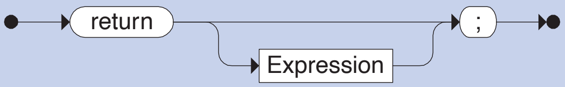


figure 4.9 A UML object diagram showing the objects of the `Banking` program

A method that does not return a value does not usually contain a `return` statement. The method automatically returns to the calling method when the end of the method is reached. A method with a `void` return type may, however, contain a `return` statement without an expression.

It is usually not good practice to use more than one `return` statement in a method, even though it is possible to do so. In general, a method should have one `return` statement as the last line of the method body, unless that makes the method overly complex.

Return Statement



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

Examples:

```
return;

return (distance * 4);
```

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

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

parameters

As we defined in Chapter 2, a parameter is a value that is passed into a method when it is invoked. The *parameter list* in the header of a method specifies the types of the values that are passed and the names by which the called method will refer to those values.

The names of the parameters in the header of the method declaration are called *formal parameters*. In an invocation, the values passed into a method are called *actual parameters*. A method invocation and definition always give the parameter list in parentheses after the method name. If there are no parameters, an empty set of parentheses is used.

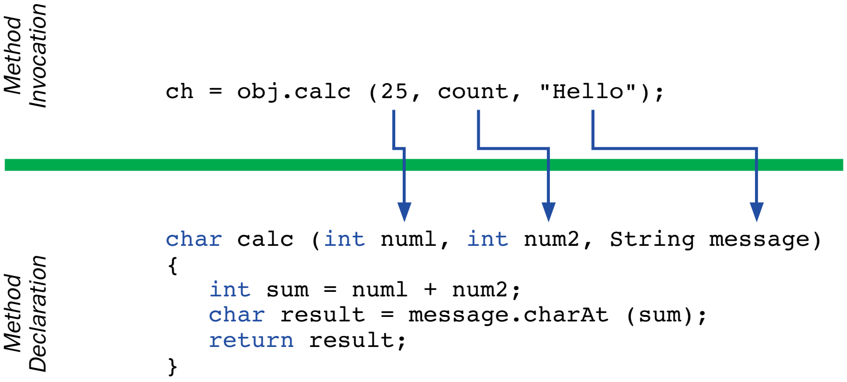
The formal parameters are identifiers that serve as variables inside the method and whose initial values come from the actual parameters in the invocation. Sometimes they are called automatic variables. When a method is called, the value in each actual parameter is copied and stored in the corresponding 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 passed as the parameter.

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 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 Fig. 4.10. The types of the actual parameters must be consistent with the specified types of the formal parameters.

The `deposit` method of the `Account` class, for instance, 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. In the case of the `withdraw` method, two parameters of type `double` are expected. The types and number of parameters must be consistent or the compiler will issue an error message.



```
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 4.10 Passing parameters from the method invocation to the declaration

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

constructors

As we stated in Chapter 2, a constructor is similar to a method that is invoked 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 differs 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 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. As such, it cannot be invoked as a constructor. This leads to error messages that are sometimes difficult to decipher.

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

A constructor is generally used to initialize the newly instantiated object. For instance, the constructor of the `Coin` class calls the `flip` method initially to determine the face value of the coin. The constructor of the `Account` class explicitly 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

As we described earlier in this chapter, the scope of a variable (or constant) is the part of a program in which 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 in which 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 the compiler to issue 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.

key concept

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

Because local data and instance data operate at 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. Referring to that name in the method will reference the local version of the variable. This naming practice obviously has the potential to 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 cease to exist when the method is exited. 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.

4.3 method overloading

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

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

key concept

The versions of an overloaded method are distinguished by their signature, which is the number, type, and order of the parameters.

The compiler must still be able to associate each invocation to a specific method declaration. If the method name for two or more methods is the same, then additional information is used to uniquely identify the version that is being invoked. In Java, a method name can be used for multiple methods as long as the number of parameters, the types of those parameters, and/or the order of the types of parameters is distinct. A method's name along with the number, type, and order of its parameters is called the method's *signature*. The compiler uses the complete method signature to *bind* a method invocation to the appropriate definition.

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

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

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

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

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

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

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

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

```
System.out.println ("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 invoked.

Constructors are a primary candidate for overloading. By providing multiple versions of a constructor, we provide 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.

4.4 method decomposition

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

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

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

listing 4.6

```
//********************************************************************
//  SnakeEyes.java       Author: Lewis/Loftus
//
//  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: " + (float)snakeEyes/ROLLS);
   }
}
```

output

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

listing
4.7

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

public class Die
{
   private final int MIN_FACES = 4;

   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 = (int) (Math.random() * numFaces) + 1;
      return faceValue;
   }
```

listing 4.7 continued

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

listing 4.8

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

import cs1.Keyboard;

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

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

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

listing 4.8 continued

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

output

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

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

Translate another sentence (y/n)? y

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

That sentence in Pig Latin is:
ayplay ityay againyay amsay

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

The workhorse behind the PigLatin program is the PigLatinTranslator class, shown in Listing 4.9. An object of type PigLatinTranslator provides one fundamental service, 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.

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

The translate method uses a StringTokenizer object to separate the string into words. Recall that the primary role of the StringTokenizer class (discussed in Chapter 3) is to separate a string into smaller elements called tokens. In this case, the tokens are separated by space characters so we can use the default white

listing
4.9

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

import java.util.StringTokenizer;

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

      sentence = sentence.toLowerCase();
      StringTokenizer tokenizer = new StringTokenizer (sentence);

      while (tokenizer.hasMoreTokens())
      {
         result += translateWord (tokenizer.nextToken());
         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
```

listing 4.9 continued

```
         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);
   }

   //-----------------------------------------------------------------
   //  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") );
   }
}
```

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 somewhat involved, so the translateWord method makes use of two other private methods, beginsWithVowel and beginsWithBlend.

The beginsWithVowel method returns a boolean value that indicates whether the word passed as a parameter begins with a vowel. Note that instead of checking each vowel separately, the code for this method declares a string that contains all of the vowels, and then invokes the String method indexOf to determine whether the first character of the word is in the vowel string. If the specified character cannot be found, the indexOf method returns a value of –1.

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

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

key concept

A complex service provided by an object can be decomposed and can make use of private support methods.

Figure 4.11 shows a UML class diagram for the PigLatin program. Note the notation showing the visibility of various methods.

Whenever a method becomes large or complex, we should consider decomposing it into multiple methods to create a more understandable class design.

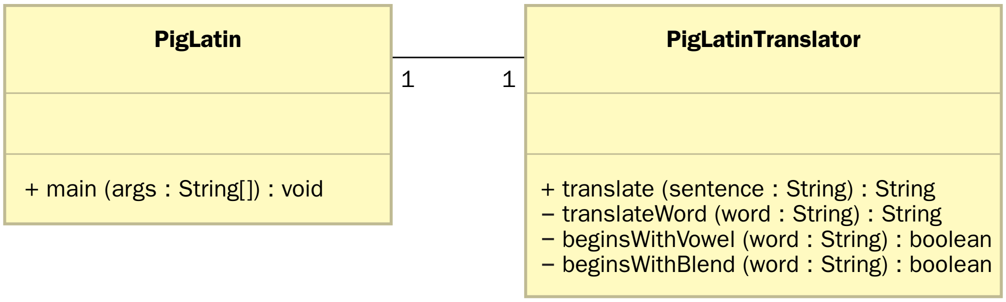

PigLatin
+ main (args : String[]) : void
1
1
PigLatinTranslator
+ translate (sentence : String) : String
– translateWord (word : String) : String
– beginsWithVowel (word : String) : boolean
– beginsWithBlend (word : String) : boolean

figure 4.11 A UML class diagram for the PigLatin program

First, however, we must consider how other classes and objects can be defined to create better overall system design. In an object-oriented design, method decomposition must be subordinate to object decomposition.

4.5 object relationships

Classes, and their associated objects, can have particular types of relationships to each other. This section revisits the idea of the general association and then extends that concept to include associations between objects of the same class. We then explore aggregation, in which one object is composed 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

In previous examples of UML diagrams, we've seen the idea of two classes having a general association. This means that those classes are "aware" of each other. Objects of those classes may use each other for the specific services that each provides. This sometimes is referred to as a *use relationship*.

An association could be described in general terms, such as the fact that an `Author` object *writes* a `Book` object. The association connections between two classes in a UML diagram can be annotated with such comments, if desired. These kinds of annotations are called *adornments*.

As we've seen, associations can have a multiplicity associated with them. They don't always have to show specific values, however. The asterisk could be used to indicate a general zero-or-more value. Ranges of values could be given if appropriate, such as 1...5.

An association relationship is intended to be very general and therefore very versatile. We introduce additional uses of general associations throughout the book as appropriate.

web bonus

You can find additional discussion and examples of UML notation on the book's Web site.

association between objects of the same class

key concept

A method invoked through one object may take as a parameter another object 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.

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

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

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

The RationalNumbers program shown in Listing 4.10 demonstrates a similar situation. Recall that a rational number is a value that can be represented as a ratio of two integers (a fraction). The RationalNumbers program creates two objects representing rational numbers and then performs various operations on them to produce new rational numbers.

listing 4.10

```
//********************************************************************
//  RationalNumbers.java       Author: Lewis/Loftus
//
//  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.
r1 + r2: 13/12
r1 - r2: 5/12
r1 * r2: 1/4
r1 / r2: 9/4
```

The `Rational` class is shown in Listing 4.11. Each object of type `Rational` represents one rational number. The `Rational` class contains various 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, including `reduce` and `gcd`, are declared with private visibility. These methods are `private` because we don't want them executed directly from outside a `Rational` object. They exist only to support the other services of the object.

listing 4.11

```
//********************************************************************
//  Rational.java       Author: Lewis/Loftus
//
//  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.
   //-----------------------------------------------------------------
```

listing 4.11 continued

```java
   public int getDenominator ()
   {
      return denominator;
   }

   //-----------------------------------------------------------------
   //  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);
   }
```

listing 4.11 continued

```
   //-----------------------------------------------------------------
   //  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();

      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)
```

listing 4.11 continued

```
            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.
   //-----------------------------------------------------------------
   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 composed, 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 several other parts. Each of these other parts could be considered separate objects. Therefore we can say that a car is an *aggregation*—it is composed, at least in part, of other objects. Aggregation is sometimes described as a *has-a relationship*. For instance, a car has a chassis.

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

Let's consider another example. The program StudentBody shown in Listing 4.12 creates two Student objects. Each Student object is composed, in part, of two Address objects, one for the student's address at school and another for the student's home address. The main method does nothing more than create these objects and print them out. Note that we once again pass objects to the println method, relying on the automatic call to the toString method to create a valid representation of the object suitable for printing.

listing 4.12

```
//********************************************************************
//  StudentBody.java       Author: Lewis/Loftus
//
//  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", "Smith", jHome, school);
```

listing 4.12 continued

```
      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 Smith
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
```

The `Student` class shown in Listing 4.13 represents a single student. This class would have to be greatly expanded if it were to represent all aspects of a student. We deliberately keep it simple for now so that the object aggregation is clearly shown. The instance data of the `Student` class includes two references to `Address` objects. We refer to those objects in the `toString` method as we create a string representation of the student. By concatenating an `Address` object to another string, the `toString` method in `Address` is automatically invoked.

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 kept generic by design and therefore could be used in any situation in which a street address is needed.

listing 4.13

```
//********************************************************************
//  Student.java       Author: Lewis/Loftus
//
//  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;
   }
}
```

listing 4.14

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

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

   //-----------------------------------------------------------------
   //  Sets up this Address object with the specified data.
   //-----------------------------------------------------------------
   public Address (String street, String town, String st, long 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;
   }
}
```

The more complex an object, the more likely it will need to be represented as an aggregate object. In UML, aggregation is represented by a connection between two classes, with an open diamond at the end near the class that is the aggregate. Figure 4.12 shows a UML class diagram for the `StudentBody` program.

Note that in previous UML diagram examples, strings were not represented as separate classes with aggregation relationships, though technically they could be. Strings are so fundamental to programming that they are usually represented the same way a primitive attribute is represented.

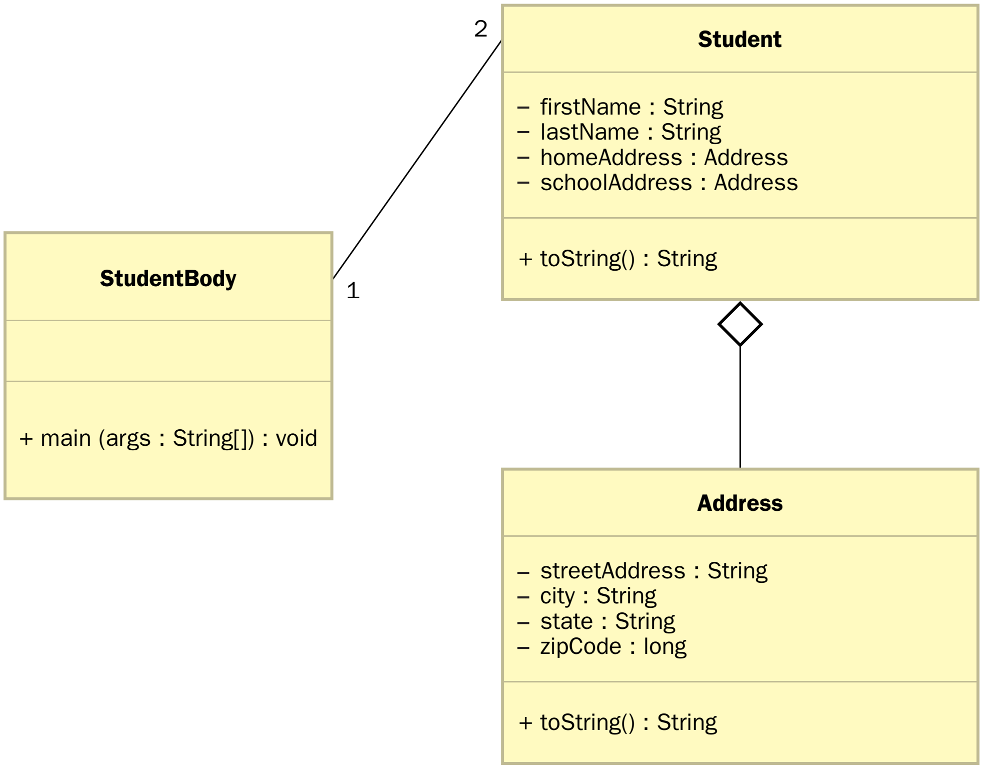


figure 4.12 A UML class diagram showing aggregation

4.6 applet methods

In applets presented in previous chapters, we've seen the use of the `paint` method to draw the contents of the applet on the screen. An applet has several other methods that perform specific duties. Because an applet is designed to work with Web pages, some applet methods are specifically designed with that concept in mind. Figure 4.13 lists several applet methods.

key concept

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

The `init` method is executed once when the applet is first loaded, such as when the browser or appletviewer initially view the applet. Therefore 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 to initially

```
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 4.13 Some methods of the `Applet` class load an applet, the applet's start method is called. We may then leave that page to visit another one, at which point the applet becomes inactive and the stop method is called. If we return to the applet's 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 implement start and stop for an applet if it actively uses CPU time, such as when it is showing an animation, so that CPU time is not wasted on an applet that is not visible.

Note that reloading the Web page in the browser does not necessarily reload the applet. To force the applet to reload, most browsers provide some key combination for that purpose. For example, in Netscape Navigator, holding down the shift key while pressing the reload button with the mouse will not only reload the Web page, it will also reload (and reinitialize) all applets linked to that page.

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

We use the various applet methods as appropriate 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 varying color and random height. The StickFigure objects are instantiated in the init method of the applet, so they are created only once when the applet is initially loaded.

The paint method of LineUp simply requests that the stick figures redraw themselves whenever the method is called. The paint method is called whenever an event occurs that might influence the graphic representation of the applet itself. For instance, when the window that the applet is displayed in is moved, paint is called to redraw 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, relative to the position and height.

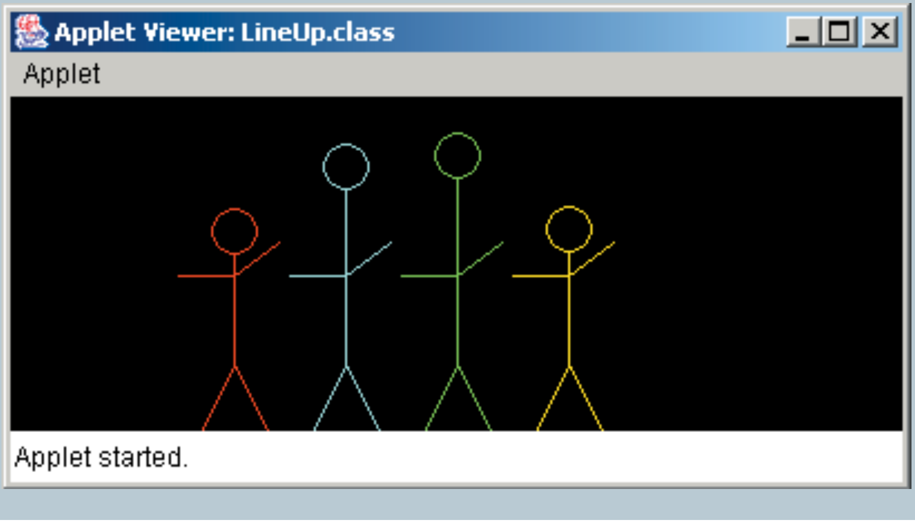
listing 4.15

```
//********************************************************************
//  LineUp.java       Author: Lewis/Loftus
//
//  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



Applet started.

listing
4.16

```
//********************************************************************
//  StickFigure.java       Author: Lewis/Loftus
//
//  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
```

listing 4.16 continued

```
      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 for an object; it reserves no memory space for data. Each object has its own data space, thus its own state.
- The scope of a variable, which determines where it can be referenced, depends on where it is declared.
- A UML diagram is a software design tool that helps us visualize the classes and objects of a program and the relationships among them.
- Objects should be encapsulated. The rest of a program should interact with an object only through a well-defined interface.
- Instance variables should be declared with private visibility to promote encapsulation.
- A method must return a value consistent with the return type specified in the method header.
- When a method is called, the actual parameters are copied into the formal parameters. The types of the corresponding parameters must match.
- A constructor cannot have any return type, even `void`.
- A variable declared in a method is local to that method and cannot be used outside of it.
- The versions of an overloaded method are distinguished by their signatures. The number, type, and order of their parameters must be distinct.
- A complex service provided by an object can be decomposed to can make use of private support methods.
- A method invoked through one object may take as a parameter another object of the same class.
- An aggregate object is composed, in part, of other objects, forming a has-a relationship.
- Several methods of the `Applet` class are designed to facilitate their execution 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 What are UML diagrams designed to do?

4.4 Objects should be self-governing. Explain.

4.5 What is a modifier?

4.6 Describe each of the following:

- public method
- private method
- public variable
- private variable

4.7 What does the `return` statement do?

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

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

4.10 How are overloaded methods distinguished from each other?

4.11 What is method decomposition?

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

4.13 What is an aggregate object?

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

exercises

4.1 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.

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

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

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

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

4.6 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 otherwise.

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

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

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

4.10 Overload the `average` method of Exercise 4.9 such that if three integers are provided as parameters, the method returns the average of all three.

4.11 Overload the `average` method of Exercise 4.9 to accept four integer parameters and return their average.

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

4.13 Overload the `multiConcat` method from Exercise 4.12 such that if the integer parameter is not provided, the method returns the string concatenated with itself. For example, if the parameter is `"test"`, the return value is `"testtest"`.

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

4.15 Write a method called `floatEquals` that accepts three floating point values as parameters. The method should return true if the first two parameters are equal within the tolerance of the third parameter. *Hint*: See the discussion in Chapter 3 on comparing floating point values for equality.

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

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

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

4.19 Write a method called `randomColor` that creates and returns a `Color` object that represents a random color. Recall that a `Color` object can be defined by three integer values between 0 and 255, representing the contributions of red, green, and blue (its RGB value).

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

4.21 Overload the `drawCircle` method of Exercise 4.20 such that if the `Color` parameter is not provided, the circle's color will default to black.

4.22 Overload the `drawCircle` method of Exercise 4.20 such that if the radius is not provided, a random radius in the range 10 to 100 (inclusive) will be used.

4.23 Overload the `drawCircle` method of Exercise 4.20 such that if both the color and the radius of the circle are not provided, the color will default to red and the radius will default to 40.

4.24 Draw a UML class diagram for the `SnakeEyes` program.

4.25 Draw a UML object diagram showing the `Die` objects of the `SnakeEyes` program at a specific point in the program.

4.26 Draw a UML object diagram for the objects of the `StudentBody` program.

4.27 Draw UML class and object diagrams for the `BoxCars` program described in Programming Project 4.3.

4.28 Draw UML class and object diagrams for the `Pig` program described in Programming Project 4.4.

programming projects

4.1 Modify the Account class to provide a service that allows funds to be transferred from one account to another. Note that a transfer can be thought of as withdrawing money from one account and depositing it into another. Modify the main method of the Banking class to demonstrate this new service.

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

4.3 Design and implement a class called PairOfDice, composed of 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.

4.4 Using the PairOfDice class from Programming Project 4.3, design and implement a class to play a game called Pig. In this game, the user competes against the computer. On each turn, the current player rolls a pair of dice and accumulates points. The goal is to reach 100 points before your opponent does. If, on any turn, the player rolls a 1, all points accumulated for that round are forfeited and control of the dice moves to the other player. If the player rolls two 1s in one turn, the player loses all points accumulated thus far in the game and loses control of the dice. The player may voluntarily turn over the dice after each roll. Therefore the player must decide to either roll again (be a pig) and risk losing points, or relinquish control of the dice, possibly allowing the other player to win. Implement the computer player such that it always relinquishes the dice after accumulating 20 or more points in any given round.

4.5 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.

4.6 Modify the Student class presented in this chapter as follows. Each student object should also contain the scores for three tests. Provide a constructor that sets all instance values based on parameter values. Overload the constructor such that each test score is assumed to be initially 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 appropriate score. Provide a method called average that computes and returns the average test score for this student. Modify the `toString` method such 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.

4.7 Design and implement a class called `Course` that represents a course taken at a school. A course object should keep track of up to five students, as represented by the modified `Student` class from the previous programming project. The constructor of the `Course` class should accept only the name of the course. Provide a method called `addStudent` that accepts one `Student` parameter (the `Course` object should keep track of how many valid students have been added to the course). Provide a method called `average` that computes and returns the average of all students' test score averages. Provide a method called `roll` that prints all students in the course. Create a driver class with a `main` method that creates a course, adds several students, prints a roll, and prints the overall course test average.

4.8 Design and implement a class called `Building` that represents a graphical depiction of a building. Allow the parameters to the constructor to specify the building's width and height. Each building should be colored black, and contain a few random windows of yellow. Create an applet that draws a random skyline of buildings.

4.9 A programming project in Chapter 3 describes an applet that draws a quilt with a repeating pattern. Design and implement an applet that draws a quilt using a separate class called `Pattern` that represents a particular pattern. Allow the constructor of the Pattern class to vary some characteristics of the pattern, such as its color scheme. Instantiate two separate Pattern objects and incorporate them in a checkerboard layout in the quilt.

4.10 Write an applet that displays a graphical seating chart for a dinner party. Create a class called `Diner` (as in one who dines) that stores the person's name, gender, and location at the dinner table. A diner is graphically represented as a circle, color-coded by gender, with the person's name printed in the circle.

4.11 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.

4.12 Create a class called `Star` that represents a graphical depiction of a star. Let the constructor of the star accept the number of points in the star (4, 5, or 6), the radius of the star, and the center point location. Write an applet that draws a sky full of various types of stars.

4.13 Enhance the concept of the `LineUp` program to create a `PoliceLineUp` class. Instead of a stick figure, create a class called `Thug` that has a more realistic graphical representation. In addition to varying the person's height, vary the clothes and shoes by color, and add a hat or necktie for some thugs.

For additional programming projects, click the CodeMate icon below:

4.14 CODEMATE

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 reserves 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 the area within a program in which the variable can be referenced. An instance variable, declared at the class level, can be referenced in any method of the class. Local variables, including the formal parameters, declared within a particular method, can be referenced only in that method.

4.3 A UML diagram helps us visualize the entities (classes and objects) in a program as well as the relationships among them. UML diagrams are tools that help us capture the design of a program prior to writing it.

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

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

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

- A public method is called a service method for an object because it defines a service that the object provides.
- A private method is called a support method because it cannot be invoked from outside the object and is used to support the activities of other methods in the class.
- A public variable is a variable that can be directly accessed and modified by a client. This explicitly violates the principle of encapsulation and therefore should be avoided.
- A private variable is a variable that can be accessed and modified only from within the class. Variables almost always are declared with private visibility.

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

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

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

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

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

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

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

4.14 The `Applet start` method is invoked automatically every time the applet becomes active, such as when a browser returns to the page it is on. The `stop` method is invoked automatically when the applet becomes inactive.