SRA 4.9 If a case does not end with a `break` statement, processing continues into the statements of the next case. We usually want to use `break` statements in order to jump to the end of the `switch`.

SRA 4.10 An infinite loop is a repetition statement that will not terminate because of the basic logic of the condition. Specifically, the body of the loop never causes the condition to become false.

SRA 4.11 A `while` loop evaluates the condition first. If it is true, it executes the loop body. The `do` loop executes the body first and then evaluates the condition. Therefore, the body of a `while` loop is executed zero or more times, and the body of a `do` loop is executed one or more times.

SRA 4.12 A `for` loop is usually used when we know, or can calculate, how many times we want to iterate through the loop body. A `while` loop handles a more generic situation.

5 Writing Classes

CHAPTER OBJECTIVES

- Explore techniques for identifying the classes and objects needed in a program.
- Discuss the structure and content of a class definition.
- Establish the concept of object state using instance data.
- Describe the effect of visibility modifiers on methods and data.
- Explore the structure of a method definition, including parameters and return values.
- Discuss the structure and purpose of a constructor.
- Discuss the relationships among classes.
- Describe the effect of the `static` modifier on methods and data.
- Discuss issues related to the design of methods, including method decomposition and method overloading.

In previous chapters we used classes and objects 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, which is the heart of object-oriented programming. This chapter explores the basics of class definitions, including the structure of methods and the scope and encapsulation of data. It also examines the creation of static class members and overloaded methods.

5.1 Classes and Objects Revisited

In Chapter 1 we introduced basic object-oriented concepts, including a brief overview of objects and classes. In Chapters 2 and 3 we used several predefined classes from the Java standard class library to create objects and use them for the particular functionality they provide.

In this chapter we turn our attention to writing our own classes. Although existing class libraries provide many useful classes, the essence of object-oriented program development is the process of designing and implementing our own classes to suit our specific needs.

Recall the basic relationship between an object and a class: a class is a blueprint of an object. The class represents the concept of an object, and any object created from that class is a realization of that concept.

For example, from Chapter 3 we know that the `String` class represents a concept of a character string, and that each `String` object represents a particular string that contains specific characters.

Let's consider another example. Suppose a class called `Student` represents a student at a university. An object created from the `Student` class would represent a particular student. The `Student` class represents the general concept of a student, and every object created from that class represents an actual student attending the school. In a system that helps manage the business of a university, we would have one `Student` class and thousands of `Student` objects.

Recall that an object has a *state*, which is defined by the values of the *attributes* associated with that object. For example, the attributes of a student might include the student's name, address, major, and grade point average. The `Student` class establishes that each student has these attributes, and each `Student` object stores the values of these attributes for a particular student. In Java, an object's attributes are defined by variables declared within a class.

An object also has *behaviors*, which are defined by the *operations* associated with that object. The operations of a student might include the ability to update that student's address and compute that student's current grade point average. The `Student` class defines the operations, such as the details of how a grade point average is computed. These operations can then be executed on (or by) a particular `Student` object. Note that the behaviors of an object may modify the state of that object. In Java, an object's operations are defined by methods declared within a class.

Figure 5.1 lists some examples of classes, with some attributes and operations that might be defined for objects of those classes. It's up to the program designer to determine what attributes and operations are needed, which depends on the purpose of the program and the role a particular object plays in serving that purpose. Consider other attributes and operations you might include for these examples.

Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
Flight	Airline Flight number Origin city Destination city Current status	Set airline Set flight number Determine status
Employee	Name Department Title Salary	Set department Set title Set salary Compute wages Compute bonus Compute taxes

FIGURE 5.1 Examples of classes with some possible attributes and operations

Identifying Classes and Objects

A fundamental part of object-oriented software design is determining which classes should be created to define the program. We have to carefully consider how we want to represent the various elements that make up the overall solution. These classes determine the objects that we will manage in the system.

One way to identify potential classes is to identify the objects discussed in the program requirements. Objects are generally nouns. You literally may want to scrutinize a problem description, or a functional specification if available, to identify the nouns found in it. For example, Figure 5.2 on the next page shows part of a problem description with the nouns circled.

Of course, not every noun in the problem specification will correspond to a class in a program. Some nouns may be represented as attributes of other objects, and the designer may decide not to represent other nouns explicitly in the program at all. This activity is just a starting point that allows a developer to think about the types of objects a program will manage.

```
The user must be allowed to specify each
product by its primary characteristics,
including its name and product number. If the
bar code does not match the product, then an
error should be generated to the message window
and entered into the error log. The summary
report of all transactions must be structured
as specified in section 7.A.
```

FIGURE 5.2 Finding potential objects by identifying the nouns in a problem description

KEY CONCEPT
The nouns in a problem description may indicate some of the classes and objects needed in a program.

Remember that a class represents a group of objects with similar behavior. A plural noun in the specification, such as *products*, may indicate the need for a class that represents one of those items, such as `Product`. Even if there is only one of a particular kind of object needed in your system, it may best be represented as a class.

Classes that represent objects should generally be given names that are singular nouns, such as `Coin`, `Student`, and `Message`. A class represents a single item from which we are free to create as many instances as we choose.

Another key decision is whether to represent something as an object or as a primitive attribute of another object. For example, we may initially think that an employee's salary should be represented as an integer, and that may work for much of the system's processing. But upon further reflection, we might realize that the salary is based on the person's rank, which has upper and lower salary bounds that must be managed with care. Therefore, the final conclusion may be that we'd be better off representing all of that data and the associated behavior as a separate class.

In addition to classes that represent objects from the problem domain, we will probably need classes that support the work necessary to get the job done. For example, in addition to `Member` objects, we may want a separate class to help us manage all of the members of a club.

DESIGN FOCUS

Given the needs of a particular program, we want to strike a good balance between classes that are too general and those that are too specific. For example, it may complicate our design unnecessarily to create a separate class for each type of appliance that exists in a house. It may be sufficient to have a single `Appliance` class, with perhaps a piece of instance data that indicates what type of appliance it is. Then again, this may not be an adequate solution. It all depends on what the software is going to accomplish.

Keep in mind that when we are producing a real system, some of the classes we identify during design may already exist. Even if nothing matches exactly, there may be an old class that's similar enough to serve as the basis for our new class. The existing class may be part of the Java standard class library, part of a solution to a problem we've solved previously, or part of a library that can be bought from a third party. These are all examples of software reuse.

Assigning Responsibilities

Part of the process of identifying the classes needed in a program is assigning responsibilities to each class. Each class represents an object with certain behaviors that are defined by the methods of the class. Any activity that the program must accomplish must be represented somewhere in the behaviors of the classes. That is, each class is responsible for carrying out certain activities, and those responsibilities must be assigned as part of designing a program.

The behaviors of a class perform actions that make up the functionality of a program. Thus we generally use verbs for the names of behaviors and the methods that accomplish them.

Sometimes it is challenging to determine which is the best class to carry out a particular responsibility. A good designer considers multiple possibilities. Sometimes such analysis makes you realize that you could benefit from defining another class to shoulder the responsibility.

It's not necessary in the early stages of a design to identify all the methods that a class will contain. It is often sufficient to assign primary responsibilities and then consider how those responsibilities translate into particular methods.

5.2 Anatomy of a Class

Now that we've reviewed some important conceptual ideas underlying the development of classes and objects, let's dive into the programming details. In all of our previous examples, we've written a single class containing a single `main` method. Each of these classes represents a small but complete program. These programs often instantiate objects using predefined classes from the Java class library and then use those objects for the services they provide. The library classes are part of the program too, but we generally don't have to concern ourselves with their internal details. We really just need to know how to interact with them and can simply trust them to provide the services they promise.

We will continue to rely on library classes, but now we will also design and implement other classes as needed. Let's look at an example. The `SnakeEyes` class shown in Listing 5.1 contains a `main` method that instantiates two `Die`

LISTING 5.1

```
//********************************************************************
//  SnakeEyes.java          Java Foundations
//
//  Demonstrates the use of a programmer-defined class.
//********************************************************************

public class SnakeEyes
{
   //-----------------------------------------------------------------
   // Creates two Die objects and rolls them several times, counting
   // the number of snake eyes that occur.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      final int ROLLS = 500;
      int num1, num2, count = 0;

      Die die1 = new Die();
      Die die2 = new Die();

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

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

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

OUTPUT

```
Number of rolls: 500
Number of snake eyes: 12
Ratio: 0.024
```


objects (*die* is the singular of *dice*). The purpose of the program is to roll the dice and count the number of times both die show a 1 on the same throw (snake eyes).

The primary difference between this example and examples we've seen in previous chapters is that the `Die` class is not a predefined part of the Java class library. For this program to compile and run, we have to write the `Die` class ourselves, defining the services we want `Die` objects to perform.

A class can contain data declarations and method declarations, as depicted in Figure 5.3. The data declarations represent the data that will be stored in each object of the class. The method declarations define the services that those objects will provide. Collectively, the data and methods of a class are called the *members* of a class.

The classes we've written in previous examples follow this model as well, but contain no data at the class level and contain only one method (the `main` method). We'll continue to define classes like this, such as the `SnakeEyes` class, to define the starting point of a program.

True object-oriented programming, however, comes from defining classes that represent objects with well-defined state and behavior. For example, at any given moment a `Die` object is showing a particular face value, which we could refer to as the state of the die. A `Die` object also has various methods we can invoke on it, such as the ability to roll the die or get its face value. These methods represent the behavior of a die.

KEY CONCEPT

The heart of object-oriented programming is defining classes that represent objects with well-defined state and behavior.

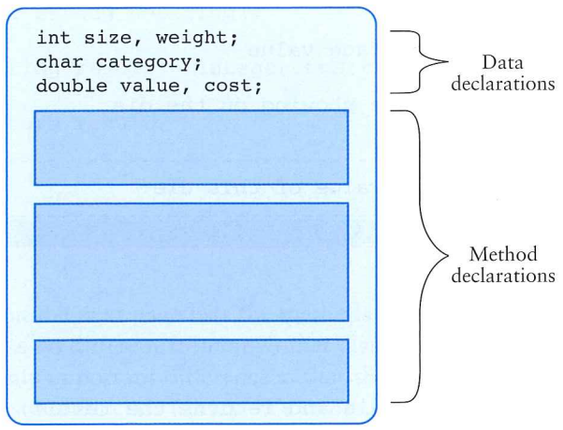


FIGURE 5.3 The members of a class: data declarations and method declarations

The `Die` class is shown in Listing 5.2. It contains two data values: an integer constant (`MAX`) that represents the maximum face value of the die, and an integer variable (`faceValue`) that represents the current face value of the die. It also contains a constructor called `Die` and four regular methods: `roll`, `setFaceValue`, `getFaceValue`, and `toString`.

You will recall from Chapters 2 and 3 that constructors are special methods that have the same name as the class. The `Die` constructor gets called when the `new` operator is used to create a new instance of the `Die` class, as occurs twice in the `main` method of the `SnakeEyes` class. The rest of the methods in the `Die` class define the various services provided by `Die` objects.

We use a header block of documentation to explain the purpose of each method in the class. This practice is not only crucial for anyone trying to understand the software but also separates the code visually so that it's easy for the eye to jump from one method to the next while reading the code.

LISTING 5.2

```
//********************************************************************
//  Die.java           Java Foundations
//
//  Represents one die (singular of dice) with faces showing values
//  between 1 and 6.
//********************************************************************

public class Die
{
   private final int MAX = 6; // maximum face value

   private int faceValue; // current value showing on the die

   //-----------------------------------------------------------------
   // Constructor: Sets the initial face value of this die.
   //-----------------------------------------------------------------
   public Die()
   {
      faceValue = 1;
   }

   //-----------------------------------------------------------------
   // Computes a new face value for this die and returns the result.
   //-----------------------------------------------------------------
```


LISTING 5.2 *continued*

```
   public int roll()
   {
      faceValue = (int)(Math.random() * MAX) + 1;

      return faceValue;
   }

   //-----------------------------------------------------------------
   // Face value mutator. The face value is not modified if the
   // specified value is not valid.
   //-----------------------------------------------------------------
   public void setFaceValue(int value)
   {
      if (value > 0 && value <= MAX)
         faceValue = value;
   }

   //-----------------------------------------------------------------
   // Face value accessor.
   //-----------------------------------------------------------------
   public int getFaceValue()
   {
      return faceValue;
   }

   //-----------------------------------------------------------------
   // Returns a string representation of this die.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = Integer.toString(faceValue);

      return result;
   }
}
```

Figure 5.4 on the next page lists the methods of the `Die` 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 `Die` class was not provided for us by the Java standard class library. We wrote it ourselves.

The methods of the `Die` class include the ability to roll the die, producing a new random face value. The `roll` method returns the new face value to the calling method,

`Die()`
Constructor: Sets the initial face value of the die to 1.

`int roll()`
Rolls the die by setting the face value to a random number in the appropriate range.

`void setFaceValue (int value)`
Sets the face value of the die to the specified value.

`int getFaceValue()`
Returns the current face value of the die.

`String toString()`
Returns a string representation of the die indicating its current face value.

FIGURE 5.4 Some methods of the `Die` class

but you can also get the current face value at any time by using the `getFaceValue` method. The `setFaceValue` method sets the face value explicitly, as if you had reached over and turned the die to whatever face you wanted. The `toString` method returns a representation of the die as a character string—in this case, it returns the numeric value of the die face as a string. The definitions of these methods have various parts, and we'll dissect them as we proceed through this chapter.

VideoNote
Dissecting the `Die` class

Let's mention the importance of the `toString` method at this point. The `toString` method of any object gets called automatically whenever you pass the object to a `print` or `println` method and when you concatenate an object to a character string. There is a default version of `toString` defined for every object, but the results are not generally useful. Therefore, it's usually a good idea to define a `toString` method for the classes that you create. The default version of `toString` is available because of inheritance, which we discuss in detail in Chapter 8.

For the examples in this book, we usually store each class in its own file. Java allows multiple classes to be stored in one file. But 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 `Die` is stored in a file called `Die.java`.

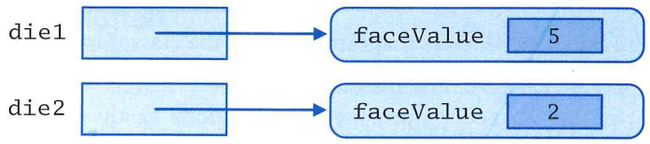
Instance Data

Note that in the `Die` class, the constant `MAX` and the variable `faceValue` 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. Because they have been declared at the class level (not within a method), these variables and constants can be referenced in any method of the class.

Attributes such as the variable `faceValue` are called *instance data* because new memory space is reserved for that variable every time an instance of the class is created. Each `Die` object has its own `faceValue` variable with its own data space. That's how each `Die` object can have its own state. That is, one die could be showing a 5 at the same time the other die is showing a 2. That's possible only because separate memory space for the `faceValue` variable is created for each `Die` object.

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

We can depict this situation as follows:



The `die1` and `die2` reference variables point to (that is, contain the address of) their respective `Die` objects. Each object contains a `faceValue` variable with its own memory space. Thus each object can store different values for its instance data.

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 Class Diagrams

As our programs become more complex, containing multiple classes, it's helpful to make use of a graphical notation to capture, visualize, and communicate the program design. Throughout the remainder of this text we use *UML diagrams* for this purpose. UML stands for the *Unified Modeling Language*, which has become the most popular notation for representing the design of an object-oriented program.

Several types of UML diagrams exist, each designed to show specific aspects of object-oriented programs. In this text, we focus primarily on UML *class diagrams* to show the contents of classes and the relationships among them.

In a UML diagram, each class is represented as a rectangle, possibly containing three sections to show the class name, its attributes (data), and its operations (methods). Figure 5.5 shows a class diagram containing the classes of the `SnakeEyes` program.

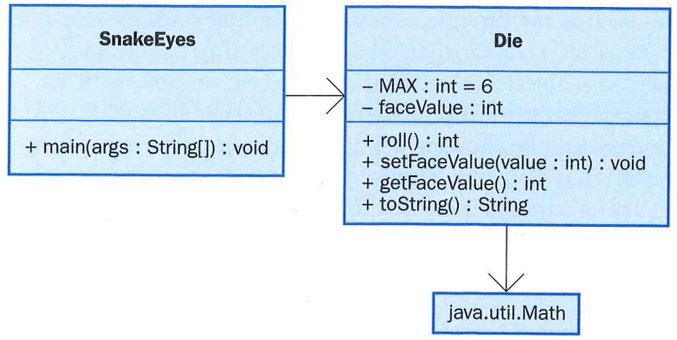


FIGURE 5.5 A UML class diagram showing the classes involved in the `SnakeEyes` program

KEY CONCEPT
A UML class diagram helps us visualize the contents of and relationships among the classes of a program.

UML is not designed specifically for Java programmers. It is intended to be language independent. Therefore, the syntax used in a UML diagram is not necessarily the same as that used in Java. For example, a variable's type is shown after the variable name, separated by a colon. Return types of methods are shown the same way. The initial value of an attribute can be shown in the class diagram if desired, as we do with the `MAX` constant in Figure 5.5. The + and – notations in front of variables and methods indicate their *visibility*, which is discussed in the next section.

A solid line connecting two classes in a UML diagram indicates that a relationship of one kind or another exists between the two classes. These lines, which are called *associations*, indicate that one class "knows about" and uses the other in some way. For example, an association might indicate that an object of one class creates an object of the other, and/or that one class invokes a method of the other. Associations can be labeled to indicate the details of the association.

A directed association uses an arrowhead to indicate that the association is particularly one-way. For example, the arrow connecting the `SnakeEyes` and `Die` classes in Figure 5.5 indicates that the `SnakeEyes` class "knows about" and uses the `Die` class, but not vice versa.

An association can show *multiplicity* by annotating the ends of the connection with numeric values. In this case, the diagram indicates that `SnakeEyes` is associated with exactly two `Die` objects. Both ends of an association can show multiplicity values, if desired. Multiplicity also can be expressed in terms of a range of values and by using wildcards for unknown values, as we'll see in later examples.

Other types of object-oriented relationships between classes are shown with different types of connecting lines and arrows. We will explore additional aspects of UML diagrams as we discuss the corresponding object-oriented programming concepts throughout the text.

UML diagrams are versatile. We can include whatever appropriate information is desired, depending on what we are trying to convey in a particular diagram. We might leave out the data and method sections of a class, for instance, if those details aren't relevant for a particular diagram. For example, the fact that the `Die` class makes use of the `Math` class from the Java API is indicated in Figure 5.5, but the details of the `Math` class are not identified. We also could have explicitly indicated the use of the `String` class, but that is rarely done because of its ubiquity.

5.3 Encapsulation

We mentioned in our overview of object-oriented concepts in Chapter 1 that an object should be *self-governing*. That is, the instance data of an object should be modified only by that object. For example, the methods of the `Die` class should be solely responsible for changing the value of the `faceValue` 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 that class. This characteristic is called *encapsulation*.

KEY CONCEPT
An object should be encapsulated, guarding its data from inappropriate access.

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 other objects that use it.

The nature of encapsulation is depicted graphically in Figure 5.6 on the next page. The code that uses an object, which is sometimes called the *client* of an object, should not be allowed to access variables directly. The client should call an object's methods, and those methods then interact with the data encapsulated within the object. For example, the `main` method in the `SnakeEyes` program calls

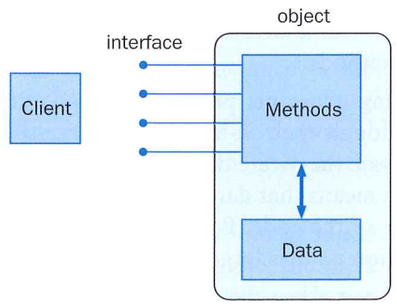


FIGURE 5.6 A client interacting with another object

the `roll` method of the `Die` objects. The `main` method should not (and in fact cannot) access the `faceValue` 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. In Chapter 2 we discussed the `final` modifier, which is used 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 text, and all of them are summarized in Appendix E.

Visibility Modifiers

Some of the 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 8.

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

Public variables violate encapsulation. They allow code external to the class in which the data are defined to reach in and access or modify the value of the data. Therefore, instance data should be defined with private visibility. Data declared as private can be accessed only by the methods of the class.

The visibility we apply to a method depends on the purpose of that method. Methods that provide services to the client 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, private methods are sometimes referred to as *support methods*.

The table in Figure 5.7 summarizes the effects of public and private visibility on both variables and methods.

The reason why giving constants public visibility is generally considered acceptable is that even though their values can be accessed directly, their values 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 *modified* directly by another part of the code. Because constants, by definition, cannot be changed, the encapsulation issue is largely moot.

UML class diagrams can show the visibility of a class member by preceding it with a particular character. A member with public visibility is preceded by a plus sign (+), and a member with private visibility is preceded by a minus sign (-). Review Figure 5.5 to see this notation used.

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

FIGURE 5.7 The effects of Public and Private visibility

Accessors and Mutators

Because instance data are generally declared with private visibility, a class usually provides services to access and modify data values. A method such as `getFaceValue` in the `Die` class is called an *accessor method* because it provides read-only access to a particular value. Likewise, a method such as `setFaceValue` is called a *mutator method* because it changes a particular value.

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

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

For example, if a class contains the instance variable `height`, it should also probably contain the methods `getHeight` and `setHeight`. Note that this naming convention capitalizes the first letter of the variable when it is used in the method names, which is consistent with how method names are written in general.

Some methods may provide accessor and/or mutator capabilities as a side effect of their primary purpose. For example, the `roll` method of the `Die` class changes the value of the variable `faceValue` and returns that new value as well. Note that the code of the `roll` method is guaranteed to keep the face value of

the die in the valid range (1 to `MAX`). Similarly, the `setFaceValue` method checks to see whether the specified value is in the valid range and ignores it if it is not. Service methods must be carefully designed to permit only appropriate access and valid changes. By encapsulating the data, the object can maintain this type of control.

Let's look at anther example. The program in Listing 5.3 instantiates a `Coin` object and then flips the coin multiple times, counting the number of times heads and tails come up. Notice that it uses a call to the method `isHeads` in the condition of an `if` statement to determine which result occurred.

The `Coin` class is shown in Listing 5.4 on page 186. It stores an integer constant called `HEADS` that represents the face value when the coin is showing heads. An instance variable called `face` represents the current state of the coin (which side is up) and has either the value 0 or the value 1. The `Coin` constructor initially flips the coin by calling the `flip` method, which determines the new state of the coin by randomly choosing a number (either 0 or 1). The `isHeads` method returns a `boolean` value based on the current face value of the coin. The `toString` method returns a character string indicating the current face showing on the coin.

A `Coin` object can be in one of two states: showing heads or showing tails. We represented this state in the `Coin` class as an integer value, `0` for tails and `1` for heads, stored in the `face` variable. Of course, this representation is arbitrary—we could have used `1` to represent tails. For that matter, we could have represented the coin's state using a `boolean` value, or a character string, or an enumerated type. We chose to use an integer because the methods for choosing a random result (`Math.random` in this case) return a numeric value and therefore eliminate extraneous conversions.

KEY CONCEPT

The way a class represents an object's state should be independent of how that object is used.

The way the `Coin` object represents its state internally is, and should be, irrelevant to the client using the object. That is, from the perspective of the `CountFlips` program, the way the `Coin` class represents its state doesn't matter.

We could have made the constant `HEADS` public so that the client could access it. But as an integer variable, its value is meaningless to the client. Providing the `isHeads` method is a cleaner object-oriented solution. The internal details of the `Coin` class could be rewritten, and as long as the `isHeads` method was written appropriately, the client would not have to change.

Although many classes will have classic getter and setter methods, we chose to design the `Coin` class without them. The only way the coin's state can be changed is to flip it randomly. Unlike a `Die` object, the user cannot explicitly set the state of a coin. This is a design decision, which could be made differently if circumstances dictated.

LISTING 5.3

```
//********************************************************************
//  CountFlips.java          Java Foundations
//
//  Demonstrates the use of 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 FLIPS = 1000;
      int heads = 0, tails = 0;

      Coin myCoin = new Coin();

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

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

      System.out.println("Number of flips: " + FLIPS);
      System.out.println("Number of heads: " + heads);
      System.out.println("Number of tails: " + tails);
   }
}
```

OUTPUT

```
Number of flips: 1000
Number of heads: 486
Number of tails: 514
```

LISTING 5.4

```
//********************************************************************
//  Coin.java           Java Foundations
//
//  Represents a coin with two sides that can be flipped.
//********************************************************************

public class Coin
{
   private final int HEADS = 0; // tails is 1

   private int face; // current side showing

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

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

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

   //-----------------------------------------------------------------
   // Returns the current face of this coin as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      return (face == HEADS) ? "Heads" : "Tails";
   }
}
```


Let's use the `Coin` class in another program. The `FlipRace` class is shown in Listing 5.5. The `main` method of `FlipRace` instantiates two `Coin` objects and flips them in tandem repeatedly until one of the coins comes up heads three times in a row.

The output of the `FlipRace` program shows the results of each coin flip. Note that the `coin1` and `coin2` objects are concatenated to character strings in the `println` statement. As we mentioned earlier, this situation causes the `toString` method of the object to be called, which returns a string to be printed. No explicit call to the `toString` method is needed.

The conditional operator is used in assignment statements to set the counters for the coins after they are flipped. For each coin, if the result is heads, the count is incremented. If not, the count is reset to zero. The `while` loop terminates when either or both counters reach the goal of three heads in a row.

LISTING 5.5

```
//********************************************************************
//  FlipRace.java           Java Foundations
//
//  Demonstrates the reuse of 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;
      Coin coin1 = new Coin(), coin2 = new Coin();

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

         System.out.println("Coin 1: " + coin1 + "\tCoin 2: " + coin2);
```

LISTING 5.5 *continued*

```
            // Increment or reset the counters
            count1 = (coin1.isHeads()) ? count1+1 : 0;
            count2 = (coin2.isHeads()) ? count2+1 : 0;
         }
         if (count1 < GOAL)
            System.out.println("Coin 2 Wins!");
         else
            if (count2 < GOAL)
               System.out.println("Coin 1 Wins!");
            else
               System.out.println("It's a TIE!");
      }
   }
```

OUTPUT

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

`FlipRace` uses the `Coin` class as part of its program, just as `CountFlips` did earlier. A well-designed class often can be reused in multiple programs, just as we've gotten used to reusing the classes from the Java API over and over.

5.4 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.

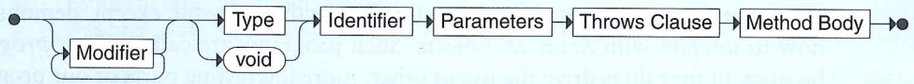
As we stated in Chapter 1, a method is a group of programming language statements that is given a name. A *method declaration* specifies the code that is executed when the method is invoked. Every method in a Java program is part of a particular class.

The header of a method declaration 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. We've defined the `main` method of a program many times in previous examples. Its definition follows the same syntax as any other method.

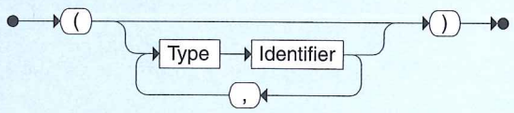
When a method is called, control transfers to that method. One by one, the statements of that method are executed. When that method is done, control returns to the location where the call was made, and execution continues.

The *called method* (the one that is invoked) might be part of the same class as the *calling method* that invoked it. If the called method is part of the same class, only the method name is needed to invoke it. If it is part of a different class, it is invoked through a reference to an object of that other class, as we've seen many times. Figure 5.8 shows the flow of execution as methods are called.

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 int computeArea(int length, int width)
{
  int area = length * width;
  return area;
}
```

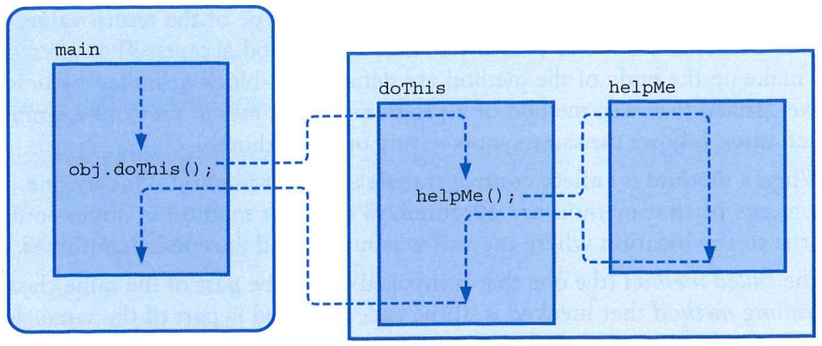


FIGURE 5.8 The flow of control following method invocations

Let's look at another example as we continue to explore the details of method declarations. The `Transactions` class shown in Listing 5.6 contains a `main` method that creates a few `Account` objects and invokes their services. The `Transactions` program doesn't really do anything useful except demonstrate how to interact with `Account` objects. Such programs are 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.

LISTING 5.6

```
//********************************************************************
//  Transactions.java          Java Foundations
//
//  Demonstrates the creation and use of multiple Account objects.
//********************************************************************

public class Transactions
{
   //-----------------------------------------------------------------
   //  Creates some bank accounts and requests various services.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Account acct1 = new Account("Ted Murphy", 72354, 25.59);
      Account acct2 = new Account("Angelica Adams", 69713, 500.00);
      Account acct3 = new Account("Edward Demsey", 93757, 769.32);
```


LISTING 5.6 *continued*

```
      acct1.deposit (44.10); // return value ignored

      double adamsBalance = acct2.deposit (75.25);
      System.out.println("Adams balance after deposit: " +
                         adamsBalance);

      System.out.println("Adams balance after withdrawal: " +
                         acct2.withdraw (480, 1.50));

      acct3.withdraw(-100.00, 1.50); // invalid transaction

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

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

OUTPUT

```
Adams balance after deposit: 575.25
Adams balance after withdrawal: 93.75

72354      Ted Murphy            $72.13
69713      Angelica Adams        $97.03
93757      Edward Demsey        $796.25
```

The `Account` class, shown in Listing 5.7 on the next page, represents a basic bank account. It contains instance data representing the name of the account's owner, the account number, and the account's current balance. The interest rate for the account is stored as a constant.

The constructor of the `Account` class accepts three parameters that are used to initialize the instance data when an `Account` object is instantiated. The other methods of the `Account` class perform various services on the account, such as making deposits and withdrawals. These methods examine the data passed into

them to make sure the requested transaction is valid. For example, the `withdraw` method prevents the withdrawal of a negative amount (which essentially would be a deposit). There is also an `addInterest` method that updates the balance by adding in the interest earned. These methods represent the valid ways to modify the balance, so a generic mutator such as `setBalance` is not provided.

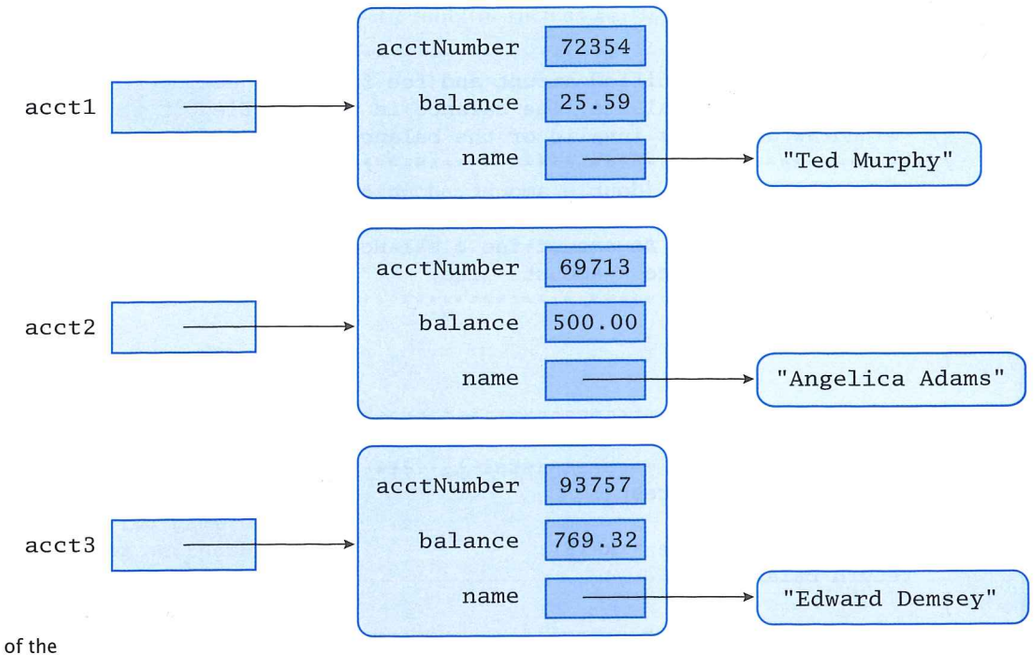
LISTING 5.7

```
//********************************************************************
//  Account.java       Java Foundations
//
//  Represents a bank account with basic services such as deposit
//  and withdraw.
//********************************************************************

import java.text.NumberFormat;

public class Account
{
   private final double RATE = 0.035; // interest rate of 3.5%

   private String name;
   private long acctNumber;
   private double balance;

   //-----------------------------------------------------------------
   //  Sets up this account with the specified owner, account number,
   //  and initial balance.
   //-----------------------------------------------------------------
   public Account(String owner, long account, double initial)
   {
      name = owner;
      acctNumber = account;
      balance = initial;
   }

   //-----------------------------------------------------------------
   //  Deposits the specified amount into this account and returns
   //  the new balance. The balance is not modified if the deposit
   //  amount is invalid.
   //-----------------------------------------------------------------
   public double deposit(double amount)
   {
      if (amount > 0)
         balance = balance + amount;
```


LISTING 5.7 *continued*

```
      return balance;
   }

   //-----------------------------------------------------------------
   //  Withdraws the specified amount and fee from this account and
   //  returns the new balance. The balance is not modified if the
   //  withdraw amount is invalid or the balance is insufficient.
   //-----------------------------------------------------------------
   public double withdraw(double amount, double fee)
   {
      if (amount+fee > 0 && amount+fee < balance)
         balance = balance - amount - fee;

      return balance;
   }

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

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

   //-----------------------------------------------------------------
   //  Returns a one-line description of this account as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      NumberFormat fmt = NumberFormat.getCurrencyInstance();

      return (acctNumber + "\t" + name + "\t" + fmt.format(balance));
   }
}
```

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



VideoNote
Discussion of the `Account` class

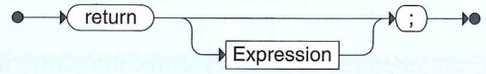
The rest of this section discusses in more detail the issues related to method declarations.

The `return` Statement

The return type specified in the method header can be a primitive type, a 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 of a program. The `setFaceValue` of the `Die` class and the `flip` method of the `Coin` class also have return types of `void`.

The `getFaceValue` and `roll` methods of the `Die` class return an `int` value that represents the value shown on the die. The `isHeads` method of the `Coin` class returns a `boolean` value that indicates whether the coin is currently showing heads. Several of the methods of the `Account` class return a `double` representing the updated balance. The `toString` method in all of these classes returns a `String` object.

Return Statement



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

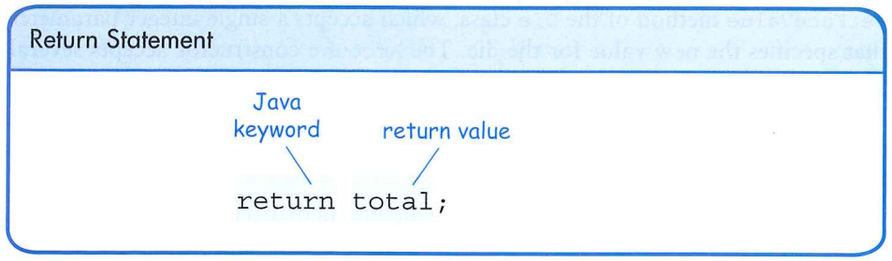
Examples:

```
return;

return distance * 4;
```

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 specified in the method header.

KEY CONCEPT
The value returned from a method must be consistent with the return type specified in the method header.

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. Such methods may 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.

The value that is returned from a method can be ignored in the calling method. Consider the following method invocation from the `Transactions` program:

```
acct1.deposit(44.10);
```

In this situation, the `deposit` method executes normally, updating the account balance accordingly, but the calling method simply makes no use of the returned value.

Constructors do not have a return type (not even `void`) and therefore cannot return a value. We discuss constructors in more detail later in this chapter.

Parameters

We introduced the concept of a parameter in Chapter 2, defining it as a value that is passed into a method when the method is invoked. Parameters provide data to a method that allow the method to do its job. Let's explore this issue in more detail.

The method declaration specifies the number and type of parameters that a method will accept. More precisely, the *parameter list* in the header of a method declaration specifies the type of each value that is passed into the method, and the name by which the called method will refer to each value. The corresponding parameter list in the invocation specifies the values that are passed in for that particular invocation.

The names of the parameters in the header of the method declaration are called *formal parameters*. The values passed into a method when it is invoked are called *actual parameters*, or *arguments*. The parameter list in both the declaration and the invocation is enclosed in parentheses after the method name. If there are no parameters, an empty set of parentheses is used.

None of the methods in the `Coin` and `Die` classes accepts parameters except the `setFaceValue` method of the `Die` class, which accepts a single integer parameter that specifies the new value for the die. The `Account` constructor accepts several parameters of various types to provide initial values for the object's instance data (this is common for constructors). The `withdraw` method in `Account` accepts two parameters of type `double`; note that the type of each formal parameter is listed separately even if the types are the same.

KEY CONCEPT
When a method is called, the actual parameters are copied into the formal parameters.

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

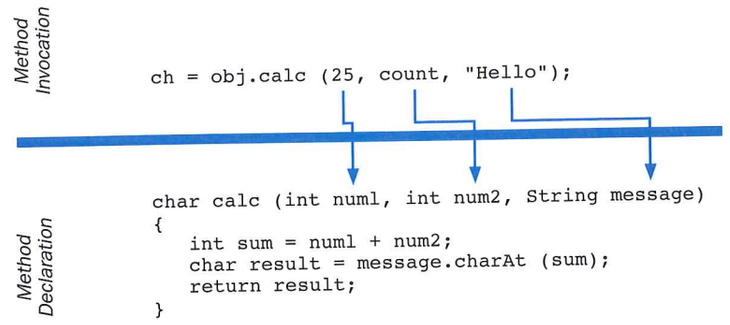


FIGURE 5.9 Passing parameters from the method invocation to the declaration

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 value of the second actual parameter into the second formal parameter, and so on, as shown in Figure 5.9. The types of the actual parameters must be consistent with the specified types of the formal parameters.

In the `Transactions` program, the following call is made:

```
acct2.withdraw(480, 1.50)
```

This call passes an integer value as the first parameter of the `withdraw` method, which is defined to accept a `double`. This is valid because the actual and formal parameters must be consistent, but they need not match exactly. A `double` variable can be assigned an integer value because this is a widening conversion. Thus it is also allowed when passing parameters.

We explore some of the details of parameter passing later in this chapter.

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 are declared in a class but not inside any particular method.

Local data have scope limited to the method in which they are is declared. The variable `result` declared in the `toString` method of the `Die` class is local data. Any reference to `result` in any other method of the `Die` class would have caused the compiler to issue an

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

error message. A local variable simply does not exist outside the method in which it is declared. On the other hand, instance data, declared at the class level, have a scope of the entire class; any method of the class can refer to instance data.

Because local data and instance data operate at different levels of scope, it's possible to declare a local variable inside a method with 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, the formal parameter `owner` in the `Account` constructor comes into existence when the constructor is called and goes out of existence when it finishes executing. To store these values in the object, the values of the parameters are copied into the instance variables of the newly created `Account` object.

Constructors Revisited

Let's discuss constructors a bit more. 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.

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

A constructor differs from a regular method in two ways. First, the name of a constructor is the same as the name of the class. Therefore, the name of the constructor in the `Die` class is `Die`, and the name of the constructor in the `Account` class is `Account`. Second, a constructor cannot return a value and does not have a return type specified in the method header.

Generally, a constructor is used to initialize the newly instantiated object. For instance, the constructor of the `Die` class sets the face value of the die to 1 initially. The constructor of the `Coin` class calls the `flip` method to put the coin

COMMON ERROR

A mistake commonly 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.

in an initial, random state. The constructor of the `Account` class sets the values of the instance variables to the values passed in as parameters to the constructor. The way you use a constructor to set up an object initially is another important design decision.

If we don't provide a constructor for a class, a *default constructor* that takes no parameters is automatically created and used. The default constructor generally has no effect on the newly created object. If the programmer provides a constructor, with or without parameters, the default constructor is not defined.

5.5 Static Class Members

We've used static methods in various situations in previous examples in the book. For example, all the methods of the `Math` class are static. Recall that a static method is one that is invoked through its class name, instead of through an object of that class.

Not only can methods be static, but variables can be static as well. We declare static class members using the `static` modifier.

Deciding whether to declare a method or variable as static is a key step in class design. Let's examine the implications of static variables and methods more closely.

Static Variables

So far, we've seen two categories of variables: local variables that are declared inside a method, and instance variables that are declared in a class but not inside a method. The term *instance variable* is used because each instance of the class has its own version of the variable. That is, each object has distinct memory space for each variable so that each object can have a distinct value for that variable.

KEY CONCEPT
A static variable is shared among all instances of a class.

A *static variable*, which is sometimes called a *class variable*, is shared among all instances of a class. There is only one copy of a static variable for all objects of the class. Therefore, changing the value of a static variable in one object changes it for all of the others. The reserved word `static` is used as a modifier to declare a static variable as follows:

```
private static int count = 0;
```

Memory space for a static variable is established when the class that contains it is referenced for the first time in a program. A local variable declared within a method cannot be static.

Constants, which are declared using the `final` modifier, are often declared using the `static` modifier. Because the value of constants cannot be changed, there might as well be only one copy of the value across all objects of the class.

Static Methods

In Chapter 3 we briefly introduced the concept of a *static method* (also called a *class method*). Static methods can be invoked through the class name. We don't have to instantiate an object of the class in order to invoke the method. In Chapter 3 we noted that all the methods of the `Math` class are static methods. For example, in the following line of code, the `sqrt` method is invoked through the `Math` class name.

```
System.out.println ("Square root of 27:" + Math.sqrt(27));
```

The methods in the `Math` class perform basic computations based on values passed as parameters. There is no object state to maintain in these situations, so there is no good reason to force us to create an object in order to request these services.

A method is made static by using the `static` modifier in the method declaration. As we've seen many times, the `main` method of a Java program is declared with the `static` modifier; this is done so that `main` can be executed by the interpreter without instantiating an object from the class that contains `main`.

Because static methods do not operate in the context of a particular object, they cannot reference instance variables, which exist only in an instance of a class. The compiler will issue an error if a static method attempts to use a nonstatic variable. A static method can, however, reference static variables, because static variables exist independent of specific objects. Therefore, all static methods, including the `main` method, can access only static or local variables.

The program in Listing 5.8 instantiates several objects of the `Slogan` class, printing each one out in turn. Then it invokes a method called `getCount` through the class name, which returns the number of `Slogan` objects that were instantiated in the program.

Listing 5.9 on page 202 shows the `Slogan` class. The constructor of `Slogan` increments a static variable called `count`, which is initialized to zero when it is declared. Therefore, `count` serves to keep track of the number of instances of `Slogan` that are created.

The `getCount` method of `Slogan` is also declared as static, which allows it to be invoked through the class name in the `main` method. Note that the only data referenced in the `getCount` method is the integer variable `count`, which is static. As a static method, `getCount` cannot reference any nonstatic data.

LISTING 5.8

```
//********************************************************************
//  SloganCounter.java       Java Foundations
//
//  Demonstrates the use of the static modifier.
//********************************************************************

public class SloganCounter
{
   //-----------------------------------------------------------------
   //  Creates several Slogan objects and prints the number of
   //  objects that were created.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Slogan obj;

      obj = new Slogan("Remember the Alamo.");
      System.out.println(obj);

      obj = new Slogan("Don't Worry. Be Happy.");
      System.out.println(obj);

      obj = new Slogan("Live Free or Die.");
      System.out.println(obj);

      obj = new Slogan("Talk is Cheap.");
      System.out.println(obj);

      obj = new Slogan("Write Once, Run Anywhere.");
      System.out.println(obj);

      System.out.println();
      System.out.println("Slogans created: " + Slogan.getCount());
   }
}
```

OUTPUT

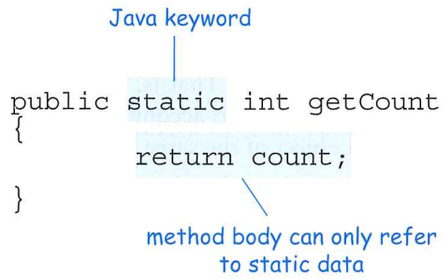
```
Remember the Alamo.
Don't Worry. Be Happy.
Live Free or Die.
Talk is Cheap.
Write Once, Run Anywhere.

Slogans created: 5
```

LISTING 5.9

```
//********************************************************************
//  Slogan.java       Java Foundations
//
//  Represents a single slogan or motto.
//********************************************************************

public class Slogan
{
   private String phrase;
   private static int count = 0;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the slogan and increments the number of
   //  instances created.
   //-----------------------------------------------------------------
   public Slogan(String str)
   {
      phrase = str;
      count++;
   }

   //-----------------------------------------------------------------
   // Returns this slogan as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      return phrase;
   }

   //-----------------------------------------------------------------
   // Returns the number of instances of this class that have been
   // created.
   //-----------------------------------------------------------------
   public static int getCount()
   {
      return count;
   }
}
```


The `getCount` method could have been declared without the `static` modifier, but then its invocation in the `main` method would have had to be done through an instance of the `Slogan` class instead of the class itself.

Static Method



5.6 Class Relationships

The classes in a software system have various types of relationships to each other. Three of the more common relationships are dependency, aggregation, and inheritance.

We've seen dependency relationships in many examples in which one class "uses" another. This section revisits the dependency relationship and explores the situation where a class depends on itself. We then explore aggregation, in which the objects of one class contain objects of another, creating a "has-a" relationship. Inheritance, which we introduced in Chapter 1, creates an "is-a" relationship between classes. We defer our detailed examination of inheritance until Chapter 8.

Dependency

In many previous examples, we've seen the idea of one class being dependent on another. This means that one class relies on another in some sense. Often the methods of one class invoke the methods of the other class. This establishes a "uses" relationship.

Generally, if class `A` uses class `B`, then one or more methods of class `A` invoke one or more methods of class `B`. If an invoked method is static, then `A` merely references `B` by name. If the invoked method is not static, then `A` must have access to a specific instance of class `B` in order to invoke the method. That is, `A` must have a reference to an object of class `B`.