

## 5.1 CLASS AND METHOD DEFINITIONS

*Concentrate all your thoughts upon the work at hand. The sun's rays do not burn until brought to a focus.*

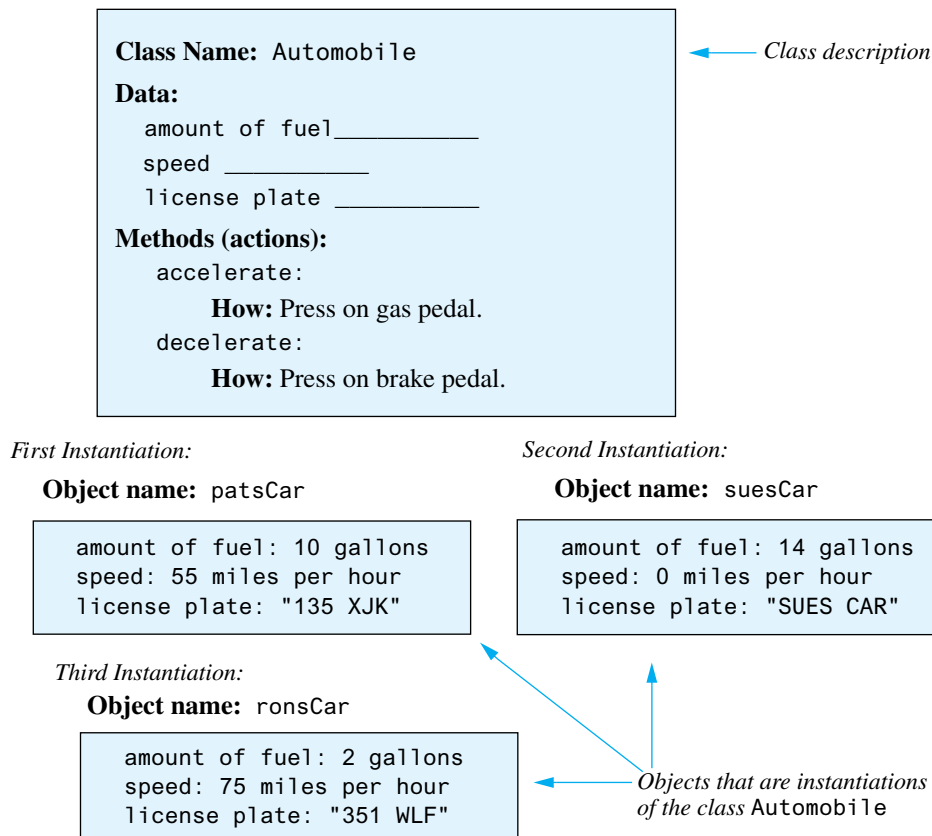
—ALEXANDER GRAHAM BELL, *How They Succeeded*, Orson Swett Marden, (1901)

A Java program consists of objects of various class types, interacting with one another. Before we go into the details of how you define your own classes and objects in Java, let's review and elaborate on what we already know about classes and objects.

**Objects** in a program can represent either objects in the real world—like automobiles, houses, and employee records—or abstractions like colors, shapes, and words. A class is the definition of a kind of object. It is like a plan or a blueprint for constructing specific objects. For example, Figure 5.1 describes a class called *Automobile*. The class is a general description of what an automobile is and what it can do.

Objects in a program can represent real-world things or abstractions

**FIGURE 5.1** A Class as a Blueprint



An instance of a class is an object

A class is like a blueprint for creating objects

A class specifies an object's attributes and defines its behaviors as methods

Use a UML class diagram to help design a class

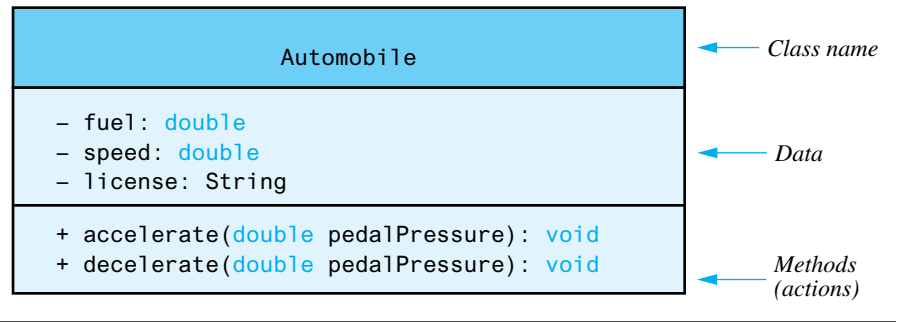
Objects of this class are particular automobiles. The figure shows three `Automobile` objects. Each of these objects satisfies the class definition of an `Automobile` object and is an **instance** of the `Automobile` class. Thus, we can create, or **instantiate**, several objects of the same class. The objects here are individual automobiles, while the `Automobile` class is a generic description of what an automobile is and does. This is, of course, a very simplified version of an automobile, but it illustrates the basic idea of what a class is. Let's look at some details.

A class specifies the attributes, or data, that objects of the class have. The `Automobile` class definition says that an `Automobile` object has three attributes or pieces of data: a number telling how many gallons of fuel are in the fuel tank, another number telling how fast the automobile is moving, and a string that shows what is written on the license plate. The class definition has no data—that is, no numbers and no string. The individual objects have the data, but the class specifies what kind of data they have.

The class also specifies what actions the objects can take and how they accomplish those actions. The `Automobile` class specifies two actions: `accelerate` and `decelerate`. Thus, in a program that uses the class `Automobile`, the only actions an `Automobile` object can take are `accelerate` and `decelerate`. These actions are described within the class by methods. All objects of any one class have the same methods. In particular, all objects of the class `Automobile` have the same methods. As you can see in our sample `Automobile` class, the definitions of the methods are given in the class definition and describe how objects perform the actions.

The notation in Figure 5.1 is a bit cumbersome, so programmers often use a simpler graphical notation to summarize some of the main properties of a class. This notation, illustrated in Figure 5.2, is called a **UML class diagram**, or simply a **class diagram**. **UML** is an abbreviation for **Universal Modeling Language**. The class described in Figure 5.2 is the same as the one described in Figure 5.1. Any annotations in Figure 5.2 that are new will be explained later in the chapter.

FIGURE 5.2 A Class Outline as a UML Class Diagram



Notice a few more things about a class and the objects that instantiate the class. Each object has a name. In Figure 5.1, the names are `patsCar`, `suesCar`, and `ronsCar`. In a Java program, these object names would be variables of type `Automobile`. That is, the data type of the variables is the class type `Automobile`.

Before we get further into the nitty-gritty of defining a simple class, let's review some of the things we said in Chapter 1 about storing classes in files and compiling them.

## Class Files and Separate Compilation

Whether you use a class taken from this book or one that you write yourself, you place each Java class definition in a file by itself. There are exceptions to this rule, but we will seldom encounter them, and we need not be concerned about them yet. The name of the file should begin with the name of the class and end in `.java`. So if you write a definition for a class called `Automobile`, it should be in a file named `Automobile.java`.

Each class is in a separate file

You can compile a Java class before you have a program in which to use it. The compiled bytecode for the class will be stored in a file of the same name, but ending in `.class` rather than `.java`. So compiling the file `Automobile.java` will create a file called `Automobile.class`. Later, you can compile a program file that uses the class `Automobile`, and you will not need to recompile the class definition for `Automobile`. This naming requirement applies to full programs as well as to classes. Notice that every program having a `main` method has a class name at the start of the file; this is the name you need to use for the file that holds the program. For example, the program you will see later in Listing 5.2 should be in a file named `DogDemo.java`. As long as all the classes you use in a program are in the same directory as the program file, you need not worry about directories. In Chapter 6, we will discuss how to use files from more than one directory.

### PROGRAMMING EXAMPLE

#### Implementing a Dog Class

To introduce the way that Java classes are defined let's create a simple class to represent a dog. Thus, we name our class `Dog` in Listing 5.1. Although its simplicity makes this first example easier to explain, it violates several important design principles. As we progress through this chapter we will discuss the weaknesses of the example's design and show how to improve it.

Each object of the `Dog` class stores the name, breed, and age of a dog. Additionally, each object has two actions, defined by the methods `writeOutput` and `getAgeInHumanYears`. The `writeOutput` method outputs the data stored about the dog and the `getAgeInHumanYears` method approximates the dog's equivalent age as if it were a human. Both the data items and the methods are

*Never tell people how to do things. Tell them what to do and they will surprise you with their ingenuity.*

—GEORGE S. PATTON

Information hiding sounds as though it could be a bad thing to do. What advantage could hiding information have? As it turns out, in computer science hiding certain kinds of information is considered a good programming technique, one that makes the programmer's job simpler and the programmer's code easier to understand. It is basically a way to avoid "information overload."

## Information Hiding

Separate the  
*what* from the  
*how*

A programmer who is using a method that you have defined does not need to know the details of the code in the body of the method definition to be able to use the method. If a method—or other piece of software—is well written, a programmer who uses the method need only know *what* the method accomplishes and not *how* the method accomplishes its task. For example, you can use the `Scanner` method `nextInt` without even looking at the definition of that method. It is not that the code contains some secret that is forbidden to you. The point is that viewing the code will not help you use the method, but it will give you more things to keep track of, which could distract you from your programming tasks.

Designing a method so that it can be used without any need to understand the fine detail of the code is called **information hiding**. The term emphasizes the fact that the programmer acts as though the body of the method were hidden from view. If the term *information hiding* sounds too negative to you, you can use the term **abstraction**. The two terms mean the same thing in this context. This use of the term *abstraction* should not be surprising. When you abstract something, you lose some of the details. For example, an abstract of a paper or a book is a brief description of the paper or book, as opposed to the entire document.

### ■ PROGRAMMING TIP When You Design a Method, Separate What from How

Methods should be self-contained units that are designed separately from the incidental details of other methods and separately from any program that uses the method. A programmer who uses a method should need only know what the method does, not how it does it. ■

## Precondition and Postcondition Comments

A precondition  
states a method's  
requirements

An efficient and standard way to describe what a method does is by means of specific kinds of comments known as preconditions and postconditions. A method's **precondition** comment states the conditions that must be true before the method is invoked. The method should not be used, and cannot be expected to perform correctly, unless the precondition is satisfied.

The **postcondition** comment describes all the effects produced by a method invocation. The postcondition tells what will be true after the method is executed in a situation in which the precondition holds. For a method that returns a value, the postcondition will include a description of the value returned by the method.

A **postcondition** states a method's effect

For example, the following shows some suitable precondition and postcondition comments for the method `writeOutput` shown in Listing 5.3:

```
/**
    Precondition: The instance variables of the calling
    object have values.
    Postcondition: The data stored in (the instance variables
    of) the receiving object have been written to the screen.
*/
public void writeOutput()
```

The comments for the method `predictPopulation` in Listing 5.6 can be expressed as follows:

```
/**
    Precondition: years is a nonnegative number.
    Postcondition: Returns the projected population of the
    receiving object
    after the specified number of years.
*/
public int predictPopulation(int years)
```

If the only postcondition is a description of the value returned, programmers usually omit the word *postcondition*. The previous comment would typically be written in the following alternative way:

```
/**
    Precondition: years is a nonnegative number.
    Returns the projected population of the receiving object
    after the specified number of years.
*/
public int predictPopulation(int years)
```

Some design specifications may require preconditions and postconditions for all methods. Others omit explicit preconditions and postconditions from certain methods whose names make their action obvious. Names such as `readInput`, `writeOutput`, and `set` are often considered self-explanatory. However, the sound rule to follow is to adhere to whatever guidelines your instructor or supervisor gives you, and when in doubt, add preconditions and postconditions.

Some programmers prefer not to use the words *precondition* and *postcondition* in their comments. However, you should always think in terms of preconditions and postconditions when writing method comments. The really important thing is not the words *precondition* and *postcondition*, but the concepts they name. Note that precondition and postcondition comments are examples of assertions.

## The public and private Modifiers

Any class can use a public class, method, or instance variable

As you know, the modifier `public`, when applied to a class, method, or instance variable, means that any other class can directly use or access the class, method, or instance variable by name. For example, the program in Listing 5.7 contains the following three lines, which set the values of the public instance variables for the object `speciesOfTheMonth`:

```
speciesOfTheMonth.name = "Klingon ox";
speciesOfTheMonth.population = 10;
speciesOfTheMonth.growthRate = 15;
```

---

### LISTING 5.7 Using a Method That Has a Parameter

---

```
/**
 * Demonstrates the use of a parameter
 * with the method predictPopulation.
 */
public class SpeciesSecondTryDemo
{
    public static void main(String[] args)
    {
        SpeciesSecondTry speciesOfTheMonth = new
            SpeciesSecondTry();
        System.out.println("Enter data on the Species of the " +
            "Month:");
        speciesOfTheMonth.readInput();
        speciesOfTheMonth.writeOutput();
        int futurePopulation =
            speciesOfTheMonth.predictPopulation(10);
        System.out.println("In ten years the population will be " +
            futurePopulation);
        //Change the species to show how to change
        //the values of instance variables:
        speciesOfTheMonth.name = "Klingon ox";
        speciesOfTheMonth.population = 10;
        speciesOfTheMonth.growthRate = 15;
        System.out.println("The new Species of the Month:");
        speciesOfTheMonth.writeOutput();
        System.out.println("In ten years the population will be " +
            speciesOfTheMonth.predictPopulation(10));
    }
}
```

---

### Sample Screen Output

The output is exactly the same as in Listing 5.4.

---

The object `speciesOfTheMonth` is an object of the class `SpeciesSecondTry`, whose definition appears in Listing 5.6. As you can see by looking at that class definition, the instance variables `name`, `population`, and `growthRate` all have the modifier `public`, and so the preceding three statements are perfectly valid.

While it is normal to have public classes and methods, it is *not* a good programming practice to make the instance variables of a class public. Typically, all instance variables should be private. You make an instance variable private by using the modifier `private` instead of the modifier `public`. The keywords `public` and `private` are examples of **access modifiers**.

Instance variables should be private

Suppose that we change the modifier that is before the instance variable `name` in the definition of the class `SpeciesSecondTry` in Listing 5.6 from `public` to `private` so that the class definition begins as follows:

```
public class SpeciesSecondTry
{
    private String name; //Private!
    public int population;
    public double growthRate;
```

With this change, the following Java statement in Listing 5.7 is invalid:

```
speciesOfTheMonth.name = "Klingon ox"; //Invalid when private.
```

The following two statements remain valid, because we left the modifiers of `population` and `growthRate` as `public`:

```
speciesOfTheMonth.population = 10;
speciesOfTheMonth.growthRate = 15;
```

When an instance variable is private, its *name* is not accessible *outside* of the class definition. Even so, you can still use its name in any way you wish within any method *inside* the class definition. In particular, you can directly change the value of the instance variable. Outside of the class definition, however, you cannot make any direct reference to the instance variable's name.

Private instance variables are accessible by name only within their own class

For example, let's make the three instance variables in the class `SpeciesSecondTry` private, but leave the method definitions unchanged. The result is shown in Listing 5.8 as the class `SpeciesThirdTry`. Because the instance variables are all private, the last three of the following statements would be invalid within any class other than the class `SpeciesThirdTry`:

```
SpeciesThirdTry secretSpecies = new SpeciesThirdTry();
                                //Valid
secretSpecies.readInput(); //Valid
secretSpecies.name = "Aardvark"; //Invalid
System.out.println(secretSpecies.population); //Invalid
System.out.println(secretSpecies.growthRate); //Invalid
```

**LISTING 5.8 A Class with Private Instance Variables**

```
import java.util.Scanner;
public class SpeciesThirdTry
{
    private String name;
    private int population;
    private double growthRate;
    <The definitions of the methods readInput, writeOutput, and
    predictPopulation are the same as in Listing 5.3 and
    Listing 5.6.>
}
```

*We will give an even better version of this class later in the chapter.*

Notice, however, that the invocation of the method `readInput` is valid. So there is still a way to set the instance variables of an object, even though those instance variables are private. Within the definition of the method `readInput` (shown in Listing 5.3) are assignment statements such as

```
name = keyboard.nextLine();
```

and

```
population = keyboard.nextInt();
```

that set the value of instance variables. Thus, making an instance variable private does not mean that you cannot change it. It means only that you cannot use the instance variable's *name* to refer directly to the variable anywhere outside of the class definition.

Methods can also be private. If a method is marked private, it cannot be invoked outside of the class definition. However, a private method can still be invoked within the definition of any other method in that same class. Most methods are public, but if you have a method that will be used only within the definition of other methods of that class, it makes sense to make this "helping" method private. Using private methods is another way of hiding implementation details within the class.

Classes themselves can be private as well, but we will not discuss that until Chapter 12.

Private methods are called only within their own class.

**RECAP The public and private Modifiers**

Within a class definition, each instance variable declaration and each method definition, as well as the class itself, can be preceded by either `public` or `private`. These access modifiers specify where a class, instance variable, or method can be used. If an instance variable is private, its name cannot be used to access it outside of the class definition. However, it can be used within the definitions of methods in its class. If an instance variable is public, there are no restrictions on where you can use its name.

*(continued)*



If a method definition is private, the method cannot be invoked outside of the class definition. However, it can be invoked within the definitions of methods in its class. If the method is public, you can invoke it anywhere without restriction.

Normally, all instance variables are private and most methods are public.

### ■ PROGRAMMING TIP Instance Variables Should Be Private

You should make all the instance variables in a class private. By doing so, you force the programmer who uses the class—whether that person is you or someone else—to access the instance variables only via the class’s methods. This allows the class to control how a programmer looks at or changes the instance variables. The next programming example illustrates why making instance variables private is important. ■

### PROGRAMMING EXAMPLE

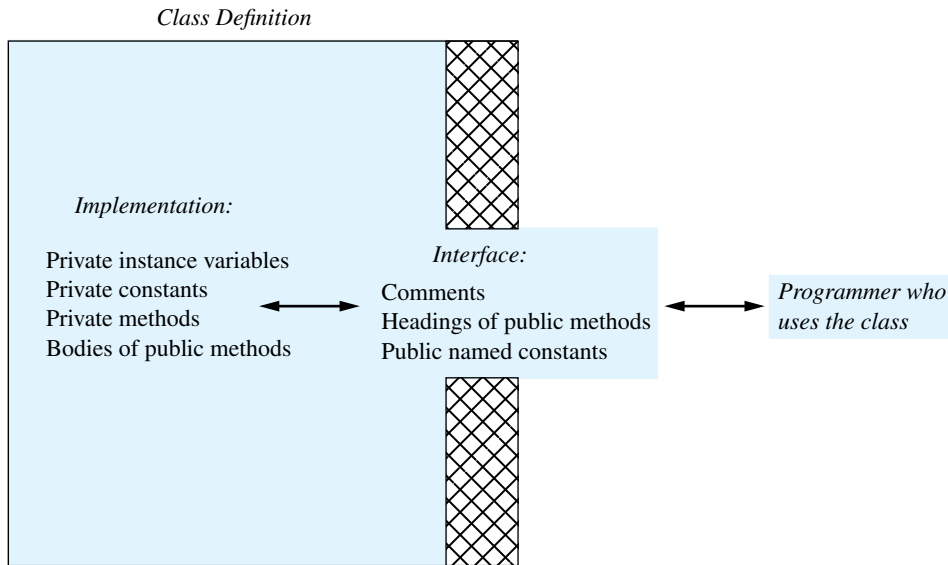
#### A Demonstration of Why Instance Variables Should Be Private

Listing 5.9 shows a simple class of rectangles. It has three private instance variables to represent a rectangle’s width, height, and area. The method `setDimensions` sets the width and height, and the method `getArea` returns the rectangle’s area.

#### LISTING 5.9 A Class of Rectangles

```
/**
 * Class that represents a rectangle.
 */
public class Rectangle
{
    private int width;
    private int height;
    private int area;

    public void setDimensions(int newWidth, int newHeight)
    {
        width = newWidth;
        height = newHeight;
        area = width * height;
    }
    public int getArea()
    {
        return area;
    }
}
```

**FIGURE 5.3 A Well-Encapsulated Class Definition**

- Declare all the instance variables in the class as private.
- Provide public accessor methods to retrieve the data in an object. Also provide public methods for any other basic needs that a programmer will have for manipulating the data in the class. Such methods could include public mutator methods.
- Place a comment before each public method heading that fully specifies how to use the method.
- Make any helping methods private.
- Write comments within the class definition to describe implementation details.

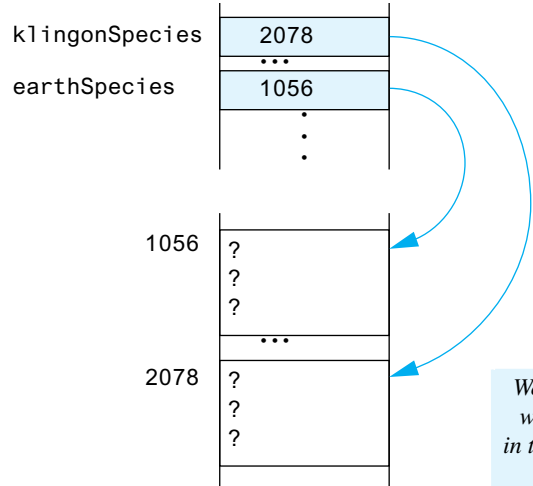
#### Encapsulation guidelines

The comments in a class definition that describe how to use both the class and each public method are part of the class interface. As we indicated, these comments are usually placed before the class definition and before each method definition. Other comments clarify the implementation. A good rule to follow is to use the `/** */` style for class-interface comments and the `//` style for implementation comments.

When you use encapsulation to define your class, you should be able to go back and change the implementation details of the class definition without requiring changes in any program that uses the class. This is a good way to test whether you have written a well-encapsulated class definition. Often, you will have very good reasons for changing the implementation details of a class definition. For example, you may come up with a more efficient way to implement a method so that the method invocations run faster. You might

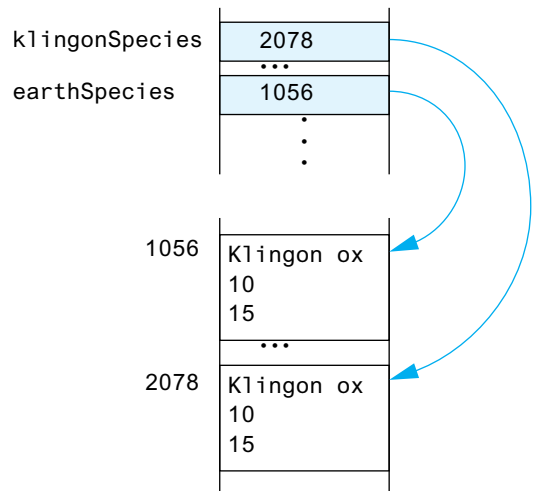
**FIGURE 5.6** The Dangers of Using == with Objects

```
klingspecies = new SpeciesFourthTry();
earthSpecies = new SpeciesFourthTry();
```



*We do not know what memory addresses will be used. We used 1056 and 2078 in this figure, but they could be almost any numbers.*

```
klingspecies.setSpecies("Klingon ox", 10, 15);
earthSpecies.setSpecies("Klingon ox", 10, 15);
```



```
if (klingspecies == earthSpecies)
    System.out.println("They are EQUAL.");
else
    System.out.println("They are NOT equal.");
```

*The output is They are Not equal, because 2078 is not equal to 1056.*

**LISTING 5.17** Defining an equals Method

---

```
import java.util.Scanner;
public class Species
{
    private String name;
    private int population;
    private double growthRate;

    <The definition of the methods readInput, writeOutput, and
    predictPopulation go here. They are the same as in
    Listing 5.3 and Listing 5.6.>

    <The definition of the methods setSpecies, getName,
    getPopulation, and getGrowthRate go here. They are the
    same as in Listing 5.11.>
    public boolean equals(Species otherObject)
    {
        return (this.name.equalsIgnoreCase(otherObject.name)) &&
            (this.population == otherObject.population) &&
            (this.growthRate == otherObject.growthRate);
    }
}
```

*equalsIgnoreCase is a method of the class String.*

---

Notice that the method `equals` in Listing 5.17 always returns a true or false value, and so the method's return type is `boolean`. The return statement may seem a bit strange, but it is nothing other than a boolean expression of the kind you might use in an `if-else` statement. It may help you to understand this method if you note that the definition of `equals` in Listing 5.17 can be expressed by the following pseudocode:

A method can  
return a boolean  
value

```
if ((this.name.equalsIgnoreCase(otherObject.name)) &&
    (this.population == otherObject.population) &&
    (this.growthRate == otherObject.growthRate))
then return true
otherwise return false
```

This change would make the following statement from the program in Listing 5.18:

```
if (s1.equals(s2))
    System.out.println("Match with the method equals.");
else
    System.out.println("Do Not match with the method " +
        "equals.");
```

## CHAPTER SUMMARY

- Classes have instance variables to store data and method definitions that perform actions.
- Each class, instance variable, and method definition can be either public or private. Those that are public can be used or accessed anywhere. A private instance variable cannot be accessed by name outside of its class definition. However, it can be used within the definitions of methods in its class. A private method definition cannot be invoked outside of its class definition. However, it can be invoked within the definitions of methods in its class.
- Instance variables should be private, even though as a result, they cannot be referenced by name except within the definition of a method of the same class.
- Accessor methods return the value of an instance variable. Mutator methods set the value of one or more instance variables.
- Every method belongs to some class and is available to objects created from that class.
- There are two kinds of methods: methods that return a value and void methods, which do not return a value.
- You can use the invocation of a method that returns a single quantity anywhere that you can use a value of the method's return type.
- You follow the invocation of a void method with a semicolon to form a statement that performs an action.
- The keyword `this`, when used within a method definition, represents the object that receives the invocation of the method.
- A local variable is a variable declared within a method definition. The variable does not exist outside of its method.
- Arguments in a method invocation must match the formal parameters in the method heading with respect to their number, their order, and their data types.
- The formal parameters of a method behave like local variables. Each is initialized to the value of the corresponding argument when the method is called. This mechanism of substituting arguments for formal parameters is known as the call-by-value mechanism.
- Methods can have parameters of a primitive type and/or parameters of a class type, but the two types of parameters behave differently. A parameter

of a primitive type is initialized to the primitive value of its corresponding argument. A parameter of a class type is initialized to the memory address, or reference, of the corresponding argument object.

- Any change made to a parameter of a primitive type is *not* made to the corresponding argument.
- A parameter of a class type becomes another name for the corresponding argument in a method invocation. Thus, any change that is made to the state of the parameter will be made to the corresponding argument. However, if a parameter is replaced by another object within the method definition, the original argument is not affected.
- A method definition can include a call to another method that is in either the same class or a different class.
- A block is a compound statement that declares a local variable.
- Encapsulation means that data and actions are combined into a single item—a class object—and that the details of the implementation are hidden. Making all instance variables private is part of the encapsulation process.
- A method's precondition comment states the conditions that must be true before the method is invoked. Its postcondition comment tells what will be true after the method is executed. That is, the postcondition describes all the effects produced by an invocation of the method if the precondition holds. Preconditions and postconditions are kinds of assertions.
- The utility program `javadoc` creates documentation from a class's comments that are written in a certain form.
- Class designers use a notation called UML to represent classes.
- Unit testing is a methodology in which the programmer writes a suite of tests to determine if individual units of code are operating properly.
- The operators `=` and `==`, when used on objects of a class, do not behave the same as they do when used on primitive types.
- You usually want to provide an `equals` method for the classes you define.
- The `GraphicsContext` object in a JavaFX application lets you draw graphics, text, or images.
- You add a label to a JavaFX application by choosing a layout and adding the label to the layout.

### Exercises

1. Design a class to represent a credit card. Think about the attributes of a credit card; that is, what data is on the card? What behaviors might be reasonable for a credit card? Use the answers to these questions to write a UML class diagram for a credit card class. Then give three examples of instances of this class.
2. Repeat Exercise 1 for a credit card account instead of a credit card. An account represents the charges and payments made using a credit card.
3. Repeat Exercise 1 for a cheque instead of a credit card.
4. Repeat Exercise 1 for an online payment on Apple Pay instead of a credit card.
5. Consider a Java class that you could use to get an acceptable integer value from the user. An object of this class will have the attributes
  - Minimum accepted value
  - Maximum accepted value
  - Prompt stringand the following method:
  - `getValue` displays the prompt and reads a value using the class `Scanner`. If the value read is not within the allowed range, the method should display an error message and ask the user for a new value, repeating these actions until an acceptable value is entered. The method then returns the value read.
  - a. Write preconditions and postconditions for the method `getValue`.
  - b. Implement the class in Java.
  - c. Write some Java statements that test the class.
6. Consider a class that keeps track of the students in a session. An object of this class will have the attributes
  - Total number of students
  - Total number of scholarships awarded
  - Scholarships per student
  - Total scholarship amount
  - Donations received for scholarships
  - Unutilized balance of donationsand the following methods:
  - `registerScholarships(n)` records the scholarship amount to  $n$  percent of the total number of students. If  $n$  is larger than 10%, the scholarship amount per student will be reduced.
  - `displayUnutilizedDonations` displays the amount of donations received and the amount that remains unutilized.

- a. Implement the class in Java.
  - b. Write some Java statements that test the class.
7. Consider a class `MotorBoat` that represents motorboats. A motorboat has attributes for
- The capacity of the fuel tank
  - The amount of fuel in the tank
  - The maximum speed of the boat
  - The current speed of the boat
  - The efficiency of the boat's motor
  - The distance traveled

The class has methods to

- Change the speed of the boat
- Operate the boat for an amount of time at the current speed
- Refuel the boat with some amount of fuel
- Return the amount of fuel in the tank
- Return the distance traveled so far

If the boat has efficiency  $e$ , the amount of fuel used when traveling at a speed  $s$  for time  $t$  is  $e \times s^2 \times t$ . The distance traveled in that time is  $s \times t$ .

- a. Write a method heading for each method.
  - b. Write preconditions and postconditions for each method.
  - c. Write some Java statements that test the class.
  - d. Implement the class.
8. Consider a class `PersonAddress` that represents an entry in an address book. Its attributes are
- The first name of the person
  - The last name of the person
  - The e-mail address of the person
  - The telephone number of the person

It will have methods to

- Access each attribute
  - Change the e-mail address
  - Change the telephone number
  - Test whether two instances are equal based solely on name
- a. Write a method heading for each method.
  - b. Write preconditions and postconditions for each method.
  - c. Write some Java statements that test the class.
  - d. Implement the class.



9. Consider a class `RatingScore` that represents a numeric rating for something such as a movie. Its attributes are

- A description of what is being rated
- The maximum possible rating
- The rating

It will have methods to

- Get the rating from a user
- Return the maximum rating possible
- Return the rating
- Return a string showing the rating in a format suitable for display

- a. Write a method heading for each method.
- b. Write preconditions and postconditions for each method.
- c. Write some Java statements that test the class.
- d. Implement the class.

10. Consider a class `ScienceFairProjectRating` that will be used to help judge a science fair project. It will use the class `RatingScore` described in the previous exercise. The attributes for the new class are

- The name of the project
- A unique identification string for the project
- The name of the person
- A rating for the creative ability (max. 30)
- A rating for the scientific thought (max. 30)
- A rating for thoroughness (max. 15)
- A rating for technical skills (max. 15)
- A rating for clarity (max. 10)

It will have methods to

- Get the number of judges
- Get all the ratings for a particular project
- Return the total of the ratings for a particular project
- Return the maximum total rating possible
- Return a string showing a project's rating in a format suitable for display

- a. Write a method heading for each method.
- b. Write preconditions and postconditions for each method.
- c. Write some Java statements that test the class.
- d. Implement the class.

## PRACTICE PROGRAMS

*Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.*

1. Write a program to answer questions like the following: Suppose the species Klingon ox has a population of 100 and a growth rate of

15 percent, and the species elephant has a population of 10 and a growth rate of 35 percent. How many years will it take for the elephant population to exceed the Klingon ox population? Use the class `Species` in Listing 5.19. Your program will ask for the data on both species and will respond by telling you how many years it will take for the species that starts with the lower population to outnumber the species that starts with the higher population. The two species may be entered in any order. It is possible that the species with the smaller population will never outnumber the other species. In this case, your program should display a suitable message stating this fact.

2. Define a class called `Salary`. An object of this class is used to calculate salaries, so it calculates a salary that is a whole number greater than basic salary. Include methods to set the basic salary to 5000, to increase the salary by 10%, and to decrease the salary by 10%. Be sure that no method allows the value of the counter to become less than the basic salary. Also include an accessor method that returns the current salary value, as well as a method that displays the salary on the screen. Do not define an input method. The only method that can set the basic salary is the one that sets it to 5000. Write a program to test your class definition. (*Hint: You need only one instance variable.*)
3. Redo or do Practice Program 8 from Chapter 4, except write a class for the Magic 8 Ball. Write a program that simulates the Magic 8 Ball game. The class should have a method that returns an answer at random. Test your class by outputting the answer several times.
4. Define a `Trivia` class that contains information about a single trivia question. The question and answer should be defined as instance variables of type `String`. Create accessor and mutator methods. In your main method create two `Trivia` objects with questions and answers of your choice. Then for each `Trivia` object have your program ask the question, input an answer, compare the typed answer to the actual answer, and output if the user's answer was correct or incorrect.
5. Define a `Beer` class that contains the following instance variables with accessors/mutators:

```
String name;    // The name of the beer
double alcohol; // The percent alcohol of the beer, e.g.
                // 0.05 for 5%
```

Add the following method:

```
// This method returns the number of drinks that a person
// of (weight) pounds can drink using the alcohol percentage
// in the beer, assuming a drink of 12 ounces. This is an
// estimate. The method assumed that the legal limit is 0.08 blood
// alcohol.
public double intoxicated(double weight)
```

## GOTCHA Privacy Leaks

A private instance variable of a class type names an object that, for certain classes, can be modified outside of the class containing the instance variable. To avoid this problem, you can do any of the following:

- Declare the instance variable's type to be of a class that has no set methods, such as the class `String`.
- Omit accessor methods that return an object named by an instance variable of a class type. Instead, define methods that return individual attributes of such an object.
- Make accessor methods return a clone of any object named by an instance variable of a class type, instead of the object itself. ■

## SELF-TEST QUESTION

40. Give the definitions of three accessor methods that you can use instead of the single accessor method `getFirst` within the definition of the class `PetPair` in Listing 6.18. One method will return the pet's name, one will return the pet's age, and one will return the pet's weight. These new accessor methods will not produce the problem described in this section. They will return all of the data in an object of the class `PetPair`, but will not return any object whose mutator methods can change its state.

## 6.6 ENUMERATION AS A CLASS

*The sands are number'd that make up my life.*

—WILLIAM SHAKESPEARE, *Henry VI, Part 3*

Chapter 3 introduced enumerations and mentioned that the compiler creates a class when it encounters an enumeration. This section talks about that class. Although you should consider using enumerations in your programs, the ideas in this section are not central to the presentation in the rest of the book.

Let's define a simple enumeration for the suits of playing cards, as follows:

```
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}
```

The compiler creates a class `Suit`. The enumerated values are names of public static objects whose type is `Suit`. As you know, you reference each of these values by prefacing its name with `Suit` and a dot, as in the following assignment statement:

```
Suit s = Suit.DIAMONDS;
```

Some  
enumeration  
methods

The class `Suit` has several methods, among which are `equals`, `compareTo`, `ordinal`, `toString`, and `valueOf`. The following examples show how we can use these methods, assuming that the variable `s` references `Suit.DIAMONDS`:

- `s.equals(Suit.HEARTS)` tests whether `s` equals `HEARTS`. In this case, `equals` returns `false`, since `DIAMONDS` is not equal to `HEARTS`.
- `s.compareTo(Suit.HEARTS)` tests whether `s` is before, at, or after `HEARTS` in the definition of `Suit`. Like the method `compareTo` in the class `String`—which Chapter 3 describes—`compareTo` here returns an integer that is negative, zero, or positive, according to the outcome of the comparison. In this case, `compareTo` return a negative integer, since `DIAMONDS` appears before `HEARTS` in the enumeration.
- `s.ordinal()` returns the position, or **ordinal value**, of `DIAMONDS` in the enumeration. The objects within an enumeration are numbered beginning with 0. So in this example, `s.ordinal()` returns 1.
- `s.toString()` returns the string `"DIAMONDS"`. That is, `toString` returns the name of its invoking object as a string.
- `Suit.valueOf("HEARTS")` returns the object `Suit.HEARTS`. The match between the string argument and the name of an enumeration object must be exact. Any whitespace within the string is *not* ignored. The method `valueOf` is a static method, so its name in an invocation is preceded by `Suit` and a dot.

You can define private instance variables and additional public methods—including constructors—for any enumeration. By defining an instance variable, you can assign values to the objects in the enumeration. Adding a `get` method will provide a way to access these values. We have done all of these things in the new definition of the enumeration `Suit` shown in Listing 6.20.

---

#### LISTING 6.20 An Enhanced Enumeration `Suit`

---


```
/** An enumeration of card suits. */
enum Suit
{
    CLUBS("black"), DIAMONDS("red"), HEARTS("red"),
    SPADES("black");

    private final String color;

    private Suit(String suitColor)
    {
        color = suitColor;
    }

    public String getColor()
    {
        return color;
    }
}
```

---



*To understand a name you must be acquainted with the particular of which it is a name.*

—BERTRAND RUSSELL

*Like mother, like daughter*

—COMMON SAYING

---

This chapter covers inheritance, polymorphism, and interfaces, three key concepts in object-oriented programming. These concepts are also needed in order to use many of the libraries that come with the Java programming language. Polymorphism makes objects behave as you expect them to and allows you to focus on the specifications of those behaviors. Inheritance will enable you to use an existing class to define new classes, making it easier to reuse software. Finally, interfaces allow you to specify the methods that a class must implement.

## OBJECTIVES

After studying this chapter, you should be able to

- Describe polymorphism and inheritance in general
- Define interfaces to specify methods
- Describe dynamic binding
- Define and use derived classes in Java
- Understand the role of event handling in a JavaFX application

## PREREQUISITES

You need to have read the material in Chapters 1 through 6 before you can understand the material in this chapter. Chapter 7 is needed to understand some of the examples presented in Sections 8.3 and 8.4.

## 8.1 INHERITANCE BASICS

*Socrates is a person.*

*All people are mortal.*

*Therefore socrates is mortal.*

—TYPICAL SYLLOGISM

Suppose we define a class for vehicles that has instance variables to record the vehicle's number of wheels and maximum number of occupants. The class also has accessor and mutator methods. Imagine that we then define a class for

automobiles that has instance variables and methods just like the ones in the class of vehicles. In addition, our automobile class would have added instance variables for such things as the amount of fuel in the fuel tank and the license plate number and would also have some added methods. Instead of repeating the definitions of the instance variables and methods of the class of vehicles within the class of automobiles, we could use Java's inheritance mechanism, and let the automobile class inherit all the instance variables and methods of the class for vehicles.

**Inheritance** allows you to define a very general class and then later define more specialized classes that add some new details to the existing general class definition. This saves work, because the more specialized class *inherits* all the properties of the general class and you, the programmer, need only program the new features.

Inheritance lets you define specialized classes from a general one

Before we construct an example of inheritance within Java, we first need to set the stage. We'll do so by defining a simple class called `Person`. This class—shown in Listing 8.1—is so simple that the only attribute it gives a person is a

---

#### LISTING 8.1 The Class `Person`

---

```
public class Person
{
    private String name;
    public Person()
    {
        name = "No name yet";
    }
    public Person(String initialName)
    {
        name = initialName;
    }
    public void setName(String newName)
    {
        name = newName;
    }
    public String getName()
    {
        return name;
    }
    public void writeOutput()
    {
        System.out.println("Name: " + name);
    }
    public boolean hasSameName(Person otherPerson)
    {
        return this.name.equalsIgnoreCase(otherPerson.name);
    }
}
```

---

name. We will not have much use for the class `Person` by itself, but we will use it when defining other classes.

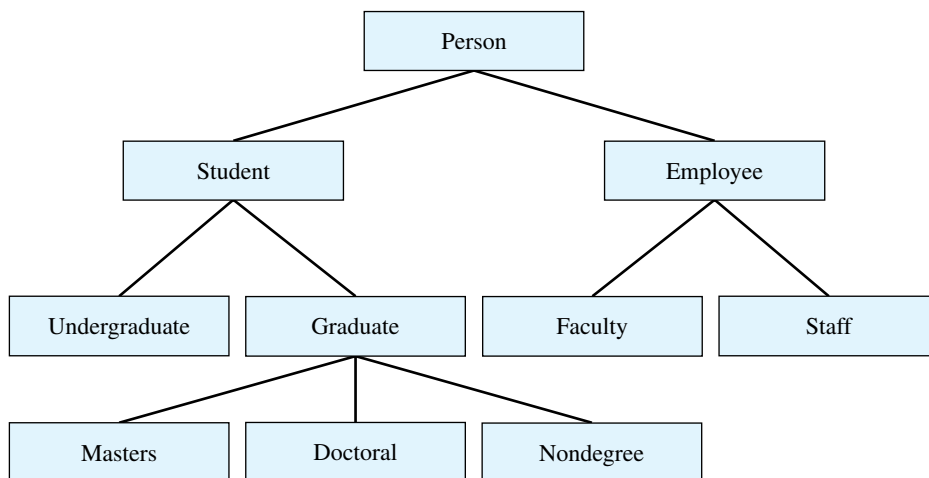
Most of the methods for the class `Person` are straightforward. For example, the method `hasSameName` is similar to the `equals` methods we've seen, but note that it considers uppercase and lowercase versions of a letter to be the same when comparing names.

## Derived Classes

Suppose we are designing a college record-keeping program that has records for students, faculty, and other staff. There is a natural hierarchy for grouping these record types: They are all records of people. Students are one subclass of people. Another subclass is employees, which includes both faculty and staff. Students divide into two smaller subclasses: undergraduate students and graduate students. These subclasses may further subdivide into still smaller subclasses.

Figure 8.1 depicts a part of this hierarchical arrangement. Although your program may not need any class corresponding to people or employees, thinking in terms of such classes can be useful. For example, all people have names, and the methods of initializing, displaying, and changing a name will be the same for student, staff, and faculty records. In Java, you can define a class called `Person` that includes instance variables for the properties that belong to all subclasses of people. The class definition can also contain all the methods that manipulate the instance variables for the class `Person`. In fact, we have already defined such a `Person` class in Listing 8.1.

**FIGURE 8.1** A Class Hierarchy



Listing 8.2 contains the definition of a class for students. A student is a person, so we define the class `Student` to be a **derived class**, or **subclass**, of the class `Person`. A derived class is a class defined by adding instance variables and methods to an existing class. We say that the derived class **extends** the existing class. The existing class that the derived class is built upon is called the **base class**, or **superclass**. In our example, `Person` is the base class and `Student` is the derived class. We indicated this in the definition of `Student` in Listing 8.2 by including the phrase `extends Person` on the first line of the class definition, so that the class definition of `Student` begins

A derived class extends a base class and inherits the base class's public members

```
public class Student extends Person
```

### LISTING 8.2 A Derived Class (part 1 of 2)

```
public class Student extends Person
{
    private int studentNumber;
    public Student()
    {
        super();
        studentNumber = 0; // Indicating no number yet
    }
    public Student(String initialName, int initialStudentNumber)
    {
        super(initialName);
        studentNumber = initialNumber;
    }
    public void reset(String newName, int newStudentNumber)
    {
        setName(newName);
        studentNumber = newStudentNumber;
    }
    public int getStudentNumber()
    {
        return studentNumber;
    }
    public void setStudentNumber(int newStudentNumber)
    {
        studentNumber = newStudentNumber;
    }
    public void writeOutput()
    {
        System.out.println("Name: " + getName());
        System.out.println("Student Number: " + studentNumber);
    }
}
```

*super is explained in a later section. Do not worry about it until you reach the discussion of it in the text.*

(continued)



**LISTING 8.2 A Derived Class** *(part 2 of 2)*

---

```
public boolean equals(Student otherStudent)
{
    return this.hasSameName(otherStudent) &&
           (this.studentNumber == otherStudent.studentNumber);
}
```

---

The class `Student`—like any other derived class—is said to **inherit** the public instance variables and public methods of the base class that it extends. When you define a derived class, you give only the added instance variables and the added methods. Even though the class `Student` has all the public instance variables and all the public methods of the class `Person`, we do not declare or define them in the definition of `Student`. For example, every object of the class `Student` has the method `getName`, but we do not define `getName` in the definition of the class `Student`.

A derived class, such as `Student`, can also add some instance variables or methods to those it inherits from its base class. For example, `Student` defines the instance variable `studentNumber` and the methods `reset`, `getStudentNumber`, `setStudentNumber`, `writeOutput`, and `equals`, as well as some constructors. (We will postpone the discussion of constructors until we finish explaining the other parts of these class definitions.)

Notice that although `Student` does not inherit the private instance variable `name` from `Person`, it does inherit the method `setName` and all the other public methods of the base class. Thus, `Student` has indirect access to `name` and so has no need to define its own version. If `s` is a new object of the class `Student`, defined as

```
Student s = new Student();
```

we could write

```
s.setName("Warren Peace");
```

Because `name` is a private instance variable of `Person`, however, you cannot write `s.name` outside of the definition of the class `Person`, not even within the definition of `Student`. The instance variable exists, however, and it can be accessed and changed using methods defined within `Person`. Listing 8.3 contains a very small demonstration program to illustrate inheritance.

An object of `Student` has all of the methods of `Person` in addition to all of the methods of `Student`. Earlier, we noted that a student is a person. The classes `Student` and `Person` model this real-world relationship in that `Student` has all the behaviors of `Person`. We call this relationship an **is-a relationship**. You should use inheritance only if an is-a relationship exists between a class and a proposed derived class.

Inheritance  
should define a  
natural is-a  
relationship  
between two  
classes

**LISTING 8.3 A Demonstration of Inheritance Using Student**

```
public class InheritanceDemo
{
    public static void main(String[] args)
    {
        Student s = new Student();
        s.setName("Warren Peace");
        s.setStudentNumber(1234);
        s.writeOutput();
    }
}
```

*setName is inherited  
from the class Person.*

**Screen Output**

```
Name: Warren Peace
Student Number: 1234
```

**PROGRAMMING TIP Use Inheritance Only to Model Is-a Relationships**

If an is-a relationship does not exist between two proposed classes, do not use inheritance to derive one class from the other. Instead, consider defining an object of one class as an instance variable within the other class. That relationship is called has-a. The programming tip titled “Is-a and Has-a Relationships” talks more about these two relationships. ■

When discussing derived classes, it is common to use terminology derived from family relationships. A base class is often called a **parent class**. A derived class is then called a **child class**. This makes the language of inheritance very smooth. For example, we can say that a child class inherits public instance variables and public methods from its parent class.

This analogy is often carried one step further. A class that is a parent of a parent of another class (or some other number of “parent of” iterations) is often called an **ancestor class**. If class A is an ancestor of class B, then class B is often called a **descendant** of class A.

A base class is also called a superclass, a parent class, and an ancestor class

**RECAP Derived Class**

You define a derived class, or subclass, by starting with another already defined class and adding (or changing) methods and instance variables. The class you start with is called the base class, or superclass. The derived

*(continued)*

class inherits all of the public methods and public instance variables from the base class and can add more instance variables and methods.

#### SYNTAX

```
public class Derived_Class_Name extends Base_Class_Name
{
    Declarations_of_Added_Instance_Variables
    Definitions_of_Added_And_Changed_Methods
}
```

#### EXAMPLE:

See Listing 8.2.

*As you will see in the next section, changed methods are said to be overridden.*

A derived class is also called a subclass, a child class, and a descendant class

## Overriding Method Definitions

Overriding a method redefines it in a descendant class

The class `Student` in Listing 8.2 defines a method named `writeOutput` that has no parameters. But the class `Person` also has a method by the same name that has no parameters. If the class `Student` were to inherit the method `writeOutput` from the base class `Person`, `Student` would contain two methods with the name `writeOutput`, both of which have no parameters. Java has a rule to avoid this problem. If a derived class defines a method with the same name, the same number and types of parameters, and the same return type as a method in the base class, the definition in the derived class is said to **override** the definition in the base class. In other words, the definition in the derived class is the one that is used for objects of the derived class. For example, the invocation

```
s.writeOutput();
```

in Listing 8.3 will use the definition of `writeOutput` in the class `Student`, not the definition in the class `Person`, since `s` is an object of the class `Student`.

When overriding a method, you can change the body of the method definition to anything you wish, but you cannot make any changes in the method's heading, including its return type.

A method overrides another if both have the same name, return type, and parameter list

#### RECAP Overriding Method Definitions

In a derived class, if you include a method definition that has the same name, the *exact* same number and types of parameters, and the same return type as a method already in the base class, this new definition

*(continued)*

replaces the old definition of the method when objects of the derived class receive a call to the method.

When overriding a method definition, you cannot change the return type of the method. Since the signature of a method does not include the return type, you can say that when one method overrides another, both methods must have the same signature and return type.

## Overriding Versus Overloading

Do not confuse *overriding* a method with *overloading* a method. When you override a method definition, the new method definition given in the derived class has the same name, the same return type, and the exact same number and types of parameters. On the other hand, if the method in the derived class were to have the same name and the same return type but a different number of parameters or a parameter of a different type from the method in the base class, the method names would be overloaded. In such cases, the derived class would have both methods.

For example, suppose we added the following method to the definition of the class `Student` in Listing 8.2:

```
public String getName(String title)
{
    return title + getName();
}
```

In this case, the class `Student` would have two methods named `getName`: It would inherit the method `getName`, with no parameters, from the base class `Person` (Listing 8.1), and it would also have the method named `getName`, with one parameter, that we just defined. This is because the two `getName` methods have different numbers of parameters, and thus the methods use overloading.

If you get overloading and overriding confused, remember this: Overloading places an additional “load” on a method name by using it for another method, whereas overriding replaces a method’s definition.

## The final Modifier

If you want to specify that a method definition cannot be overridden by a new definition within a derived class, you can add the `final` modifier to the method heading, as in the following sample heading:

```
public final void specialMethod()
```

When a method is declared as `final`, the compiler knows more about how it will be used, and so the compiler can generate more efficient code for the method.

A method overloads another if both have the same name and return type but different parameter lists

A final method cannot be overridden

A final class cannot be a base class

An entire class can be declared final, in which case you cannot use it as a base class to derive any other class. You are not very likely to need the final modifier right now, but you will see it in the specifications of some methods in the standard Java libraries.

## ■ PROGRAMMING TIP Constructors That Call Methods

If a constructor calls a public method, a derived class could override that method, thereby affecting the behavior of the constructor. To prevent that from happening, declare such public methods as final. ■

## Private Instance Variables and Private Methods of a Base Class

An object of the derived class Student (Listing 8.2) does not inherit the instance variable name from the base class Person (Listing 8.1), but it can access or change name's value via the public methods of Person. For example, the following statements create a Student object and set the values of the instance variables name and studentNumber:

```
Student joe = new Student();
joe.reset("Joesy", 9892);
```



VideoNote  
Protected instance  
variables and methods

Since the instance variable name is a private instance variable in the definition of the class Person, it cannot be directly accessed by name within the definition of the class Student. Thus, the definition of the method reset in the class Student is

```
public void reset(String newName, int newStudentNumber)
{
    setName(newName);
    studentNumber = newStudentNumber;
}
```

*Valid definition*

It cannot be as follows:

```
public void reset(String newName, int newStudentNumber)
{
    name = newName; // ILLEGAL!
    studentNumber = newStudentNumber;
}
```

*Illegal definition*

As the comment indicates, this assignment will not work, because a derived class does not inherit private instance variables from its base class. Thus, the definition of reset in the class Student uses the method setName to set the name instance variable.

Private instance variables in a base class are not inherited by a derived class; they cannot be referenced directly by name within a derived class

## **GOTCHA** Private Instance Variables Are Not Directly Accessible from Derived Classes

---

A derived class cannot access the private instance variables of its base class directly by name. It knows only about the public behavior of the base class. The derived class is not supposed to know—or care—how its base class stores data. However, an inherited public method may contain a reference to a private instance variable. ■

The fact that a private instance variable of a base class cannot be accessed by name within the definition of a method of a derived class often seems wrong to people. After all, students should be able to change their own names, rather than being told “Sorry, name is a private instance variable of the class Person.” If you are a student, you are also a person. In Java, this is also true; an object of the class `Student` is also an object of the class `Person`. However, the rules regarding the use of private instance variables must be as we have described, or else the private designation would be pointless. If private instance variables of a class were accessible in method definitions of a derived class, whenever you wanted to access a private instance variable, you could simply create a derived class and access it in a method of that class. This would mean that all private instance variables would be accessible to anybody who wanted to put in a little extra effort.

Similarly, private methods in a base class are not directly accessible by name within any other class, not even a derived class. The private methods still exist, however. If a derived class calls an inherited *public* method that contains an invocation of a private method, that invocation still works. However, a derived class cannot define a method that invokes a private method of the base class. This should not be a problem. Private methods should serve only as helping methods, and so their use should be limited to the class in which they are defined. If you want a method to serve as a helping method in a number of derived classes, it is more than just a helping method, and you should make the method public.

Private methods in a base class are not inherited by a derived class; they cannot be called directly by name from a derived class

## **GOTCHA** Private Methods Are Not Directly Accessible from Derived Classes

---

A derived class cannot call a private method defined within the base class. However, the derived class can call a public method that in turn calls a private method when both methods are in the base class. ■

## **PROGRAMMING TIP** Assume That Your Coworkers Are Malicious

The reason private instance variables cannot be accessed by name in a derived class is that otherwise a malicious programmer could access them by using a trick. You may argue that your coworkers are not malicious. In fact, in a

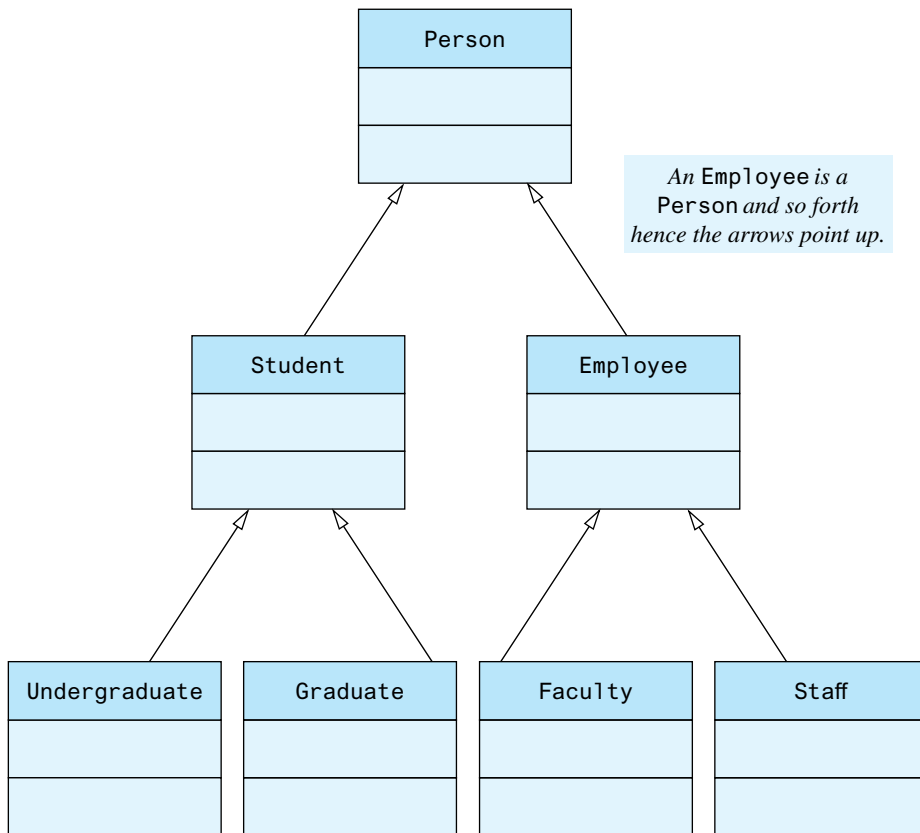
beginning course you may sometimes be the only programmer on an assignment, and you certainly are not trying to sabotage your own work. Those are good points. However, your coworkers—or even you yourself—might inadvertently do something that, although not intended to be malicious, still creates a problem. We think in terms of a malicious programmer not because we think our coworkers are malicious, but because that is the best way to protect against honest mistakes by well-meaning programmers—including you! ■

An arrow points up from a derived class to a base class in a UML diagram

## UML Inheritance Diagrams

Figure 8.2 shows a portion of the class hierarchy given in Figure 8.1, but uses UML notation. Note that the class diagrams are incomplete. You normally show only as much of the class diagram as you need for the design task at hand.

**FIGURE 8.2** A Class Hierarchy in UML Notation

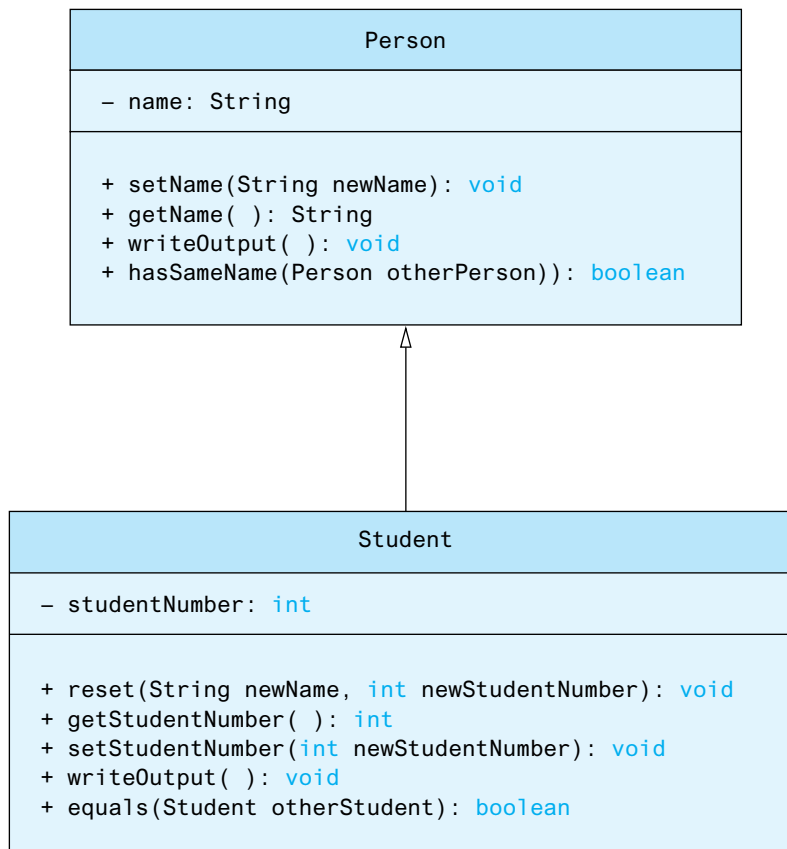


The only significant difference between the notation in Figure 8.2 and that in Figure 8.1 is that the lines indicating inheritance in Figure 8.2 have unfilled arrowheads. Note that the arrowheads point up from the derived class to the base class. These arrows show the is-a relationship. For example, a `Student` is a `Person`. In Java terms, an object of type `Student` is also of type `Person`.

The arrows also help in locating method definitions. If you are looking for a method definition for some class, the arrows show the path you (or the computer) should follow. If you are looking for the definition of a method used by an object of the class `Undergraduate`, you first look in the definition of the class `Undergraduate`; if it is not there, you look in the definition of `Student`; if it is not there, you look in the definition of the class `Person`.

Figure 8.3 shows more details of the inheritance hierarchy for two classes: `Person` and one of its derived classes, `Student`. Suppose `s` references an object

**FIGURE 8.3** Some Details of the UML Class Hierarchy Shown in Figure 8.2





of the class `Student`. The diagram in Figure 8.3 tells you that definition of the method `getStudentNumber` in the call

```
int num = s.getStudentNumber();
```

is found within the class `Student`, but that the definition of `setName` in

```
s.setName("Joe Student");
```

is in the class `Person`.

## SELF-TEST QUESTIONS

1. What is the difference between overriding a method and overloading a method name?
2. Suppose the class `SportsCar` is a derived class of a class `Automobile`. Suppose also that the class `Automobile` has private instance variables named `speed`, `manufacturer`, and `numberOfCylinders`. Will an object of the class `SportsCar` have instance variables named `speed`, `manufacturer`, and `numberOfCylinders`?
3. Suppose the class `SportsCar` is a derived class of a class `Automobile`, and suppose also that the class `Automobile` has public methods named `accelerate` and `addGas`. Will an object of the class `SportsCar` have methods named `accelerate` and `addGas`? If so, do these methods have to perform the exact same actions in the class `SportsCar` as in the class `Automobile`?
4. Can a derived class directly access by name a private instance variable of the base class?
5. Can a derived class directly invoke a private method of the base class?
6. Suppose `s` is an object of the class `Student`. Based on the inheritance diagram in Figure 8.3, where will you find the definition of the method `hasSameName`, used in the following invocation? Explain your answer.

```
Student other = new Student("Joe Student", 777);  
if (s.hasSameName(other))  
    System.out.println("Wow!");
```

7. Suppose `s` is an object of the class `Student`. Based on the inheritance diagram in Figure 8.3, where will you find the definition of the method used in the following invocation? Explain your answer.

```
s.setStudentNumber(1234);
```

## 8.2 PROGRAMMING WITH INHERITANCE

*You do not have to die in order to pass along your inheritance.*

—AD FOR AN ESTATE PLANNING SEMINAR

This section presents some basic programming techniques you need when defining or using derived classes.

### Constructors in Derived Classes

A derived class, such as the class `Student` in Listing 8.2, has its own constructors. It does not inherit any constructors from the base class. A base class, such as `Person`, also has its own constructors. In the definition of a constructor for the derived class, the typical first action is to call a constructor of the base class. For example, consider defining a constructor for the class `Student`. One thing that needs to be initialized is the student's name. Since the instance variable `name` is defined in the definition of `Person`, it is normally initialized by the constructors for the base class `Person`.

Consider the following constructor definition in the derived class `Student` (Listing 8.2):

```
public Student(String initialName, int initialStudentNumber)
{
    super(initialName);
    studentNumber = initialStudentNumber;
}
```

A derived class does not inherit constructors from its base class

Constructors in a derived class invoke constructors from the base class

This constructor uses the reserved word `super` as a method name to call a constructor of the base class. Although the base class `Person` defines two constructors, the invocation

```
super(initialName);
```

is a call to the constructor in that class that has one parameter, a string. Notice that you use the keyword `super`, not the name of the constructor. That is, you do *not* use

```
Person(initialName); //ILLEGAL
```

Use `super` within a derived class as the name of a constructor in the base class (superclass)

#### **FAQ** How can I remember that `super` invokes a constructor in the base class instead of in a derived class?

Recall that another name for a base class is superclass. So `super` invokes the constructor in a class's superclass.

The use of `super` involves some details: It must always be the first action taken in a constructor definition. You cannot use `super` later in the definition.

Any call to `super` must be first within a constructor

If you do not include an explicit call to the base-class constructor in any constructor for a derived class, Java will automatically include a call to the base class's default constructor. For example, the definition of the default constructor for the class `Student` given in Listing 8.2,

```
public Student()
{
    super();
    studentNumber = 0; //Indicating no number yet
}
```

Without `super`, a constructor invokes the default constructor in the base class

is completely equivalent to the following definition:

```
public Student()
{
    studentNumber = 0; //Indicating no number yet
}
```

### RECAP Calling a Base-Class Constructor

When defining a constructor for a derived class, you can use `super` as a name for the constructor of the base class. Any call to `super` must be the first action taken by the constructor.

#### EXAMPLE

```
public Student(String initialName, int initialStudentNumber)
{
    super(initialName);
    studentNumber = initialStudentNumber;
}
```

### GOTCHA Omitting a Call to `super` in a Constructor

When you omit a call to the base-class constructor in any derived-class constructor, the default constructor of the base class is called as the first action in the new constructor. This default constructor—the one without parameters—might not be the one that should be called. Thus, including your own call to the base-class constructor is often a good idea.

For example, omitting `super(initialName)` from the second constructor in the class `Student` would cause `Person`'s default constructor to be invoked. This action would set the student's name to "No name yet" instead of to the string `initialName`. ■

## The `this` Method—Again

Another common action when defining a constructor is to call another constructor in the same class. Chapter 6 introduced you to this idea, using the keyword `this`. Now that you know about `super`, it's clear that you can use `this` and `super` in similar ways.

You can use both `this` and `super` to call a constructor

We can revise the default constructor in the class `Person` (Listing 8.1) to call another constructor in that class by using `this`, as follows:

```
public Person()  
{  
    this("No name yet");  
}
```

In this way, the default constructor calls the constructor

```
public Person(String initialName)  
{  
    name = initialName;  
}
```

thereby setting the instance variable `name` to the string "No name yet".

As with `super`, any use of `this` must be the first action in a constructor definition. Thus, a constructor definition cannot contain both a call using `super` and a call using `this`. What if you want to include both calls? In that case, use `this` to call a constructor that has `super` as its first action.

### **REMEMBER** `this` and `super` Within a Constructor

When used in a constructor, `this` calls a constructor of the same class, but `super` invokes a constructor of the base class.

## Calling an Overridden Method

We just saw how a constructor of a derived class can use `super` as a name for a constructor of the base class. A method of a derived class that overrides (redefines) a method in the base class can use `super` to call the overridden method, but in a slightly different way.

For example, consider the method `writeOutput` for the class `Student` in Listing 8.2. It contains the statement

```
System.out.println("Name: " + getName());
```

to display the name of the `Student`. Alternatively, you could display the name by calling the method `writeOutput` of the class `Person` in Listing 8.1, since the `writeOutput` method for the class `Person` will display the person's name. The only problem is that if you use the method name `writeOutput` within the class

Student, it will invoke the method named `writeOutput` in the class Student. What you need is a way to say “`writeOutput()` as it is defined in the base class.” The way you say that is `super.writeOutput()`. So an alternative definition of the `writeOutput` method for the class Student is the following:

Using `super` as  
an object calls a  
base-class method

```
public void writeOutput()
{
    super.writeOutput(); //Display the name
    System.out.println("Student Number: " + studentNumber);
}
```

If you replace the definition of `writeOutput` in the definition of Student (Listing 8.2) with the preceding definition, the class Student will behave exactly the same as it did before.

### RECAP Calling an Overridden Method

Within the definition of a method of a derived class, you can call an overridden method of the base class by prefacing the method name with `super` and a dot.

#### SYNTAX

```
super.Overridden_Method_Name(Argument_List)
```

#### EXAMPLE

```
public void writeOutput()
{
    super.writeOutput(); //Calls writeOutput in the base
                        //class
    System.out.println("Student Number: " + studentNumber);
}
```

## PROGRAMMING EXAMPLE

### A Derived Class of a Derived Class

You can form a derived class from a derived class. In fact, this is common. For example, we previously derived the class Student (Listing 8.2) from the class Person (Listing 8.1). We now derive a class Undergraduate from Student, as shown in Listing 8.4. Figure 8.4 contains a UML diagram showing the relationship among the classes Person, Student, and Undergraduate.

**LISTING 8.5 A Better equals Method for the Class Student**

---

```
public boolean equals(Object otherObject)
{
    boolean isEqual = false;
    if ((otherObject != null) &&
        (otherObject instanceof Student))
    {
        Student otherStudent = (Student)otherObject;
        isEqual = this.sameName(otherStudent) &&
            (this.studentNumber ==
             otherStudent.studentNumber);
    }
    return isEqual;
}
```

---

`null` can be plugged in for a parameter of type `Object`. The Java documentation says that an `equals` method should return `false` when comparing an object and the value `null`. So that is what we have done.

## 8.3 POLYMORPHISM

*What's in a name? That which we call a rose*

*By any other name would smell as sweet.*

—WILLIAM SHAKESPEARE, *Romeo and Juliet*

Inheritance allows you to define a base class and derive classes from the base class. **Polymorphism** allows you to make changes in the method definition for the derived classes and have those changes apply to the methods written *in the base class*. This all happens automatically in Java, but it is important to understand the process.



VideoNote  
Exploring polymorphism

### Dynamic Binding and Inheritance

Consider a program that uses the `Person`, `Student`, and `Undergraduate` classes as depicted in Figure 8.4. Let's say that we would like to set up a committee that consists of four people who are either students or employees. If we use an array to store the list of committee members, then it makes sense to make the array of type `Person` so it can accommodate any class derived from it. Here is a possible array declaration:

```
Person[] people = new Person[4];
```

Next we might add objects to the array that represent members of the committee. In the example below we have added three objects of type `Undergraduate` and one object of type `Student` (perhaps we don't know if this person is an undergraduate or graduate):

```
people[0] = new Undergraduate("Cotty, Manny", 4910, 1);
people[1] = new Undergraduate("Kick, Anita", 9931, 2);
people[2] = new Student("DeBanque, Robin", 8812);
people[3] = new Undergraduate("Bugg, June", 9901, 4);
```

In this case we are assigning an object of a derived class (either `Student` or `Undergraduate`) to a variable defined as an ancestor of the derived class (`Person`). This is valid because `Person` encompasses the derived classes. In other words, `Student` "is-a" `Person` and `Undergraduate` "is-a" `Person`, so we can assign either one to a variable of type `Person`.

Next, let's output a report containing information about all of the committee members. The report should be as detailed as possible. For example, if a student is an undergraduate, then the report should contain the student's name, student number, and student level. If the student is of type `Student`, then the report should contain the name and student number. Similar details would be expected for employees. The `writeOutput` method contains this detail, but which one is invoked? There are three of them, one defined for `Undergraduate`, `Student`, and `Person`.

If we focus on just `people[0]`, then we can see that it is declared to be an object of type `Person`. If we invoke:

```
people[0].writeOutput();
```

then it is logical to assume that the `writeOutput` method defined in the `Person` object will be invoked. But that is not what happens! Instead, Java recognizes that an object of type `Undergraduate` is stored in `people[0]`. As a result, even though `people[0]` is declared to be of type `Person`, the method associated with the class used to create the object is invoked. This is called **dynamic binding** or **late binding**.

More precisely, when an overridden method is invoked, its action is the one defined in the class used to create the object using the `new` operator. It is not determined by the type of the variable naming the object. A variable of any ancestor class can reference an object of a descendant class, but the object always remembers which method actions to use for every method name. The type of the variable does not matter. What matters is the class name when the object was created.

Returning to our report, the code below could be used to generate it:

```
for (Person p : people)
{
    p.writeOutput();
    System.out.println();
}
```

This code would output:

```
Name: Cotty, Manny  
Student Number: 4910  
Student Level: 1
```

```
Name: Kick, Anita  
Student Number: 9931  
Student Level: 2
```

```
Name: DeBanque, Robin  
Student Number: 8812
```

```
Name: Bugg, June  
Student Number: 9901  
Student Level: 4
```

A complete program is given in Listing 8.6.

---

**LISTING 8.6 A Demo of Polymorphism (part 1 of 2)**

---

```
public class PolymorphismDemo  
{  
    public static void main(String[] args)  
    {  
        Person[] people = new Person[4];  
        people[0] = new Undergraduate("Cotty, Manny", 4910, 1);  
        people[1] = new Undergraduate("Kick, Anita", 9931, 2);  
        people[2] = new Student("DeBanque, Robin", 8812);  
        people[3] = new Undergraduate("Bugg, June", 9901, 4);  
        for (Person p : people)  
        {  
            p.writeOutput();  
            System.out.println();  
        }  
    }  
}
```

*Even though `p` is of type `Person`, the `writeOutput` method associated with `Undergraduate` or `Student` is invoked depending upon which class was used to create the object.*

---

**Screen Output**

```
Name: Cotty, Manny  
Student Number: 4910  
Student Level: 1
```

```
Name: Kick, Anita  
Student Number: 9931  
Student Level: 2
```

(continued)



**LISTING 8.6** A Demo of Polymorphism (*part 2 of 2*)

```
Name: DeBanque, Robin
Student Number: 8812

Name: Bugg, June
Student Number: 9901
Student Level: 4
```

One of the amazing things about polymorphism is it lets us invoke methods that might not even exist yet! For example, assume the program in Listing 8.6 exists and runs with only the `Person`, `Student`, and `Undergraduate` classes defined. At some later date we could write the `Employee`, `Faculty`, and `Staff` classes as depicted in Figure 8.1. Since all of these classes would be derived from the `Person` class, as long as each implements a `writeOutput` method then we could add one of these objects to the array and its `writeOutput` method would be invoked in the `for` loop. We wouldn't even need to recompile the `PolymorphismDemo` class in Listing 8.6 to invoke the new methods via dynamic binding.

**ASIDE** Java Assumes Dynamic Binding

In many other languages, you must specify in advance what methods may need dynamic binding. Java always assumes that dynamic binding will occur. Although making this assumption is less efficient, Java is easier to program and less prone to errors as a result.

**REMEMBER** Objects Know How They Are Supposed to Act

When an overridden method is invoked, its action is the one defined in the class used to create the object using the `new` operator. It is not determined by the type of the variable naming the object. A variable of any ancestor class can reference an object of a descendant class, but the object always remembers which method actions to use for every method name. The type of the variable does not matter. What matters is the class name when the object was created. This is because Java uses dynamic binding.

**Dynamic Binding with `toString`**

If you include an appropriate `toString` method in the definition of a class, then you can output an object of the class using `System.out.println`. For example, in the previous section we described adding a `toString` method to the `Student` class:

```
public String toString()
{
    return "Name: " + getName() +
           "\nStudent number: " + studentNumber;
}
```

For the `Student` object `joe`, we can invoke the method with the statement

```
System.out.println(joe.toString());
```

However, we can get the exact same result without the `toString`:

```
System.out.println(joe);
```

This happens because Java uses dynamic binding with the `toString` method. The various `println` methods that belong to the object `System.out` were written long before we defined the class `Student`. Yet the invocation calls the definition of `toString` in the class `Student`, not the definition of `toString` in the class `Object`, because `joe` references an object of type `Student`. Dynamic binding is what makes this work. Because `System.out.println` invokes `toString` in this manner, always defining a suitable `toString` method for your classes is a good idea.

### RECAP Dynamic Binding and Polymorphism

With dynamic, or late, binding the definition of a method is not bound to an invocation of the method until run time when the method is called. Polymorphism refers to the ability to associate many meanings to one method name through the dynamic binding mechanism. Thus, polymorphism and dynamic binding are really the same topic.

## SELF-TEST QUESTIONS

18. What is polymorphism?
19. What is dynamic binding? What is late binding? Give an example of each.
20. Is overloading a method name an example of polymorphism?

21. In the following code, will the two invocations of `writeOutput` produce the same output on the screen or not? (The relevant classes are defined in Listings 8.1, 8.2, and 8.4.)

```
Person person = new Student("Sam", 999);  
person.writeOutput();  
person = new Undergraduate("Sam", 999, 1);  
person.writeOutput();
```

22. In the following code, which definition of `writeOutput` is invoked? (The classes are defined in the previous case study.)

```
Undergraduate ug = new Undergraduate("Sam", 999, 1);  
Person p = (Person) ug;  
p.writeOutput();
```

## 8.4 INTERFACES AND ABSTRACT CLASSES

*Art, it seems to me, should simplify. That, indeed, is very nearly the whole of the higher artistic process; finding what conventions of form and what details one can do without and yet preserve the spirit of the whole . . .*

—WILLA CATHER, *ON The Art of Fiction*

Chapter 5 defined a class interface as the portion of a class that tells a programmer how to use it. In particular, a class interface consists of the headings for the public methods and public named constants of the class, along with any explanatory comments. Knowing only a class's interface—that is, the specifications of its public methods—a programmer can write code that uses the class. The programmer does not need to know anything about the class's implementation.

Until now, we have integrated a class's interface into its definition. Java, however, enables you to write an interface and to store it in its own file, separate from the implementation file. Let's further examine the idea of a class interface and see how this concept translates to a Java interface.

### Class Interfaces

In Chapter 1, we imagined a person calling her pets to dinner by whistling. Each animal responded in its own way: Some ran, some flew, and some swam. Let's specify some behaviors for these pets. For example, suppose our pets are able to

- Be named
- Eat
- Respond to a command

We could specify the following method headings for these behaviors:

- */\*\* Sets a pets name to petName. \*/*  
`public void setName(String petName)`
- */\*\* Returns true if a pet eats the given food.\*/*  
`public boolean eat(String food)`
- */\*\* Returns a description of a pet's response to the given command. \*/*  
`public String respond(String command)`

An example of a  
class interface

These method headings can form a class interface.

Now imagine that each of the three classes `Dog`, `Bird`, and `Fish` implements all of these methods. The objects of these classes then have the same behaviors—that is, each object can be named, can eat, and can respond. The nature of these behaviors, however, can be different among the objects.

Although dogs, birds, and fish respond to a command, for example, the way they respond differs.

Imagine a Java statement such as

```
String response = myPet.respond("Come!");
```

This statement is legal regardless of whether `myPet` names a `Dog` object, a `Bird` object, or a `Fish` object. The value of the string `response`, however, differs according to the type of object that `myPet` names. We can substitute one type of object for the other with no problem, as long as each of the three classes implements the method `respond` in its own way. How can we be sure that a class implements certain methods? Read on.

## Java Interfaces

A **Java interface** is a program component that contains the headings for a number of public methods. Some interfaces describe all the public methods in a class, while others specify only certain methods. An interface also can define public named constants. In addition, an interface should include comments that describe the methods, so a programmer will have the necessary information to implement them. In this way, a class designer can specify methods for other programmers. In fact, the Java Class Library contains interfaces that are already written for your use, but you can also define your own.

The class interface that we wrote in the previous section is almost in the form necessary for a Java interface. A Java interface begins like a class definition, except that you use the reserved word `interface` instead of `class`. That is, an interface begins with

```
public interface Interface_Name
```

rather than

```
public class Class_Name
```

The interface can contain any number of public method headings, each followed by a semicolon. For example, Listing 8.7 contains a Java interface for objects whose methods return their perimeters and areas.

By convention, an interface name begins with an uppercase letter, just as class names do. You store an interface in its own file, using a name that begins with the name of the interface, followed by `.java`. For example, the interface in Listing 8.7 is in the file `Measurable.java`. This interface provides a programmer with a handy summary of the methods' specifications. The programmer should be able to use these methods given only the information in the interface, without looking at the method bodies.

An interface does not declare any constructors for a class. Methods within an interface must be public, so you can omit `public` from their headings. An interface can also define any number of public named constants. It contains no instance variables, however, nor any complete method definitions—that is, methods cannot have bodies.

You name and store an interface as you would a class

An interface has no instance variables, constructors, or method definitions

### LISTING 8.7 A Java Interface

```
/**
 * An interface for methods that return
 * the perimeter and area of an object.
 */
public interface Measurable
{
    /** Returns the perimeter. */
    public double getPerimeter();
    /** Returns the area. */
    public double getArea();
}
```

*Do not forget the semicolons at the end of the method headings.*

#### RECAP Java Interfaces

##### SYNTAX

```
public interface Interface_Name
{
    Public_Named_Constant_Definitions
    ...
    Public_Method_Heading_1;
    ...
    Public_Method_Heading_n;
}
```

##### EXAMPLE

```
/**
 * An interface of static methods to convert measurements
 * between feet and inches.
 */
public interface Convertible
{
    public static final int INCHES_PER_FOOT = 12;
    public static double convertToInches(double feet);
    public static double convertToFeet(double inches);
}
```

## Implementing an Interface

When you write a class that defines the methods declared in an interface, we say that the class **implements the interface**. A class that implements an interface must define a body for every method that the interface specifies. It

A class that implements an interface defines each specified method

might also define methods not declared in the interface. That is, an interface need not declare every method defined in a class. In addition, a class can implement more than one interface.

To implement an interface, a class must do two things:

1. Include the phrase

```
implements Interface_Name
```

at the start of the class definition. To implement more than one interface, just list all the interface names, separated by commas, as in

```
implements MyInterface, YourInterface
```

2. Define each method declared in the interface(s).

In this way, a programmer can guarantee—and indicate to other programmers—that a class defines certain methods. Additionally, recall that Java does not allow a class to be derived from multiple parent classes. However, a class can implement multiple interfaces. This is a way to capture some of the behavior that would be possible with multiple inheritance.

For example, to implement the interface `Measurable` shown in Listing 8.7, a class `Rectangle` must begin as follows:

```
public class Rectangle implements Measurable
```

The class must also implement the two methods `getPerimeter` and `getArea`. A full definition of the class `Rectangle` is given in Listing 8.8.

Other classes, such as the class `Circle` shown in Listing 8.9, can implement the interface `Measurable`. Notice that `Circle` defines the method `getCircumference` in addition to the methods declared in the interface. It isn't unusual for a class to define two methods that perform the same task. Doing so provides a convenience for programmers who use the class but prefer a more familiar name for a particular method. Notice, however, that `getCircumference` calls `getPerimeter` instead of performing its own calculation. Doing so makes the class easier to maintain. For example, if we ever discovered a problem with the statements in `getPerimeter`, fixing it would also fix `getCircumference`.

### REMEMBER Interfaces Help Designers and Programmers

Writing an interface is a way for a class designer to specify methods for another programmer. Implementing an interface is a way for a programmer to guarantee that a class defines certain methods.

**LISTING 8.8 An Implementation of the Interface Measurable**

---

```
/**
 * A class of rectangles.
 */
public class Rectangle implements Measurable
{
    private double myWidth;
    private double myHeight;
    public Rectangle(double width, double height)
    {
        myWidth = width;
        myHeight = height;
    }
    public double getPerimeter()
    {
        return 2 * (myWidth + myHeight);
    }
    public double getArea()
    {
        return myWidth * myHeight;
    }
}
```

---

**REMEMBER Several Classes Can Implement the Same Interface**

Different classes can implement the same interface, perhaps in different ways. For example, many classes can implement the interface `Measurable` and provide their own version of the methods `getPerimeter` and `getArea`.

## An Interface as a Type

An interface is a reference type. Thus, you can write a method that has a parameter of an interface type, such as a parameter of type `Measurable`. For example, suppose that your program defines the following method:

An interface is a reference type

```
public static void display(Measurable figure)
{
    double perimeter = figure.getPerimeter();
    double area = figure.getArea();
    System.out.println("Perimeter = " + perimeter +
        "; area = " + area);
}
```

Your program can invoke this method, passing it an object of any class that implements the interface `Measurable`.

### LISTING 8.9 Another Implementation of the Interface Measurable

---

```

/**
 * A class of circles.
 */
public class Circle implements Measurable
{
    private double myRadius;
    public Circle(double radius)
    {
        myRadius = radius;
    }
    public double getPerimeter()
    {
        return 2 * Math.PI * myRadius;
    }
    public double getCircumference()
    {
        return getPerimeter();
    }
    public double getArea()
    {
        return Math.PI * myRadius * myRadius;
    }
}

```

---

*This method is not declared in the interface.*

*Calls another method instead of repeating its body*

For instance, your program might contain the following statements:

```

Measurable box = new Rectangle(5.0, 5.0);
Measurable disc = new Circle(5.0);

```

Even though the type of both variables is `Measurable`, the objects referenced by `box` and `disc` have different definitions of `getPerimeter` and `getArea`. The variable `box` references a `Rectangle` object; `disc` references a `Circle` object. Thus, the invocation

```
display(box);
```

displays

```
Perimeter = 20.0; area = 25.0
```

while the invocation

```
display(disc);
```

displays

```
Perimeter = 31.4; area = 78.5
```

The classes `Rectangle` and `Circle` implement the same interface, so we are able to substitute an instance of one for an instance of the other when we call the method `display`. This is another example of polymorphism—the ability



to substitute one object for another using dynamic binding. These terms refer to the fact that the method invocation is not bound to the method definition until the program executes.

As another example, consider the following code:

```
Measurable m;  
Rectangle box = new Rectangle(5.0, 5.0);  
m = box;  
display(m);  
Circle disc = new Circle(5.0);  
m = disc;  
display(m);
```

The two calls to `display` are identical, and the code within the method `display` is identical in both cases. Thus, the invocations of `getPerimeter` and `getArea` within `display` are identical. Yet these invocations use different definitions for `getPerimeter` and `getArea`, and so the two invocations of `display` produce different output, just as they did in our earlier example.

A variable of an interface type can reference an object of a class that implements the interface, but the object itself always determines which method actions to use for every method name. The type of the variable does not matter. What matters is the class name when the object was created, because Java uses dynamic binding. Not even a type cast will fool Java.

You therefore need to be aware of how dynamic binding interacts with the Java compiler's type checking. For example, consider

```
Measurable m = new Circle(5.0);
```

We can assign an object of type `Circle` to a variable of type `Measurable`, again because `Circle` implements `Measurable`. However, we can use the variable to invoke only a method that is in the interface `Measurable`. Thus, the invocation of `getCircumference` in

```
System.out.println(m.getCircumference()); //ILLEGAL!
```

is illegal, because `getCircumference` is not the name of a method in the `Measurable` interface. In this invocation, the variable `m` is of type `Measurable`, but the object referenced by `m` is still an object of type `Circle`. Thus, although the object has the method `getCircumference`, the compiler does not know this! To make the invocation valid, you need a type cast, such as the following:

```
Circle c = (Circle)m;  
System.out.println(c.getCircumference()); //Legal
```

Objects having  
the same  
interface  
can be used  
interchangeably

### REMEMBER What Is Legal and What Happens

A variable's type determines what method names can be used, but the object the variable references determines which definition of the method will be used.



VideoNote  
Exploring interfaces

### RECAP Dynamic Binding and Polymorphism Apply to Interfaces

Dynamic binding applies to interfaces just as it does with classes. The process enables objects of different classes to substitute for one another, if they have the same interface. This ability—called polymorphism—allows different objects to use different method actions for the same method name.

## SELF-TEST QUESTIONS

23. Imagine a class `Oval` that defines the methods `getPerimeter` and `getArea` but does not have the clause `implements Measurable`. Could you pass an instance of `Oval` as an argument to the method `display` given in the previous section?
24. Can a class implement more than one interface?

## Extending an Interface

Once you have an interface, you can define another interface that builds on, or **extends**, the first one by using a kind of inheritance. Thus, you can create an interface that consists of the methods in an existing interface plus some new methods.

For example, consider the classes of pets we discussed earlier and the following interface:

```
public interface Nameable
{
    public void setName(String petName);
    public String getName();
}
```

We can extend `Nameable` to create the interface `Callable`:

```
public interface Callable extends Nameable
{
    public void come(String petName);
}
```

A class that implements `Callable` must implement the methods `come`, `setName`, and `getName`.

You also can combine several interfaces into a new interface and add even more methods if you like. For example, suppose that in addition to the previous two interfaces, we define the following interfaces:

You can define an interface based on another interface

```

public interface Capable
{
    public void hear();
    public String respond();
}
public interface Trainable extends Callable, Capable
{
    public void sit();
    public String speak();
    public void lieDown();
}

```

A class that implements `Trainable` must implement the methods `setName`, `getName`, `come`, `hear`, and `respond`, as well as the methods `sit`, `speak`, and `lieDown`.

## SELF-TEST QUESTIONS

25. Suppose a class `C` implements the interface `Trainable`, as defined in the previous section. Can you pass an instance of `C` to a method whose parameter is of type `Capable`?
26. Suppose a class `D` implements the interfaces `Callable` and `Capable`, as defined in the previous section. Can you pass an instance of `D` to a method whose parameter is of type `Trainable`?

## CASE STUDY Character Graphics

Java has methods to draw graphics on your computer screen. Suppose, however, that the screen on the inexpensive device you are designing for has no graphics capability, allowing only text output. In this case study, we will design three interfaces and three classes that produce graphics on a screen by placing ordinary keyboard characters on each line to draw simple shapes. Our drawings will not be sophisticated, but we will be able to explore the use of interfaces and inheritance in solving a problem.

Let's begin by writing an interface that specifies the methods that our objects should have. Suppose the method `drawHere` draws the shape beginning at the current line and `drawAt` draws it after moving a given number of lines down from the current one.

All shapes have some properties in common. For example, each of the shapes will have an offset telling how far it is indented from the left edge of the screen. We can include `set` and `get` methods for this offset. Each shape will also have a size, but the size of some shapes is described by a single number, while the size of others is determined by several numbers. Since the size will

Specify the solution by writing an interface

An **exception** is an object that signals the occurrence of an unusual event during the execution of a program. The process of creating this object—that is, generating an exception—is called **throwing an exception**. You place the code that deals with the exceptional case at another place in your program—perhaps in a separate class or method. The code that detects and deals with the exception is said to **handle the exception**.

An exception signals an unusual event during execution

Using exceptions is perhaps most important when a method has a special case that some programs will treat in one way, but others will treat in another way. As you will see, such a method can throw an exception if the special case occurs. This allows the special case to be handled outside of the method in a way that is appropriate to the situation. For example, if a division by zero occurs in a method, it may turn out that, for some invocations of the method, the program should end, but for other invocations of the method, something else should happen.

Java code can handle an exception that is thrown

## Exceptions in Java

Most short programs need very little, if any, exception handling, and the exception handling they do use is often not easy to see in the code. For simplicity, we will use a toy program for our initial example. First we will write the program without using Java's exception-handling facilities, and then we will rewrite it using exception handling.

For this example, suppose that milk is such an important item in our society that people almost never allow themselves to run out of it, but still we would like our programs to accommodate the very unlikely situation of running out of milk. The basic code, which assumes that we do not run out of milk, might be as follows:

```
System.out.println("Enter number of donuts:");
int donutCount = keyboard.nextInt();

System.out.println("Enter number of glasses of milk:");
int milkCount = keyboard.nextInt();

double donutsPerGlass = donutCount / (double)milkCount;
System.out.println(donutCount + " donuts.");
System.out.println(milkCount + " glasses of milk.");
System.out.println("You have " + donutsPerGlass +
    " donuts for each glass of milk.");
```

If the number of glasses of milk entered is zero—that is, if we have no milk—this code will produce a division by zero. To take care of the unlikely scenario in which we run out of milk and get an erroneous division, we can add a test for this unusual situation. The complete program with this added test for the special situation is shown in Listing 9.1.

Now let's revise this program, using Java's exception-handling facilities. A division by zero produces an exception. So instead of avoiding such a division, we can let it occur but react to the resulting exception, as we have done in Listing 9.2. Because this example is so simple, we probably would not actually

**LISTING 9.1 One Way to Deal with a Problem Situation**

---

```
import java.util.Scanner;

public class GotMilk
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);

        System.out.println("Enter number of donuts:");
        int donutCount = keyboard.nextInt();

        System.out.println("Enter number of glasses of milk:");
        int milkCount = keyboard.nextInt();

        //Dealing with an unusual event without Java's exception
        //handling features:
        if (milkCount < 1)
        {
            System.out.println("No milk!");
            System.out.println("Go buy some milk.");
        }
        else
        {
            double donutsPerGlass = donutCount / (double)milkCount;
            System.out.println(donutCount + " donuts.");
            System.out.println(milkCount + " glasses of milk.");
            System.out.println("You have " + donutsPerGlass +
                               " donuts for each glass of milk.");
        }
        System.out.println("End of program.");
    }
}
```

---

**Sample Screen Output**

```
Enter number of donuts:
2
Enter number of glasses of milk:
0
No milk!
Go buy some milk.
End of program.
```

---

use exception handling here. The revised program as a whole is certainly not simpler than the original one in Listing 9.1. However, the code between the words `try` and `catch` is cleaner than the code used in the original program and

hints at the advantage of handling exceptions. A clear organization and structure are essential for large, complex programs.

The code in Listing 9.2 is basically the same as the code in Listing 9.1, except that instead of the big `if-else` statement, this program has the following smaller `if` statement:

```
if (milkCount < 1)
    throw new Exception("Exception: No milk!");
```

This `if` statement says that if we have no milk, the program should do something exceptional. That something exceptional is given after the word

### LISTING 9.2 An Example of Exception Handling (part 1 of 2)

	<pre>import java.util.Scanner;  public class ExceptionDemo {     public static void main(String[] args)     {         Scanner keyboard = new Scanner(System.in);          try         {             System.out.println("Enter number of donuts:");             int donutCount = keyboard.nextInt();              System.out.println("Enter number of glasses of milk:");             int milkCount = keyboard.nextInt();              if (milkCount &lt; 1)                 throw new Exception("Exception: No milk!");              double donutsPerGlass = donutCount / (double)milkCount;             System.out.println(donutCount + " donuts.");             System.out.println(milkCount + " glasses of milk.");             System.out.println("You have " + donutsPerGlass +                                " donuts for each glass of milk.");         }          catch(Exception e)         {             System.out.println(e.getMessage());             System.out.println("Go buy some milk.");         }          System.out.println("End of program.");     } }</pre>	<p><i>This program is just a simple example of the basic syntax for exception handling.</i></p>
--	--	---

*try block*

*catch block*

(continued)

**LISTING 9.2** An Example of Exception Handling (*part 2 of 2*)*Sample Screen Output 1*

```

Enter number of donuts:
3
Enter number of glasses of milk:
2
3 donuts.
2 glasses of milk.
You have 1.5 donuts for each glass of milk.
End of program.

```

*Sample Screen Output 1*

```

Enter number of donuts:
2
Enter number of glasses of milk:
0
Exception: No milk!
Go buy some milk.
End of program.

```

catch. The idea is that the normal situation is handled by the code following the word `try` and that the code following the word `catch` is used only in exceptional circumstances. Let's look at the details.

The basic way of handling exceptions in Java consists of a `try`-`throw`-`catch` threesome. A `try` **block** has the syntax

```

try
{
    Code_To_Try
}

```

A `try` block contains the code for the basic algorithm when everything goes smoothly. It is called a `try` block because you are not 100 percent sure that all will go smoothly, but you want to give it a try.

If something does go wrong, you want to throw an exception, which is a way of indicating that there is some sort of problem. So the basic outline, when we add a `throw` **statement**, is as follows:

```

try
{
    Code_To_Try
    Possibly_Throw_An_Exception
    More_Code
}

```

A `try` block  
detects an  
exception

The `try` block in Listing 9.2 has this form and contains the statement

```
throw new Exception("Exception: No milk!");
```

If this `throw` statement executes, it creates a new object of the predefined class `Exception` with the expression

```
new Exception("Exception: No milk!")
```

and **throws** the exception object. The string `"Exception: No milk!"` is an argument for the constructor of the class `Exception`. The `Exception` object created here stores this string in an instance variable of the object so that it can be recovered later.

When an exception is thrown, the code in the surrounding block stops execution, and another portion of code, known as a **catch block**, begins execution. Executing the catch block is called **catching the exception**. When an exception is thrown, it should ultimately be caught by some catch block. In Listing 9.2, the catch block immediately follows the `try` block.

The catch block looks a little like a method definition that has a parameter. Although it is not a method definition, a catch block behaves like a method in some ways. It is a separate piece of code that is executed when a program executes a `throw` statement from within the preceding `try` block. This `throw` statement is similar to a method call, but instead of calling a method, it calls the catch block and causes the code in the catch block to be executed.

Let's examine the first line of the catch block in Listing 9.2:

```
catch(Exception e)
```

The identifier `e` looks like a parameter and acts very much like a parameter. So even though the catch block is not a method, we call this `e` the **catch-block parameter**. The catch-block parameter gives us a name for the exception that is caught, so that we can write code within the catch block to manipulate the exception object. The most common name for a catch-block parameter is `e`, but you can use any legal identifier.

When an exception object is thrown, it is plugged in for the catch-block parameter `e`, and the code in the catch block is executed. So in this case, you can think of `e` as the name of the exception object that was thrown. Every exception object has a method called `getMessage`, and unless you provide code specifying otherwise, this method retrieves the string that was given to the exception object by its constructor when the exception was thrown. In our example, the invocation `e.getMessage()` returns `"Exception: No milk!"`. Thus, when the catch block in Listing 9.2 executes, it writes the following lines to the screen:

```
Exception: No milk!
Go buy some milk.
```

The class name preceding the catch-block parameter specifies what kind of exception the catch block can catch. The class `Exception` in our example indicates that this catch block can catch an exception of type `Exception`. Thus, a thrown exception object must be of type `Exception` in order for this particular

A `throw` statement throws an exception

A catch block deals with a particular exception

An exception's `getMessage` method returns a description of the exception



Only one catch  
block executes  
per try block

catch block to apply. As you will see, other types of exceptions are possible, and several catch blocks—one for each type of exception to be handled—can follow a try block. Only one of these catch blocks—the one corresponding to the type of exception thrown—can execute, however. The rest are ignored when handling a particular exception. Since all exceptions are of type `Exception`, the solitary catch block in our example will catch any exception. Although this might sound like a good idea, it generally is not. Several specific catch blocks are usually preferable to one general one.

For the program in Listing 9.2, when the user enters a positive number for the number of glasses of milk, no exception is thrown. The flow of control in this case is shown in Listing 9.3. Listing 9.4 shows the flow of control when an exception is thrown because the user enters zero or a negative number for the number of glasses of milk.

### LISTING 9.3 Flow of Control When No Exception Is Thrown

```
import java.util.Scanner;

public class ExceptionDemo
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);

        try
        {
            System.out.println("Enter number of donuts:");
            int donutCount = keyboard.nextInt();
            System.out.println("Enter number of glasses of milk:");
            int milkCount = keyboard.nextInt();
            if (milkCount < 1)
                throw new Exception("Exception: No milk!");
            double donutsPerGlass = donutCount / (double)milkCount;
            System.out.println(donutCount + " donuts.");
            System.out.println(milkCount + " glasses of milk.");
            System.out.println("You have " + donutsPerGlass
                               + " donuts for each glass of milk.");
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
            System.out.println("Go buy some milk.");
        }
        System.out.println("End of program.");
    }
}
```

Here we assume that the user enters a positive number for the number of glasses of milk.

*milkCount is positive, so an exception is NOT thrown here.*

*This code is NOT executed.*

**LISTING 9.4** Flow of Control When an Exception Is Thrown

---

```

import java.util.Scanner;
public class ExceptionDemo
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        try
        {
            System.out.println("Enter number of donuts:");
            int donutCount = keyboard.nextInt();

            System.out.println("Enter number of glasses of milk:");
            int milkCount = keyboard.nextInt();

            if (milkCount < 1)
                throw new Exception("Exception: No milk!");

            double donutsPerGlass = donutCount / (double)milkCount;
            System.out.println(donutCount + " donuts.");
            System.out.println(milkCount + " glasses of milk.");
            System.out.println("You have " + donutsPerGlass
                               + " donuts for each glass of milk.");
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
            System.out.println("Go buy some milk.");
        }
        System.out.println("End of program.");
    }
}

```

---

*Here we assume that the user enters zero for the number of glasses of milk, and so an exception is thrown.*

*milkCount is zero or negative, so an exception IS thrown here.*

*This code is NOT executed.*

In summary, a try block contains some code that can throw an exception. It is able to throw an exception because it either includes a throw statement or, as you will see later, invokes another method that contains a throw statement. The throw statement is normally executed only in exceptional circumstances, but when it is executed, it throws an exception of some exception class. (So far, Exception is the only exception class we have discussed, but we will talk about others soon.) When an exception is

If an exception occurs within a try block, the rest of the block is ignored

If no exception occurs within a try block, the catch blocks are ignored

Comparing try-throw-catch and if-else

thrown, execution of the try block ends. All the rest of the code in the try block is ignored, and control passes to a suitable catch block, if one exists. A catch block applies only to an immediately preceding try block. If an exception is thrown and caught, that exception object is plugged in for the catch-block parameter and the statements in the catch block are executed. After the catch-block code is executed, the program proceeds with the code after the last catch block; it does not return to the try block. So none of the try-block code after the statement that threw the exception is executed, as shown in Listing 9.4. (Later, we will discuss what happens when there is no appropriate catch block.)

Now let's look at what happens when no exception is thrown in a try block. When the try block executes normally to completion, without throwing an exception, program execution continues with the code after the last catch block. In other words, if no exception is thrown, all associated catch blocks are ignored, as shown in Listing 9.3.

This explanation makes it seem as though a try-throw-catch sequence is equivalent to an if-else statement. They are almost equivalent, except for the message carried by the thrown exception. An if-else statement cannot send a message to one of its branches. This may not seem like much of a difference, but as you will see, the ability to send a message gives the exception-handling mechanism more versatility than an if-else statement.

### REMEMBER An Exception Is an Object

A statement, such as

```
throw new Exception("Illegal character.");
```

does not just specify some action that is taken and forgotten. It creates an object that has a message. In this example, the message is "Illegal character."

To see this, note that the previous statement is equivalent to the following:

```
Exception exceptObject = new Exception  
    ("Illegal character.");  
throw exceptObject;
```

The first of these statements invokes a constructor of the class `Exception`, creating an object of the class `Exception`. The second statement throws this exception object. This object and its message are available to a catch block, and so the effect of throwing an exception is more than just a transfer of control to the first statement of a catch block.

**RECAP Throwing Exceptions**

The throw statement throws an exception.

**SYNTAX**

```
throw new Exception_Class_Name(Possibly_Some_Arguments);
```

A throw statement is usually embedded in an if statement or an if-else statement. A try block can contain any number of explicit statements or any number of method invocations that might throw exceptions.

**EXAMPLE**

```
throw new Exception("Unexpected End of Input.");
```

**RECAP Handling Exceptions**

The try and catch statements, used together, are the basic mechanism for handling exceptions.

```
try
{
    Code_To_Try
    Possibly_Throw_An_Exception
    More_Code
}
catch (Exception_Class_Name Catch_Block_Parameter)
{
    Process_Exception_Of_Type_Exception_Class_Name
}
Possibly_Other_Catch_Blocks
```

If an exception is thrown within the try block, the rest of the try block is ignored and execution continues with the first catch block that matches the type of the thrown exception. After the catch block is completed, any code after the last catch block is executed.

If no exception is thrown in the try block, after it completes execution, program execution continues with the code after the last catch block. In other words, if no exception is thrown, the catch blocks are ignored.

More than one block is allowed for each try block, but each catch block can handle only one class of exceptions. *Catch\_Block\_Parameter* is an identifier that serves as a place holder for an exception that might

(continued)

be thrown. When an exception of the class *Exception\_Class\_Name* is thrown in the preceding try block, that exception is plugged in for the *Catch\_Block\_Parameter*. The code in the catch block may refer to the *Catch\_Block\_Parameter*. A common choice for *Catch\_Block\_Parameter* is *e*; however, you may use any legal identifier.

### RECAP The getMessage Method

Every exception object has a `String` instance variable that contains some message, which typically identifies the reason for the exception. For example, if the exception is thrown by the statement

```
throw new Exception(String_Argument);
```

the value of this `String` instance variable is *String\_Argument*. If the exception object is called *e*, the invocation `e.getMessage()` returns this string.

## SELF-TEST QUESTIONS

1. What output is produced by the following code?

```
int waitTime = 46;
try
{
    System.out.println("Try block entered.");
    if (waitTime > 30)
        throw new Exception("Time Limit Exceeded.");
    System.out.println("Leaving try block.");
}
catch (Exception e)
{
    System.out.println("Exception: " + e.getMessage());
}
System.out.println("After catch block");
```

2. What output would the code in the previous question produce if we changed 46 in the first statement to 12?
3. What is an exception? Is it an identifier? A variable? A method? An object? A class? Something else?
4. Is the following statement legal?

```
Exception myException = new Exception("Hi Mom!");
```

and do need to be imported. For example, the class `IOException` is in the package `java.util`. When you examine the documentation for an exception, note which package contains it so you can provide an `import` statement if necessary. ■

## SELF-TEST QUESTIONS

13. Are the following statements legal?

```
IOException sos = new IOException("Hello Houston!");
throw sos;
```

14. Is the following catch block legal?

```
catch(NoSuchMethodException exception)
{
    System.out.println(exception.getMessage());
    System.exit(0);
}
```

## 9.2 DEFINING YOUR OWN EXCEPTION CLASSES

*I'll make an exception this time.*

—MY MOTHER

You can define your own exception classes, but they must be derived classes of some already defined exception class. An exception class can be a derived class of any predefined exception class or of any exception class that you have already successfully defined. Our examples will be derived classes of the class `Exception`.

When defining an exception class, the constructors are the most important and often the only methods, other than those inherited from the base class. For example, Listing 9.5 contains an exception class, called `DivideByZeroException`, whose only methods are a default constructor and a constructor having one `String` parameter. For our purposes, that is all we need to define. However, the class does inherit all the methods of the base class `Exception`. In particular, the class `DivideByZeroException` inherits the method `getMessage`, which returns a string message. In the definition of the default constructor, this string message is set by the following statement:

```
super("Dividing by Zero!");
```

This statement calls a constructor of the base class `Exception`. As we have already noted, when you pass a string to the constructor of the class `Exception`, the value of a `String` instance variable is set. You can recover this value later

When you define an exception class, you typically define only constructors

Call the base-class constructor and pass it a message

**LISTING 9.5    A Programmer-Defined Exception Class**

---

```

public class DivideByZeroException extends Exception
{
    public DivideByZeroException()
    {
        super("Dividing by Zero!");
    }
    public DivideByZeroException(String message)
    {
        super(message);
    }
}

```

*You can do more in an exception constructor, but this form is common.*

*super is an invocation of the constructor for the base class Exception.*

by calling the method `getMessage`, which is an ordinary accessor method of the class `Exception` and is inherited by the class `DivideByZeroException`. For example, Listing 9.6 shows a sample program that uses this exception class. The exception is created by the default constructor and then thrown, as follows:

```
throw new DivideByZeroException();
```

**LISTING 9.6    Using a Programmer-Defined Exception Class (part 1 of 3)**

---

```

import java.util.Scanner;
public class DivideByZeroDemo
{
    private int numerator;
    private int denominator;
    private double quotient;

    public static void main(String[] args)
    {
        DivideByZeroDemo oneTime = new DivideByZeroDemo();
        oneTime.doIt();
    }
    public void doIt()
    {
        try
        {
            System.out.println("Enter numerator:");
            Scanner keyboard = new Scanner(System.in);
            numerator = keyboard.nextInt();

```

*We will present an improved version of this program later in this chapter.*

(continued)

**LISTING 9.6 Using a Programmer-Defined Exception Class (part 2 of 3)**

```

        System.out.println("Enter denominator:");
        denominator = keyboard.nextInt();

        if (denominator == 0)
            throw new DivideByZeroException();

        quotient = numerator / (double)denominator;
        System.out.println(numerator + "/" + denominator +
                           " = " + quotient);
    }
    catch(DivideByZeroException e)
    {
        System.out.println(e.getMessage());
        giveSecondChance();
    }
    System.out.println("End of program.");
}
public void giveSecondChance()
{
    System.out.println("Try again:");
    System.out.println("Enter numerator:");
    Scanner keyboard = new Scanner(System.in);
    numerator = keyboard.nextInt();
    System.out.println("Enter denominator:");
    System.out.println("Be sure the denominator is not zero.");
    denominator = keyboard.nextInt();

    if (denominator == 0)
    {
        System.out.println("I cannot do division by zero.");
        System.out.println("Since I cannot do what you want,");
        System.out.println("the program will now end.");
        System.exit(0);
    }

    quotient = ((double)numerator) / denominator;
    System.out.println(numerator + "/" + denominator +
                       " = " + quotient);
}
}

```

*Sometimes, dealing with an exceptional case without throwing an exception is better.*

(continued)



This exception is caught in the catch block, which contains the following statement:

```
System.out.println(e.getMessage());
```

This statement displays the following output on the screen, as shown in the sample screen output of Listing 9.6:

```
Dividing by Zero!
```

The class `DivideByZeroException` in Listing 9.5 also defines a second constructor. This constructor has one parameter of type `String`, allowing you to choose any message you like when you throw an exception. If the throw statement in Listing 9.6 was instead

```
throw new DivideByZeroException(  
    "Oops. Shouldn't Have Used Zero.");
```

the statement

```
System.out.println(e.getMessage());
```

would have produced the following output to the screen:

```
Oops. Shouldn't Have Used Zero.
```

Notice that the try block in Listing 9.6 contains the normal part of the program. If all goes normally, that is the only code that will be executed, and the output will be like that shown in Sample Screen Output 1. In the exceptional case, when the user enters zero for the denominator, the exception is thrown and then is caught in the catch block. The catch block displays the message of the exception and then calls the method `giveSecondChance`. The method `giveSecondChance` gives the user a second chance to enter the input correctly and then carries out the calculation. If the user tries a second time to divide by zero, the method ends the program without throwing an exception. The method `giveSecondChance` exists only for this exceptional case. So we have separated the code for the exceptional case of a division by zero into a separate method, where it will not clutter the code for the normal case.

A try block contains the normal case; a catch block deals with the exceptional case

## ■ PROGRAMMING TIP Preserve `getMessage` in Your Own Exception Classes

For all predefined exception classes, the method `getMessage` will return the string that is passed as an argument to the constructor. (If no argument is passed to the constructor—that is, if you invoke the default constructor—`getMessage` returns a default string.) For example, suppose the exception is thrown as follows:

```
throw new Exception("This is a big exception!");
```

The value of the `String` instance variable is set to "This is a big exception!" If the exception object is called `e`, the method call `e.getMessage()` returns "This is a big exception!"

You should preserve this behavior of the method `getMessage` in any exception class you define. For example, suppose you define an exception class called `MySpecialException` and throw an exception as follows:

```
throw new MySpecialException("Wow, what an exception!");
```

If `e` is a name for the exception thrown, `e.getMessage()` should return "Wow, what an exception!" To ensure that the exception classes that you define behave in this way, be sure to include a constructor that has a string parameter and whose definition begins with a call to `super`, as illustrated by the following constructor:

```
public MySpecialException(String message)
{
    super(message);
    //There can be more code here, but often there is none.
}
```

The call to `super` is a call to a constructor of the base class. If the base-class constructor handles the message correctly, so will a class defined in this way.

You should also include a default constructor in each exception class. This default constructor should set up a default value to be retrieved by `getMessage`. The constructor's definition should begin with a call to `super`, as illustrated by the following constructor:

```
public MySpecialException()
{
    super("MySpecialException thrown.");
    //There can be more code here, but often there is none.
}
```

If `getMessage` works as we described for the base class, this default constructor will work correctly for the new exception class being defined. ■

### REMEMBER Characteristics of Exception Objects

The two most important things about an exception object are

- The object's type—that is, the name of the exception class. The upcoming sections explain why this is important.
- The message that the object carries in an instance variable of type `String`. This string can be recovered by calling the accessor method `getMessage`. The string allows your code to send a message along with an exception object so that the `catch` block can recover the message.

## ■ PROGRAMMING TIP When to Define an Exception Class

As a general rule, if you are going to insert a `throw` statement in your code, it is probably best to define your own exception class. That way, when your code catches an exception, your catch blocks can tell the difference between your exceptions and exceptions thrown by methods in predefined classes. For example, in Listing 9.6, we used the exception class `DivideByZeroException`, which we defined in Listing 9.5.

Although doing so would not be a good idea, you might be tempted to use the predefined class `Exception` to throw the exception in Listing 9.6, as follows:

```
throw new Exception("Dividing by Zero!");
```

You could then catch this exception with the catch block

```
catch(Exception e)
{
    System.out.println(e.getMessage());
    giveSecondChance();
}
```

Although this approach will work for the program in Listing 9.6, it is not the best technique, because the previous catch block will catch any exception, such as an `IOException`. An `IOException`, however, might need a different action than the one provided by a call to `giveSecondChance`. Thus, rather than using the class `Exception` to deal with a division by zero, it is better to use the more specialized programmer-defined class `DivideByZeroException`, as we did in Listing 9.6. ■

### RECAP Programmer-Defined Exception Classes

You can define your own exception classes, but every such class must be derived from an existing exception class—either a predefined class or one defined by you.

#### GUIDELINES

- Use the class `Exception` as the base, if you have no compelling reason to use any other class as the base class.
- You should define at least two constructors, including a default constructor and one that has a single `String` parameter.
- You should start each constructor definition with a call to the constructor of the base class, using `super`. For the default constructor, the call to `super` should have a string argument that indicates what kind of exception it is. For example,

```
super("Tidal Wave Exception thrown!");
```

(continued)

If the constructor has a `String` parameter, the parameter should be the argument in the call to `super`. For example,

```
super(message);
```

In this way, the string can then be recovered using the `getMessage` method.

- Your exception class inherits the method `getMessage`. You should not override it.
- Normally, you do not need to define any other methods, but it is legal to do so.

#### EXAMPLE

```
public class TidalWaveException extends Exception
{
    public TidalWaveException()
    {
        super("Tidal Wave Exception thrown!");
    }
    public TidalWaveException(String message)
        super(message);
    }
}
```

`super` is a call to the constructor of the base class `Exception`


## SELF-TEST QUESTIONS



VideoNote  
Using your exception  
classes

15. Define an exception class called `CoreBreachException`. The class should have a default constructor. If an exception is thrown using this zero-argument constructor, `getMessage` should return "Core Breach! Evacuate Ship!" The class should also define a constructor having a single parameter of type `String`. If an exception is thrown using this constructor, `getMessage` should return the value that was used as an argument to the constructor.
16. Repeat the previous question, but instead name the class `MessageTooLongException`. Also, if an exception is thrown using the default constructor, `getMessage` should return "Message Too Long!"
17. Suppose the exception class `ExerciseException` is defined as follows:

```
public class ExerciseException extends Exception
{
```



*All is in flux, nothing stays still.*

—HERACLITUS

---

As Chapter 5 noted, an abstract data type, or ADT, specifies a set of data and the operations on that data. It describes what the operations do but not how to implement them or store the data. Basically, an ADT specifies a particular data organization. You can express the specifications of an ADT by writing a Java interface, as described in Chapter 8. You know that a class can implement an interface in various ways. Thus, you implement an ADT by defining a Java class. In doing so, you use various data structures. A **data structure** is a construct, such as a class or an array, within a programming language.

This chapter focuses on data structures. In particular, we discuss two kinds of data structures whose size can grow or shrink while your program is running. Such structures are said to be **dynamic**. One kind of dynamic structure is based on an array. As an example, we will introduce the class `ArrayList`, which is available in the Java Class Library. The other kind links its data items to one another, so that one data item “knows” where the next one is. Although there are many different kinds of linked data structures, we will emphasize one simple but useful linked data structure known as a linked list. In the process of covering linked lists, we will introduce inner classes, which are a kind of class definition within another class definition.

Starting with version 5.0, Java allows class definitions to have parameters for the data types they use. These parameters are known as **generic types**. In Section 12.1 we show you how to use one such definition—`ArrayList`—which is in the Java Class Library. Section 12.3 shows you how to write class definitions that have generic data types.

## OBJECTIVES

After studying this chapter, you should be able to

- Define and use an instance of `ArrayList`
- Describe the general idea of linked data structures and how to implement them in Java
- Manipulate linked lists
- Use inner classes in defining linked data structures
- Describe, create, and use iterators
- Define and use classes that have generic types
- Understand how the Scene Builder can help you create a JavaFX Application

## PREREQUISITES

You should read Section 12.1 before Section 12.2. Section 12.3 stands on its own but you should read Sections 12.1 through 12.3 before Section 12.4. Chapters 1 through 7 are needed to fully understand this chapter. Some familiarity with basic inheritance and basic exception handling will be helpful. The details are as follows:

Section	Prerequisite
12.1 Array-Based Data Structures	Chapters 1 through 7 and Section 9.1
12.2 The Java Collections Framework	Section 12.1
12.3 Linked Data Structures (excluding "Exception Handling with Linked Lists")	Chapters 1 through 7
The subsection of Section 12.3 entitled "Exception Handling with Linked lists"	Chapters 1 through 7 and Chapter 9
12.4 Generics	Sections 12.1 through 12.3 up to, but not including, "Exception Handling with Linked Lists"

## 12.1 ARRAY-BASED DATA STRUCTURES

*"Well, I'll eat it," said Alice, "and if it makes me grow larger, I can reach the key; and if it makes me grow smaller, I can creep under the door; so either way I'll get into the garden."*

—LEWIS CARROLL, *Alice's Adventures in Wonderland*

In Java, you can read the length of an array as data when the program is running, but once your program creates an array of that length, it cannot change the length of the array. For example, suppose you write a program to record customer orders for a mail-order house, and suppose you store all the items ordered by one customer in an array order of objects of a class called `OrderItem`. You could ask the user for the number of items in an order, store the number in a variable called `numberOfItems`, and then create the array order, using the following statement:

```
OrderItem[] order = new OrderItem[numberOfItems];
```

An array of a certain size

But suppose the customer enters `numberOfItems` items and then decides to order another item? There is no way to actually increase the size of the array order. However, we can simulate increasing its size: We create a new and larger array, copy the elements from the original array to the new array, and

then rename the new array order. For example, the following statements effectively double the size of our array:

```
OrderItem[] largerArray = new OrderItem[2 * numberOfItems];
for (int index = 0; index < numberOfItems; index++)
    largerArray[index] = order[index];
order = largerArray;
```

Doubling the size  
of an array

## The Class ArrayList

Instead of worrying about changing the size of the array order, we can use an instance of the class `ArrayList`, which is in the package `java.util` of the Java Class Library. Such an instance can serve the same purpose as an array, except that it can change in length while the program is running. An `ArrayList` object could handle the customer's extra order without any problems.

`ArrayList`  
versus an array

If we can simply use `ArrayList` to overcome the main disadvantage of using an array, why did we study arrays? Why don't we always use `ArrayList`? It often seems that every silver lining has a cloud, and that is true here as well. There are two main drawbacks when using `ArrayList`:

- Using an instance of `ArrayList` is less efficient than using an array.
- An instance of `ArrayList` can store only objects; it cannot contain values of a primitive type, such as `int`, `double`, or `char`.

The implementation of `ArrayList` uses arrays. In fact, to expand the capacity of its underlying array, it uses the same technique we used to expand our array order. Using `ArrayList` instead of an array in your program will require more computer time, but that might or might not be significant. You would need to analyze your situation before making a choice. The second drawback can be addressed as follows: Instead of storing `int` values, for example, you could store `Integer` values, where `Integer` is the wrapper class whose objects simulate `int` values. Automatic boxing and unboxing—as discussed in Chapter 6—makes using a wrapper class convenient. But using one does add to the time overhead of a program.

The ADT list

Note that `ArrayList` is an implementation of an ADT called a **list**. The ADT list organizes data in the same way that you do when you write a list in everyday life. Lists of tasks, address lists, gift lists, and grocery lists are all examples of familiar lists. In each case, you can add entries to the list—at the beginning, at the end, or between items—delete entries, look at entries, and count entries. These are the same kinds of operations that an object of `ArrayList` can perform.

## Creating an Instance of ArrayList

Using an instance of `ArrayList` has similarities to using an array, but there are some important differences. First, the definition of the class `ArrayList` is not provided automatically. The definition is in the package `java.util`, and any code that uses the class `ArrayList` must contain the following statement at the start of the file:

```
import java.util.ArrayList;
```

You create and name an instance of `ArrayList` in the same way that you create and name objects of any class, except that you specify the base type using a different notation. For example,

```
ArrayList<String>list = new ArrayList<String>(20);
```

This statement makes `list` the name of an object that stores instances of the class `String`. The type `String` is the base type. An object of the class `ArrayList` stores objects of its base type, just as an array stores items of its base type. The difference here is that the base type of `ArrayList` must be a class type; you cannot use a primitive type, such as `int` or `double`, as the base type.

Our object `list` has an **initial capacity** of 20 items. When we say that an `ArrayList` object has an initial capacity, we mean that it has been allocated memory for that many items, but if it needs to hold more items, the system will automatically allocate more memory. By carefully choosing the initial capacity, you can often make your code more efficient. If you choose an initial capacity that is large enough, the system will not need to reallocate memory too often, and as a result, your program should run faster. On the other hand, if you make your initial capacity too large, you will waste storage space. No matter what capacity you choose, it has no effect on how many items an `ArrayList` object can hold. Note that if you omit the initial capacity, you will invoke `ArrayList`'s default constructor, which assumes a capacity of ten.

`ArrayList` is in the package `java.util`

An object of `ArrayList` stores objects of a specified base type

An object of `ArrayList` can increase its capacity

### RECAP Creating and Naming an Instance of `ArrayList`

An object of the class `ArrayList` is created and named in the same way as any other object, except that you specify a base type.

#### SYNTAX

```
ArrayList<Base_Type> Variable =  
    new ArrayList<Base_Type>();  
ArrayList<Base_Type> Variable =  
    new ArrayList<Base_Type>(Capacity);
```

The *Base\_Type* must be a class type; it cannot be a primitive type such as `int` or `double`. When a number *Capacity* is given as an argument to the constructor, that number determines the initial capacity of the list. Omitting an argument results in an initial capacity of ten.

#### EXAMPLES

```
ArrayList<String>aList = new ArrayList<String>();  
ArrayList<Double>bList = new ArrayList<Double>(30);
```



## ■ PROGRAMMING TIP Newer Versions of Java Require the Base Type Only Once

Prior to JDK version 7, the syntax to create an object of the class `ArrayList` required the base type to be repeated in the constructor.

```
ArrayList<Base_Type> Variable = new ArrayList<Base_Type>();
```

For example:

```
ArrayList<String> aList = new ArrayList<String>();
```

Starting with JDK version 7 the format does not require the base type to be repeated in the constructor. The syntax is:

```
ArrayList<Base_Type> Variable = new ArrayList<>();
```

For example:

```
ArrayList<String> aList = new ArrayList<>();
```

This saves a bit of typing over the old version and can make code more readable for more complex base types. The base type is no longer required in the constructor due to a feature called **type inference**. The compiler is able to infer the base type from the variable declaration. Programmers have been using the earlier format for many years, so you are likely to see it in existing code. ■

## Using the Methods of `ArrayList`

An object of `ArrayList` can be used like an array, but you must use methods instead of an array's square-bracket notation. Let's define an array and an object of `ArrayList` and give them the same capacity:

Creating an array and an object of `ArrayList`

```
String[] anArray = new String[20];
ArrayList<String> aList = new ArrayList<String>(20);
```

Objects of `ArrayList` are indexed in the same way as an array: The index of the first item is 0. So if you would use

```
anArray[index] = "Hi Mom!";
```

for the array `anArray`, the analogous statement for the object `aList` would be

```
aList.set(index, "Hi Mom!");
```

If you would use

```
String temp = anArray[index];
```

to retrieve an element from the array `anArray`, the analogous statement for `aList` would be

```
String temp = aList.get(index);
```

An array and an object of `ArrayList` use the same indexing

The two methods `set` and `get` give objects of `ArrayList` approximately the same functionality that square brackets give to arrays. However, you need to be aware of one important point: The method invocation

The methods `set` and `get`

```
aList.set(index, "Hi Mom!");
```

is *not* always completely analogous to

```
anArray[index] = "Hi Mom!";
```

The method `set` can replace any *existing* element, but you cannot use `set` to put an element at just any index, as you would with an array. The method `set` is used to change the value of existing elements, not to set them for the first time.

To set an element for the first time, you use the method `add`. This method adds elements at index positions 0, 1, 2, and so forth, in that order. An `ArrayList` object must always be filled in this order. The method `add` has two forms; that is, it is overloaded. When given one argument, `add` adds an element immediately after the last currently occupied position. Given two arguments, the method adds the element at the indicated position, assuming that the indicated position is not after an unoccupied position. For example, if `aList` contains five elements, either

```
aList.add("Caboose");
```

or

```
aList.add(5, "Caboose");
```

adds "Caboose" as the sixth—and last—element. But notice that for a list of five items, the statement

```
aList.add(6, "Caboose");
```

would cause an `IndexOutOfBoundsException` exception. Because the list contains five items, the index of the last item is 4. Attempting to add an item at index 6 before placing one at index 5 is illegal.

After adding elements to `aList`, you also can use `add` to insert an element before or between existing elements. The statement

The method `add`

```
aList.add(0, "Engine");
```

adds (inserts) a string before all other elements in `aList`. Existing elements are shifted to make room for the new element. So the original string at index 0 is not replaced, but rather is shifted to index position 1. Likewise,

```
aList.add(4, "BoxCar");
```

inserts "BoxCar" as the element at index 4. Elements at indices before 4 remain in place, while those originally after it are shifted to the next higher index position. When you use the method `add` to insert a new element at an index position, all the elements that were at that index position or higher

### ■ PROGRAMMING TIP Use a For-Each Loop to Access All Elements in an Instance of ArrayList

The last loop in Listing 12.1, which displays all the elements in an instance of `ArrayList`, can be replaced by the following for-each loop:

```
for (String element : toDoList)
    System.out.println(element);
```

This loop is much simpler to write than the original one when you want to access all of the elements in a collection such as an `ArrayList` object. ■

### ■ PROGRAMMING TIP Use `trimToSize` to Save Memory

`ArrayList` objects automatically double their capacity when your program needs them to have additional capacity. However, the new capacity may be more than your program requires. In such cases, the capacity does not automatically shrink. If the capacity of your expanded list is much larger than you need, you can save memory by using the method `trimToSize`. For example, if `aList` is an instance

of `ArrayList`, the invocation `aList.trimToSize()` will shrink the capacity of `aList` down to its actual size, leaving it with no unused capacity. Normally, you should use `trimToSize` only when you know that you will not soon need the extra capacity. ■

#### ASIDE Early Versions of ArrayList

Versions of Java before version 5.0 had a class named `ArrayList` that was not parameterized and was used just like any other class. For example,

```
ArrayList aList = new
    ArrayList(10);
```

This older class is still available in recent versions of Java and is approximately equivalent to the class `ArrayList<Object>`. So the previous line of code is approximately equivalent to

```
ArrayList<Object>aList = new
    ArrayList<Object>(10);
```

The class `ArrayList<Object>` and the older class `ArrayList` are not completely equivalent, but for simple applications they can be considered equivalent. The differences between these classes become relevant only when you are dealing with issues related to type casting or subtyping. This older class is in the package `java.util`, as is the newer parameterized `ArrayList`. Knowing about the older class is relevant if you have a program written in earlier versions of Java.

### GOTCHA Using an Assignment Statement to Copy a List

As was true for objects of other classes, as well as for arrays, you cannot make a copy of an instance of `ArrayList` by using an assignment statement. For example, consider the following code:

```
ArrayList<String>aList = new
    ArrayList<String>();
<Some code to fill aList>
ArrayList<String>anotherName = aList;
//Defines an alias
```

This code simply makes `anotherName` another name for `aList` so that you have two names but only one list. If you want to make an identical copy of `aList` so that you have two separate copies, you use the method `clone`. So instead of the previous assignment statement, you would write

```
ArrayList<String> duplicateList =
    (ArrayList<String>)aList.clone();
```

For lists of objects other than strings, using the method `clone` can be more complicated and can lead to a few pitfalls. Appendix 10, explains more about this method. ■

## Parameterized Classes and Generic Data Types

The class `ArrayList` is a **parameterized class**. That is, it has a parameter, which we have been denoting *Base\_Type*, that can be replaced with any class type to obtain a class that stores objects having the specified base type. *Base\_Type* is called a **generic data type**. You already know how to use parameterized classes, since you know how to use `ArrayList`. Section 12.3 will outline how to define such a class yourself.

### SELF-TEST QUESTIONS

10. Can you have a list of ints?
11. Suppose `list` is an instance of `ArrayList`. What is the difference between the capacity of `list` and the value of `list.size()`?
12. Suppose `list` is an instance of `ArrayList` that was defined with an initial capacity of 20. Imagine that we add 10 entries to `list`. What will be the values of the elements at indices 10 through 19? Garbage values? A default value? Something else?

## 12.2 THE JAVA COLLECTIONS FRAMEWORK

*Any idea is a generalization, and generalization is a property of thinking. To generalize something means to think it.*

—GEORG WILHELM FRIEDRICH HEGEL, *Elements of the Philosophy of Right*, (1820)

The **Java Collections Framework** is a collection of interfaces and classes that may be used to manipulate groups of objects. The classes implemented in the Java Collections Framework serve as reusable data structures and include algorithms for common tasks such as sorting or searching. The framework uses parameterized classes so you can use them with the classes of your choice. Utilizing the framework can free the programmer from lots of low-level details if they aren't the focus of the program. This section is just enough for you to use a few common components of the Java Collections Framework.

### The Collection Interface

The **Collection interface** is the highest level of Java's framework for collection classes and it describes the basic operations that all collection classes should implement. Selected methods for the **Collection** interface are

**FIGURE 12.2** Selected Methods in the Collection Interface

<code>public boolean add(Base_Type newElement)</code> Adds the specified element to the collection. Returns <code>true</code> if the collection is changed as a result of the call.
<code>public void clear()</code> Removes all of the elements from the collection.
<code>public boolean remove(Object o)</code> Removes a single instance of the specified element from the collection if it is present. Returns <code>true</code> if the collection is changed as a result of the call.
<code>public boolean contains(Object o)</code> Returns <code>true</code> if the specified element is a member of the collection.
<code>public boolean isEmpty()</code> Returns <code>true</code> if the collection is empty.
<code>public int size()</code> Returns the number of elements in the collection.
<code>public Object[] to Array()</code> Returns an array containing all of the elements in the collection. The array is of a type <code>Object</code> so each element may need to be type cast back into the original base type.

given in Figure 12.2. The methods support basic operations such as adding, removing, or checking to see if an object exists in the collection. The `Collection` interface takes a base type that allows you to create a collection of objects of your choice. Note that the `toArray` method returns an array of type `Object`. You may need to type cast elements of the array to the appropriate class or abstract data type.

You may have noticed that many of the methods from Figure 12.2 look the same as those in Figure 12.1, the listing of methods for the `ArrayList` class. This is no coincidence. The `ArrayList` class implements the `Collection` interface, so it must have the same methods. Other classes that implement the `Collection` interface must also have the same methods, but of course the implementation of the methods may differ. Additionally, classes can add their own methods not specified by the interface. For example, the `get` and `set` methods for the `ArrayList` class are specified not in the `Collection` interface but by a derived interface, `List`, that in turn is implemented by `ArrayList`. While these methods provide specialized functionality, an understanding of the `Collection` interface is enough to give you a basic understanding of how to use any class that implements it.

The Class `HashSet`

The `HashSet` class stores a set of objects

The `HashSet` class is used to store a set of objects. Like the `ArrayList`, the `HashSet` class also implements the `Collection` interface. However, the `HashSet` stores a set rather than a list of items. This means that there can be no duplicate elements, unlike an `ArrayList`, which can have many duplicates.

The class is named `HashSet` because the algorithm used to implement the set is called a hash table. A description of how the hash table algorithm works is beyond the scope of this text, but in summary it provides a fast and efficient way to look up items. Listing 12.2 demonstrates how to use the `HashSet` class to store and manage a set of integers. The base type is the wrapper class `Integer` (described in Section 6.2) rather than `int` because primitive types are not allowed as a base type. Only objects may be stored in any collection. The example uses only the methods specified in the `Collection` interface.

It is important to note that if you intend to use the `HashSet<T>` class with your own class as the parameterized type `T` then your class `T` must override the following methods:

```
public int hashCode();
public boolean equals(Object obj);
```

The `hashCode()` method should return a numeric key that is ideally a unique identifier for each object in your class. It is always a good idea to override the

---

#### LISTING 12.2 A `HashSet` Demonstration (part 1 of 2)

---

```
import java.util.HashSet;
public class HashSetDemo
{
    public static void main(String[] args)
    {
        HashSet<Integer> intSet = new HashSet<Integer>();
        intSet.add(2);
        intSet.add(7);
        intSet.add(7);           ← Ignored since 7 is already in the set
        intSet.add(3);
        printSet(intSet);
        intSet.remove(3);
        printSet(intSet);
        System.out.println("Set contains 2: " +
            intSet.contains(2));
        System.out.println("Set contains 3: " +
            intSet.contains(3));
    }
    public static void printSet(HashSet<Integer> intSet)
    {
        System.out.println("The set contains:");
        for (Object obj : intSet.toArray())
        {
            Integer num = (Integer) obj;
            System.out.println(num.intValue());
        }
    }
}
```

(continued)

**LISTING 12.2** A `HashSet` Demonstration (part 2 of 2)*Sample Screen Output*

```
The set contains:
2
3
7
The set contains:
2
7
Set contains 2: true
Set contains 3: false
```

`equals()` method for any class you write, but you must override it in this scenario. Java will use the hash code to index the object and then use the `equals()` method to check if an object exists in the set. If the hash code for two different objects is identical they will still be indexed correctly as long as `equals()` indicates they are unique. However, the identical hash codes will decrease performance.

## The Map Interface

The Map interface describes a mapping from a key object to a value object

The Map **interface** is also a top-level interface in the Java Collection Framework. The Map interface is similar in character to the `Collection` interface, except that it deals with collections of ordered pairs. Think of the pair as consisting of a key *K* (to search for) and an associated value *V*. For example, the key might be a student ID number and the value might be an object storing information about the student (such as the name, major, address, or phone number) associated with that ID number. Selected methods for the Map interface are given in Figure 12.3. The Map interface takes a base type for the key and a base type for the value. Use the `put` method to add a key/value pair to the collection and the `get` method to retrieve the value for a given key. Just like the `Collection` interface, the base types must be objects and cannot be primitive data types.

## The Class `HashMap`

The `HashMap` class implements the Map interface

The `HashMap` class implements the Map interface and is used to store a map from a key object to a value object. Like the `HashSet`, the class is called `HashMap` because it also uses the hash table algorithm. It can be used like a small database and is able to quickly retrieve the value object when given the key object. Listing 12.3 demonstrates how to use the `HashMap` class to map from the names of mountain peaks to their height in feet. The name of the mountain is the key and the height is the mapped value. The base type of the key is `String` and the base type of the value is `Integer`. We must use the wrapper class `Integer` rather than `int` because primitive data types are not allowed as a base type. The example uses only the methods specified in the Map interface to add several mappings, look up a mapping, modify a mapping, and remove a mapping.



VideoNote  
Walkthrough of the  
`HashMap` demonstration

**FIGURE 12.3** Selected Methods in the Map Interface

<code>public Base_Type_Value put(Base_Type_Key k, Base_Type_Value v)</code> Associates the value <code>v</code> with the key <code>k</code> . Returns the previous value for <code>k</code> or <code>null</code> if there was no previous mapping.
<code>public Base_Type_Value get(Object k)</code> Returns the value mapped to the key <code>k</code> or <code>null</code> if no mapping exists.
<code>public void clear()</code> Removes a single instance of the specified element from the collection if it is present. Returns <code>true</code> if the collection is changed as a result of the call.
<code>public Base_Type_Value remove(Object k)</code> Removes the mapping of key <code>k</code> from the map if present. Returns the previous value for the key <code>k</code> or <code>null</code> if there was no previous mapping.
<code>public boolean containsKey(Object k)</code> Returns <code>true</code> if the key <code>k</code> is a key in the map.
<code>public boolean containsValue(Object v)</code> Returns <code>true</code> if the value <code>v</code> is a value in the map.
<code>public boolean hashCode()</code> Returns the hash code value for the calling object.
<code>public boolean isEmpty()</code> Returns <code>true</code> if the map contains no mappings.
<code>public int size()</code> Returns the number of mappings in the map.
<code>public Set &lt;Base_Type_Key&gt; keySet()</code> Returns a set containing all of the keys in the map.
<code>public Collection &lt;Base_Type_Value&gt; values()</code> Returns a collection containing all of the values in the map.

**LISTING 12.3** A HashMap Demonstration (*part 1 of 2*)

```
import java.util.HashMap;
public class HashMapDemo
{
    public static void main(String[] args)
    {
        HashMap<String, Integer> mountains =
            new HashMap<String, Integer>();
        mountains.put("Everest", 29029);
        mountains.put("K2", 28251);
        mountains.put("Kangchenjunga", 28169);
        mountains.put("Denali", 20335);
        printMap(mountains);
        System.out.println("Denali in the map: " +
            mountains.containsKey("Denali"));
        System.out.println();
    }
}
```

*(continued)*



**LISTING 12.3 A HashMap Demonstration (part 2 of 2)**

```

        System.out.println("Changing height of Denali.");
        mountains.put("Denali", 20320);
        printMap(mountains);
        System.out.println("Removing Kangchenjunga.");
        mountains.remove("Kangchenjunga");
        printMap(mountains);
    }
    public static void printMap(HashMap<String, Integer> map)
    {
        System.out.println("Map contains:");
        for (String keyMountainName : map.keySet())
        {
            Integer height = map.get(keyMountainName);
            System.out.println(keyMountainName + " --> " +
                               height.intValue() + " feet.");
        }
        System.out.println();
    }
}

```

*Overwrites the old value for Denali*

**Sample Screen Output**

```

Map contains:
K2 --> 28251 feet.
Denali --> 20355 feet.
Kangchenjunga --> 28169 feet.
Everest --> 29029 feet.
Denali in the map: true
Changing height of Denali.
Map contains:
K2 --> 28251 feet.
Denali --> 20320 feet.
Kangchenjunga --> 28169 feet.
Everest --> 29029 feet.
Removing Kangchenjunga.
Map contains:
K2 --> 28251 feet.
Denali --> 20320 feet.
Everest --> 29029 feet.

```



As with the `HashSet<T>` class, if you intend to use your own class as the parameterized type `K` in a `HashMap<K, V>` then your class must override the following methods:

```
public int hashCode();  
public boolean equals(Object obj);
```

These methods are required for indexing and checking for uniqueness of the key.

### ■ PROGRAMMING TIP Other Classes in the Java Collections Framework

There are many other methods, classes, and interfaces in the Java Collections Framework. You will be happy to know that the methods specified in either the `Collection` or `Map` interfaces provide a uniform interface for all classes in the framework. For example, there is a `TreeSet` class that stores data in a tree instead of a hash table like the `HashSet` class. The program in Listing 12.2 will produce identical output if the data type is changed to `TreeSet`. However, many classes have additional methods specific to the type of data structure or algorithm being implemented. ■

## SELF-TEST QUESTIONS

13. Define and invoke the constructor for a `HashSet` variable named `colors` capable of holding strings.
14. Given the variable `colors` defined in Question 13, write the code to add "red" and "blue" to the set, output if the set contains "blue", then remove "blue" from the set.
15. Define and invoke the constructor for a `HashMap` named `studentIDs` that holds a mapping of integers to strings.
16. Given the variable `studentIDs` defined in Question 15, write the code to map 5531 to "George", 9102 to "John", and print the name associated with ID 9102.

## 12.3 LINKED DATA STRUCTURES

*Do not mistake the pointing finger for the moon.*

—ZEN SAYING

A **linked data structure** is a collection of objects, each of which contains data and a reference to another object in the collection. We will confine most of our discussion of linked data structures to a simple but widely used kind known as a linked list.

A collection of  
linked objects

### The Class `LinkedList`

Instead of using the class `ArrayList` and worrying about allocating too much memory, we can use an instance of the class `LinkedList`, which is also in the

The class  
LinkedList  
implements a list

package `java.util` of the Java Class Library. Like `ArrayList`, `LinkedList` is another implementation of the ADT list. Although the two classes do not have the same methods, both have the same basic list operations described in Figure 12.1, with the exception of the constructors and the last method, `trimToSize`. As you will see, `LinkedList`—as a linked data structure—allocates memory only as needed to accommodate new entries and deallocates memory when an entry is removed.

After importing `LinkedList` from the package `java.util`, you create a new instance of `LinkedList` by invoking its default constructor. For example, the following statement creates `myList` as a list of strings:

```
LinkedList<String> myList = new LinkedList<String>();
```

You then can go on to use the methods of `LinkedList`—such as `add`, `set`, `get`, `remove`, and `size`—just as you did when using the class `ArrayList`.

It makes sense to use predefined classes such as `LinkedList` and `ArrayList`, since they were written by experts, are well tested, and will save you a lot of work. However, using `LinkedList` will not teach you how to implement linked data structures in Java. To do that, you need to see a simple example of building at least one linked data structure. A linked list is both a simple and a typical linked data structure. We will construct our own simplified example of a linked list so you can see how linked data structures work.

## PROGRAMMING TIP The Interface List

The Java Collections Framework contains the interface `List`, which specifies the operations for an ADT list. The methods listed in Figure 12.1 are a part of this interface. Both of the classes `ArrayList` and `LinkedList` implement the interface `List`.



VideoNote  
Using List, ArrayList,  
and LinkedList

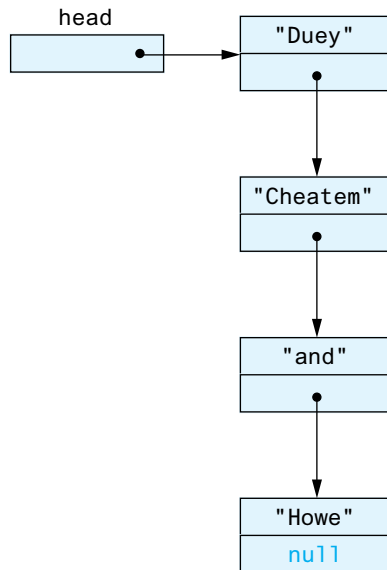
## SELF-TEST QUESTION

- What changes would you make to the program in Listing 12.1 to use `LinkedList` instead of `ArrayList`?

## Linked Lists

A linked list links  
nodes

A **linked list** is a dynamic data structure that links the items in a list to one another. Figure 12.4 shows a linked list in diagrammatic form. Like all linked data structures, a linked list consists of objects known as **nodes**. In the figure, the nodes are the boxes that are divided in half by a horizontal line. Each node has a place for some data and a place to hold a **link** to another node. The links are shown as arrows that point to the node they “link” to. In Java, the links are implemented as references to a node, and in practice they are instance variables of the node type. However, for your first look at a linked list, you can simply think of the links as arrows. In a linked list, each node contains only one link, and the nodes are arranged one after the other so as to form a list, as in Figure 12.4. In an intuitive sense, you or, more properly, your code moves from node to node, following the links.

**FIGURE 12.4 A Linked List**

The link marked `head` is not on the list of nodes; in fact, it is not even a node, but is a link that gets you to the first node. In implementations, `head` will contain a reference to a node, so `head` is a variable of the node type. Your program can easily move through the list in order, from the first node to the last node, by following the “arrows.”

#### **RECAP** Linked List

A linked list is a data structure consisting of objects known as nodes. Each node can contain both data and a reference to one other node so that the nodes link together to form a list, as illustrated in Figure 12.4.

Now let’s see exactly how we can implement a linked list in Java. Each node is an object of a class that has two instance variables, one for the data and one for the link. Listing 12.4 gives the definition of a Java class that can serve as the node class for linked lists like the one shown in Figure 12.4. In this case, the data in each node consists of a single `String` value.<sup>1</sup> As noted in Listing 12.4, we will later make this node class private.

<sup>1</sup> Technically speaking, the node does not contain the string, but only a reference to the string, as would be true of any variable of type `String`. However, for our purposes, we can think of the node as containing the string, since we never use this string reference as an “arrow.”

**LISTING 12.4 A Node Class**

---

```
public class ListNode
{
    private String data;
    private ListNode link;

    public ListNode()
    {
        link = null;
        data = null;
    }
    public ListNode(String newData, ListNode linkValue)
    {
        data = newData;
        link = linkValue;
    }
    public void setData(String newData)
    {
        data = newData;
    }
    public String getData()
    {
        return data;
    }
    public void setLink(ListNode newLink)
    {
        link = newLink;
    }
    public ListNode getLink()
    {
        return link;
    }
}
```

*Later in this chapter, we will hide this class by making it private.*

The class  
`ListNode`

Notice that the `link` instance variable of the class `ListNode` in Listing 12.4 is of type `ListNode`. This relationship sounds circular, and in a sense, it is. However, this kind of class definition is perfectly legal in Java. Recall that a variable of a class type holds a reference to an object of that class. So the `link` instance variable of an object of the `ListNode` class will contain a reference to another object of the class `ListNode`. Thus, as the arrows in the diagram in Figure 12.4 show, each node object of a linked list contains in its `link` instance variable a reference to another object of the class `ListNode`; this other object contains a reference to another object of the class `ListNode`, and so on, until the end of the linked list.

When dealing with a linked list, your code needs to be able to “get to” the first node, and you need some way to detect when the last node is reached. To get to the first node, you use a variable of type `ListNode` that contains a reference to the first node. As we noted earlier, this variable is not a node on the list. In Figure 12.4, the variable containing a reference to the first node is represented by the box labeled `head`. The first node in a linked list is called the **head node**, and it is common to use the name `head` for a variable that contains a reference to this first node. In fact, we call `head` a **head reference**.

A reference to the head (first) node is called the head reference

In Java, you indicate the end of a linked list by setting the `link` instance variable of the last node object to `null`, as shown in Figure 12.4. That way your code can test whether a node is the last node in a linked list by testing whether the node’s `link` instance variable contains `null`. Recall that you check whether a variable contains `null` by using the operator `==`. In contrast, the data instance variable in Listing 12.4 is of type `String`, and you normally check two `String` variables for equality by using the `equals` method.

Linked lists usually start out empty. Since the variable `head` is supposed to contain a reference to the first node in a linked list, what value do you give `head` when there is no first node? You give `head` the value `null` in order to indicate an empty list. This technique is traditional and works out nicely for many algorithms that manipulate a linked list.

`head` is `null` for an empty list

### **REMEMBER** Use `null` for the Empty List and to Indicate Its End

The head reference of an empty linked list contains `null`, as does the `link` portion of the last node of a nonempty linked list.

## Implementing the Operations of a Linked List

Listing 12.5 contains a definition of a linked-list class that uses the node class definition given in Listing 12.4. Note that this new class has only one instance variable, and it is named `head`. This `head` instance variable contains a reference to the first node in the linked list, or it contains `null` if the linked list is empty—that is, when the linked list contains no nodes. The one constructor sets this `head` instance variable to `null`, indicating an empty list.

Before we go on to discuss how nodes are added and removed from a linked list, let’s suppose that a linked list already has a few nodes and that you want to display the contents of all the nodes to the screen. You can do so by writing the following statements:

```
ListNode position = head;
while (position != null)
{
    System.out.println(position.getData());
    position = position.getLink();
}
```

**LISTING 12.5 A Linked-List Class (part 1 of 2)**

---

```

public class StringLinkedList
{
    private ListNode head;
    public StringLinkedList()
    {
        head = null;
    }
    /**
    Displays the data on the list.
    */
    public void showList()
    {
        ListNode position = head;
        while (position != null)
        {
            System.out.println(position.getData());
            position = position.getLink();
        }
    }
    /**
    Returns the number of nodes on the list.
    */
    public int length()
    {
        int count = 0;
        ListNode position = head;
        while (position != null)
        {
            count++;
            position = position.getLink();
        }
        return count;
    }
    /**
    Adds a node containing the data addData at the
    start of the list.
    */
    public void addANodeToStart(String addData)
    {
        head = new ListNode(addData, head);
    }
}

```

*We will give another definition of this class later in this chapter.*

(continued)

**LISTING 12.5 A Linked-List Class (part 2 of 2)**

---

```

/**
 *Deletes the first node on the list.
 */
public void deleteHeadNode()
{
    if (head != null)
        head = head.getLink();
    else
    {
        System.out.println("Deleting from an empty list.");
        System.exit(0);
    }
}
/**
 *Sees whether target is on the list.
 */
public boolean onList(String target)
{
    return find(target) != null;
}
//Returns a reference to the first node containing the
//target data. If target is not on the list, returns null.
private ListNode find(String target)
{
    boolean found = false;
    ListNode position = head;
    while ((position != null) && !found)
    {
        String dataAtPosition = position.getData();
        if (dataAtPosition.equals(target))
            found = true;
        else
            position = position.getLink();
    }
    return position;
}
}

```

---

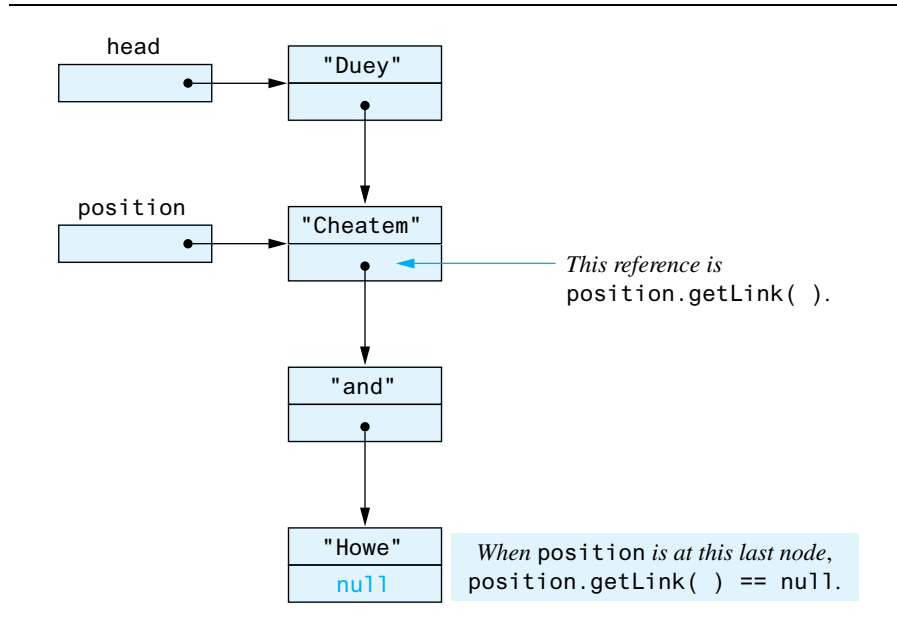
The variable named `position` contains a reference to one node. Initially, `position` contains the same reference as the `head` instance variable; thus, it starts out positioned at the first node. After we display the data in that node, the reference in `position` changes from one node to the next via the assignment

```
position = position.getLink();
```

This process is illustrated in Figure 12.5.

Stepping through  
a list



**FIGURE 12.5** Moving Down a Linked List

To see that this assignment “moves” the position variable to the next node, note that position contains a reference to the node pointed to by the position arrow in Figure 12.5. So position is a name for that node, and position.link is a name for the link portion of that node. Thus, position.link references the next node. The value of link is produced by calling the accessor method getLink. Therefore, a reference to the next node in the linked list is position.getLink(). You “move” the position variable by giving it the value of position.getLink().

The previous loop continues to move the position variable down the linked list, displaying the data in each node as it goes along. When position reaches the last node, it displays the data in that node and again executes

```
position = position.getLink();
```

If you study Figure 12.5, you will see that position’s value is set to null at this point. When this occurs, we want to stop the loop, so we iterate the loop while position is not null.

The method showList contains the loop that we just discussed. The method length uses a similar loop, but instead of displaying the data in each node, length just counts the nodes as the loop moves from node to node. When the loop ends, length returns the number of entries on the list. By the way, a process that moves from node to node in a linked list is said to **traverse** the list. Both showList and length traverse our list.

Next, let’s consider how the method addANodeToStart adds a node to the start of the linked list so that the new node becomes the first node of the list.

This operation is performed by the single statement

```
head = new ListNode(addData, head);
```

Adding a node at the beginning of the list

In other words, the variable `head` is set equal to a reference to a new node, making the new node the first node in the linked list. To link this new node to the rest of the list, we need only set the `link` instance variable of the new node equal to a reference to the old first node. But we have already done that: The new node produced by `newListNode(addData, head)` references the old first node, because `head` contains a reference to the old first node before it is assigned a reference to the new node. Therefore, everything works out as it should. This process is illustrated in Figure 12.6.

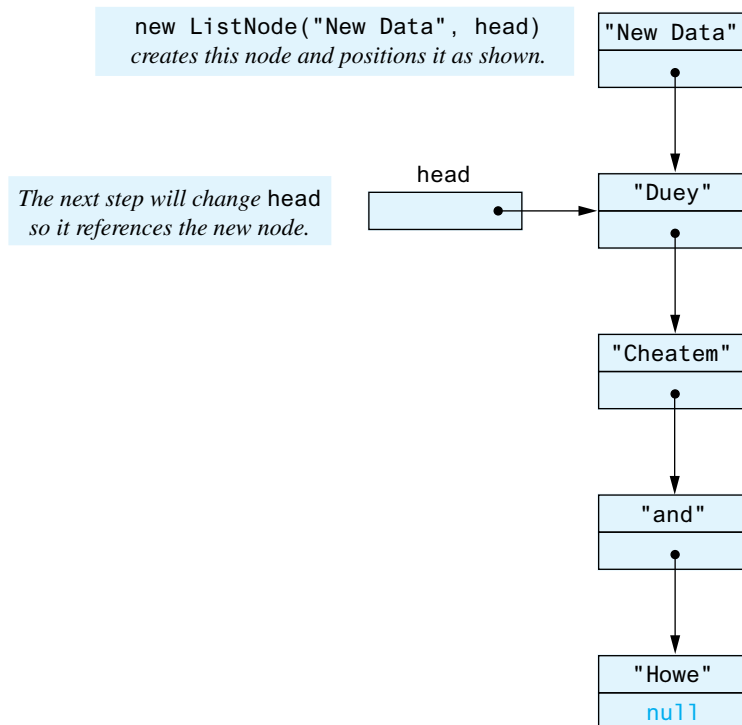
So far, we have added a node only at the start of the linked list. Later, we will discuss adding nodes at other places in a linked list, but the easiest place to add a node is at the start of the list. This is also the easiest place to delete a node.

The method `deleteHeadNode` removes the first node from the linked list and leaves the `head` variable referencing the old second node—which is now the first node—in the linked list. We will leave it to you to figure out that the following assignment correctly accomplishes this deletion:

```
head = head.getLink();
```

Removing the first node

**FIGURE 12.6** Adding a Node at the Start of a Linked List



Listing 12.6 contains a simple program that demonstrates the behavior of some of the methods we just discussed.

---

**LISTING 12.6 A Linked-List Demonstration**

---

```
public class StringLinkedListDemo
{
    public static void main(String[] args)
    {
        StringLinkedList list = new StringLinkedList();
        list.addANodeToStart("One");
        list.addANodeToStart("Two");
        list.addANodeToStart("Three");
        System.out.println("List has " + list.length() +
                           " entries.");
        list.showList();
        if (list.onList("Three"));
            System.out.println("Three is on list.");
        else
            System.out.println("Three is NOT on list.");
        list.deleteHeadNode();
        if (list.onList("Three"));
            System.out.println("Three is on list.");
        else
            System.out.println("Three is NOT on list.");
        list.deleteHeadNode();
        list.deleteHeadNode();
        System.out.println("Start of list:");
        list.showList();
        System.out.println("End of list");
    }
}
```

---

**Screen Output**

```
List has 3 entries.
Three
Two
One
Three is on list.
Three is NOT on list.
Start of list:
End of list.
```

---

**FAQ What happens to a deleted node?**

When your code deletes a node from a linked list, it removes the linked list's reference to that node. So as far as the linked list is concerned, the node is no longer in the linked list. But you gave no command to destroy the node, so it must be some place in the computer's memory. If there is no other reference to the deleted node, the storage that it occupies should be made available for other uses. In many programming languages, you, the programmer, must keep track of deleted nodes and give explicit commands to return their memory for recycling to other uses. This process is called **garbage collecting**. In Java, this task is done for you automatically, or, as it is ordinarily phrased, Java has **automatic garbage collection**.

**GOTCHA `NullPointerException`**

You have undoubtedly received the message `NullPointerException` at some time when you have run a program. (If you have not received the message, congratulations! You are an exceptionally careful programmer.) The message `NullPointerException` indicates that your code tried to use a variable of a class type to reference an object, but the variable contains `null`. That is, the variable does not contain a reference to any object. This message may make more sense to you now. In our nodes, we used `null` to indicate that a link instance variable contains no reference. So a value of `null` indicates no object reference, and that is why the exception is called `NullPointerException`.

A `NullPointerException` is one of the exceptions that you do not need to catch in a catch block or declare in a throws clause. Instead, it indicates that you need to fix your code. ■

**REMEMBER Most Nodes Have No Name**

The variable `head` contains a reference to the first node in a linked list. So `head` can be used as a name for the first node. However, the other nodes in the linked list have no named variable that contains a reference to any of them, so the rest of the nodes in the linked list are nameless. The only way to name one of them is via some indirect reference, like `head.getLink()`, or by using another variable of type `ListNode`, such as the local variable `position` in the method `showList` (Listing 12.5).

## SELF-TEST QUESTIONS

18. How do you mark the end of a linked list?
19. Assume that the variable `head` is supposed to contain a reference to the first node in a linked list and that the linked list is empty. What value should `head` have?
20. Write a definition of a method `isEmpty` for the class `StringLinkedList` that returns `true` if the list is empty, that is, if it has no nodes.
21. What output is produced by the following code?

```
StringLinkedList list = new StringLinkedList();  
list.addANodeToStart("A");  
list.addANodeToStart("B");  
list.addANodeToStart("C");  
list.showList();
```

## A Privacy Leak

The point made in this section is important but a bit subtle. It will help you to understand this section if you first review Section 6.5 in Chapter 6. There we said that a privacy problem can arise in any situation in which a method returns a reference to a private instance variable of a class type. The private restriction on the instance variable could be defeated easily if it names an object that has a set method. Getting a reference to such an object could allow a programmer to change the private instance variables of the object.

Consider the method `getLink` in the class `ListNode` (Listing 12.4). It returns a value of type `ListNode`. That is, it returns a reference to a `ListNode` object, that is, a node. Moreover, `ListNode` has public set methods that can damage the contents of a node. Thus, `getLink` can cause a privacy leak.

On the other hand, the method `getData` in the class `ListNode` causes no privacy leak, but only because the class `String` has no set methods. The class `String` is a special case. If the data were of another class type, `getData` could produce a privacy leak.

If the class `ListNode` is used only in the definition of the class `StringLinkedList` and classes like it, there is no privacy leak. This is the because no public method in the class `StringLinkedList` returns a reference to a node. Although the return type of the method `find` is `ListNode`, it is a private method. If the method `find` were public, a privacy leak would result. Therefore, note that making `find` private is not simply a minor stylistic point.

Although there is no problem with the class definition of `ListNode` when it is used in a class definition like `StringLinkedList`, we cannot guarantee that the class `ListNode` will be used only in this way. You can fix this privacy-leak

problem in a number of ways. The most straightforward way is to make the class `ListNode` a private inner class in the class `StringLinkedList`, as discussed in the next two sections, “Inner Classes” and “Node Inner Classes.” Another, and similar, solution is to place both of the classes `ListNode` and `StringLinkedList` into a package; change the private instance variable restriction to the package restriction, as discussed in Appendix 3; and omit the accessor method `getLink`.

## Inner Classes

**Inner classes** are classes defined within other classes. Although a full description of inner classes is beyond the scope of this book, some simple uses of inner classes can be both easy to understand and helpful. In this section, we describe inner classes in general and show how we will use them. In the next section, we give an example of an inner class defined within the linked-list class. That node inner class will be a solution to the privacy leak we just described.

Define an inner class entirely within an outer class

Defining an inner class is straightforward; simply include the definition of the inner class within another class—the **outer class**—as follows:

```
public class OuterClass
{
    Declarations_of_OuterClass_Instance_Variables
    Definitions_of_OuterClass_Methods
    private class InnerClass
    {
        Declarations_of_InnerClass_Instance_Variables
        Definitions_of_InnerClass_Methods
    }
}
```

The definition of the inner class need not be the last item of the outer class, but it is good to place it either last or first so that it is easy to find. The inner class need not be private, but that is the only case we will consider in this book.

An inner-class definition is local to the outer-class definition. So you may reuse the name of the inner class for something else outside the definition of the outer class. If the inner class is private, as ours will always be, the inner class cannot be used outside the definition of the outer class.

The inner-class definition can be used anywhere within the definition of the outer class, and it has a special property that facilitates this use: The methods of the inner and outer classes have access to each other’s methods and instance variables, even if they are private. Within the definitions of the inner or outer classes, the modifiers `public` and `private` are equivalent. Although some details of inner-class usage can be tricky, we will consider only simple cases similar to the one in the next section.

We will revisit inner classes in Chapter 15, which is on the book’s Web site.

## Node Inner Classes

**ListNode as an inner class**

You can make classes like `StringLinkedList` self-contained by making node classes such as `ListNode` inner classes, as follows:

```
public class StringLinkedList
{
    private ListNode head;
    >The methods in Listing 12.5 are inserted here.<
    private class ListNode
    {
        >The rest of the definition of ListNode is the same as in Listing 12.4.<
    }
}
```

Note that we've made the class `ListNode` a private inner class. If an inner class is not intended to be used elsewhere, it should be made private. Making `ListNode` a private inner class is also safer, because it hides the method `getLink` from the world outside the `StringLinkedList` definition. As we noted earlier, in the section entitled "A Privacy Leak," it can be dangerous for a publicly available method to return a reference to an instance variable of a class type.

If you are going to make the class `ListNode` a private inner class in the definition of `StringLinkedList`, you can safely simplify the definition of `ListNode` by eliminating the accessor and mutator methods (the `set` and `get` methods) and just use direct access to the instance variables `data` and `link`. In Listing 12.7, we have rewritten the class `StringLinkedList` in this way. This version, named `StringLinkedListSelf-Contained`, is equivalent to the class `StringLinkedList` in Listing 12.5 in that it has the same methods that perform the same actions. In fact, if you run the program `StringLinkedListDemo` in Listing 12.6, replacing the class `StringLinkedList` with `StringLinkedListSelfContained`, the sample output will not change. (The program `StringLinkedListSelfContainedDemo` is included with the book's source code and does just that.)

**Extra code on the Web**



**VideoNote**  
Adding a node anywhere  
in a linked list

### REMEMBER Node Inner Class

You can make a linked data structure, such as a linked list, self-contained by making the node class an inner class of the linked-list class.

## Iterators

**An iterator steps through a collection**

When you have a collection of objects, such as the nodes of a linked list, you often need to step through all the objects in the collection and perform some action on each object, such as displaying it on the screen or in some way editing its data. An **iterator** is any variable that allows you to step through the list in this way.

**LISTING 12.7 A Linked List with a Node Class as an Inner Class (part 1 of 2)**



---

```

public class StringLinkedListSelfContained
{
    private ListNode head;
    public StringLinkedListSelfContained()
    {
        head = null;
    }
    /**
    Displays the data on the list.
    */
    public void showList()
    {
        ListNode position = head;
        while (position != null)
        {
            System.out.println(position.data);
            position = position.link;
        }
    }
    /**
    Returns the number of nodes on the list.
    */
    public int length()
    {
        int count = 0;
        ListNode position = head;
        while (position != null)
        {
            count++;
            position = position.link;
        }
        return count;
    }
    /**
    Adds a node containing the data addData at the
    start of the list.
    */
    public void addANodeToStart(String addData)
    {
        head = new ListNode(addData, head);
    }
    /**
    Deletes the first node on the list.
    */
    public void deleteHeadNode()
    {

```

Note that the outer class has direct access to the inner-class instance variables `data` and `link`.



(continued)



### LISTING 12.7 A Linked List with a Node Class as an Inner Class (part 2 of 2)

```

        if (head != null)
            head = head.link;
        else
        {
            System.out.println("Deleting from an empty list.");
            System.exit(0);
        }
    }
    /**
     Sees whether target is on the list.
     */
    public boolean onList(String target)
    {
        return (find(target) != null);
    }
    //Returns a reference to the first node containing the
    //target data. If target is not on the list, returns null.
    private ListNode find(String target)
    {
        boolean found = false;
        ListNode position = head;
        while ((position != null) && !found)
        {
            String dataAtPosition = position.data;
            if (dataAtPosition.equals(target))
                found = true;
            else
                position = position.link;
        }
        return position;
    }
    private class ListNode
    {
        private String data;
        private ListNode link;
        public ListNode()
        {
            link = null;
            data = null;
        }
        public ListNode(String newData, ListNode linkValue)
        {
            data = newData;
            link = linkValue;
        }
    }
}

```

*An inner class* →

← *End of outer-class definition*

We will begin by looking at an iterator for an array, as it is easier to program than an iterator for a linked list. An array is a collection of elements having the same data type. An iterator for an array is an `int` variable. If the array is named `anArray` and the `int` variable is named `index`, you can step through all the elements in the array as follows:

```
for (index = 0; index < anArray.length; index++)
    Process anArray[index]
```

The `int` variable `index` is the iterator. You can **iterate**—that is, go from element to element—with the action `index++`.

If you create an array of all the data objects in a linked list, you can iterate through the array. That process is equivalent to iterating through the linked list, provided that you do not want to change the data in the linked list but want only to look at it. For this reason, it is common to have a method in a list class that places all the data in the linked list into an array. Such a method is shown in Listing 12.8. This method, named `toArray`, can be added to the linked list in Listing 12.7. (The class `StringLinkedListSelfContained`, included in the source code available on the Web, contains this method.)

[Extra code on Web](#)

#### LISTING 12.8 Placing the Linked-List Data into an Array

*This method can be added to the linked-list class in Listing 12.7.*

```
/**
Returns an array of the elements on the list.
*/
public String[] toArray()
{
    String[] anArray = new String[length()];
    ListNode position = head;
    int i = 0;
    while (position != null)
    {
        anArray[i] = position.data;
        i++;
        position = position.link;
    }
    return anArray;
}
```

#### RECAP Iterators

Any variable that allows you to step through a collection of data items, such as an array or a linked list, one item at a time in a reasonable way is called an iterator. By “a reasonable way,” we mean that each item is visited exactly once in one full cycle of iterations and that each item can have its data read and, if the data items allow it, can have the data changed.

If you want an iterator that will move through a linked list and allow you to perform operations, such as change the data at a node or even insert or delete a node, having an array of the linked-list data will not suffice. However, an iterator for an array hints at how to proceed. Just as an index specifies an array element, a reference for a node specifies a node. Thus, if you add an instance variable, perhaps named `current`, to the linked-list class `StringLinkedListSelfContained` given in Listing 12.7, you can use this instance variable as an iterator. We have done so in Listing 12.9 and have renamed the class `StringLinkedListWithIterator`. As you can see, we have added a number of methods to manipulate the instance variable `current`. This instance variable is the iterator, but because it is marked `private`, we need methods to manipulate it. We have also added methods for adding and deleting a node at any place in the linked list. The iterator makes it easier to express these methods for adding and deleting nodes, because it gives us a way to name an arbitrary node. Let's go over the details.

#### LISTING 12.9 A Linked List with an Iterator (part 1 of 4)

---

```
/**
 * Linked list with an iterator. One node is the "current node."
 * Initially, the current node is the first node. It can be changed
 * to the next node until the iteration has moved beyond the end
 * of the list.
 */
public class StringLinkedListWithIterator
{
    private ListNode head;
    private ListNode current;
    private ListNode previous;

    public StringLinkedListWithIterator()
    {
        head = null;
        current = null;
        previous = null;
    }

    public void addANodeToStart(String addData)
    {
        head = new ListNode(addData, head);
        if ((current == head.link) && (current != null))
            //if current is at old start node
            previous = head;
    }
}
```

(continued)

**LISTING 12.9** A Linked List with an Iterator  
(part 2 of 4)

---

```
/**
Sets iterator to beginning of list.
*/
public void resetIteration()
{
    current = head;
    previous = null;
}
/**
Returns true if iteration is not finished.
*/
public boolean moreToIterate()
{
    return current != null;
}
/**
Advances iterator to next node.
*/
public void goToNext()
{
    if (current != null)
    {
        previous = current;
        current = current.link;
    }
    else if (head != null)
    {
        System.out.println(
            "Iterated too many times or uninitialized
            iteration.");
        System.exit(0);
    }
    else
    {
        System.out.println("Iterating with an empty list.");
        System.exit(0);
    }
}
/**
Returns the data at the current node.
*/
public String getDataAtCurrent()
{
    String result = null;
    if (current != null)
        result = current.data;
```

(continued)

### LISTING 12.9 A Linked List with an Iterator (part 3 of 4)

---

```

        else
        {
            System.out.println(
                "Getting data when current is not at any node.");
            System.exit(0);
        }
        return result;
    }
    /**
    Replaces the data at the current node.
    */
    public void setDataAtCurrent(String newData)
    {
        if (current != null)
        {
            current.data = newData;
        }
        else
        {
            System.out.println(
                "Setting data when current is not at any node.");
            System.exit(0);
        }
    }
    /**
    Inserts a new node containing newData after the current node.
    The current node is the same after invocation as it is before.
    Precondition: List is not empty; current node is not
    beyond the entire list.
    */
    public void insertNodeAfterCurrent(String newData)
    {
        ListNode newNode = new LisNode();
        newNode.data = newData;
        if (current != null)
        {
            newNode.link = current.link;
            current.link = newNode;
        }
        else if (head != null)
        {
            System.out.println(
                "Inserting when iterator is past all " +
                "nodes or is not initialized.");
            System.exit(0);
        }
    }

```

(continued)

### LISTING 12.9 A Linked List with an Iterator (part 4 of 4)

---

```

else
{
    System.out.println(
        "Using insertNodeAfterCurrent with empty list.";
    System.exit(0);
}
}
/**
Deletes the current node. After the invocation,
the current node is either the node after the
deleted node or null if there is no next node.
*/
public void deleteCurrentNode()
{
    if ((current != null) && (previous == null))
    {
        previous.link = current.link;
        current = current.link;
    }
    else if ((current != null) && (previous == null))
    {//At head node
        head = current.link;
        current = head;
    }
    else //current==null
    {
        System.out.println(
            "Deleting with uninitialized current or an empty " +
            "list.");
        System.exit(0);
    }
}
<The methods length, onList, find, and showList, as well as the private
inner class ListNode are the same as in Listing 12.7.>
<The method toArray is the same as in Listing 12.8.>
}

```

*deleteHeadNode is no longer needed, since you have deleteCurrentNode, but if you want to retain deleteHeadNode, it must be redefined to account for current and previous.*

---

Instance variables  
current and  
previous  
The method  
goToNext  
An internal  
iterator  
An external  
iterator

In addition to the instance variables head and current, we have added an instance variable named previous. The idea is that, as the reference current moves down the linked list, the reference previous follows behind by one node. This setup gives us a way to reference the node before the one named by current. Since the links in the linked list all move in one direction, we need the node previous so that we can do something equivalent to backing up one node.

The method resetIteration starts current at the beginning of the linked list by giving it a reference to the first (head) node, as follows:

```
current = head;
```

#### ASIDE Internal and External Iterators

The class StringLinkedListWithIterator (Listing 12.9) uses the instance variable current of type ListNode as an iterator to step through the nodes of the linked list one after the other. An iterator defined within the linked-list class in this way is known as an **internal iterator**.

If you copy the values in a linked list to an array via the method toArray, you can use a variable of type int as an iterator on the array. The int variable holds one index of the array and thus specifies one element of the array and one data item of the linked list. If the int variable is named position and the array is named a, the iterator position specifies the element a[position]. To move to the next item, simply increase the value of position by 1. An iterator that is defined outside the linked list, such as the int variable position, is known as an **external iterator**. Note that the important thing is not that the array is outside the linked list, but that the int variable position, which is the iterator, is outside the linked list. To better understand this point, note that the int variable position is also an external iterator for the array.

You can have several external iterators at one time, each at a different position within the same list. This is not possible with internal iterators, which is a distinct disadvantage.

It is possible to define an external iterator that works directly with the linked list, rather than with an array of the linked-list data. However, that technique is a bit more complicated, and we will not go into it in this text.

Because the instance variable previous has no previous node to reference, it is simply given the value null by the resetIteration method.

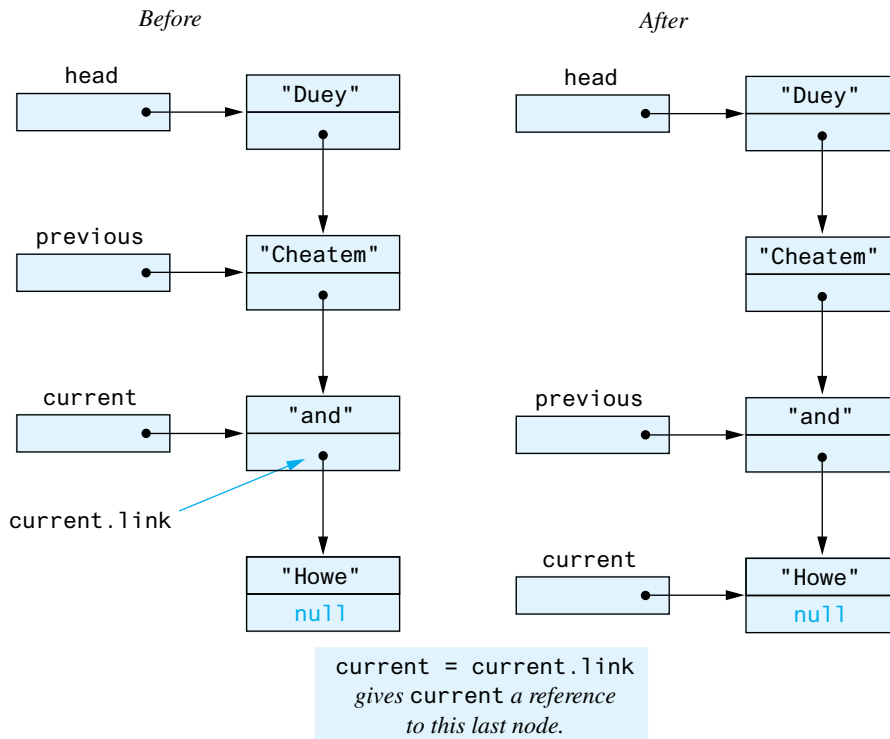
The method goToNext moves the iterator to the next node, as follows:

```
previous = current;  
current = current.link;
```

This process is illustrated in Figure 12.7. In the goToNext method, the last two clauses of the multibranch if-else statement simply produce an error message when the method goToNext is used in a situation where using it does not make sense.

The method moreToIterate returns true as long as current is not equal to null, that is, as long as current contains a reference to some node. This result makes sense most of the time, but why does the method return true when current contains a reference to the last node? When current references the last node, your program cannot tell that it is at the last node until it invokes goToNext one more time, at which point current is set to null. Study Figure 12.7 or the definition of goToNext to see how this process works. When current is equal to null, moreToIterate returns false, indicating that the entire list has been traversed.

Now that our linked list has an iterator, our code has a way to reference any node in the linked list. The current instance variable can hold a reference to any one node; that one node is known as the **node at the iterator**. The method insertAfterIterator inserts a new node after the node at the iterator (at current). This process is illustrated in Figure 12.8. The method

**FIGURE 12.7** The Effect of `goToNext` on a Linked List

`deleteCurrentNode` deletes the node at the iterator. This process is illustrated in Figure 12.9.

The other methods in the class `StringLinkedListWithIterator` (Listing 12.9) are fairly straightforward, and we will leave it up to you to read their definitions and see how they work.

The method  
`moreToIterate`

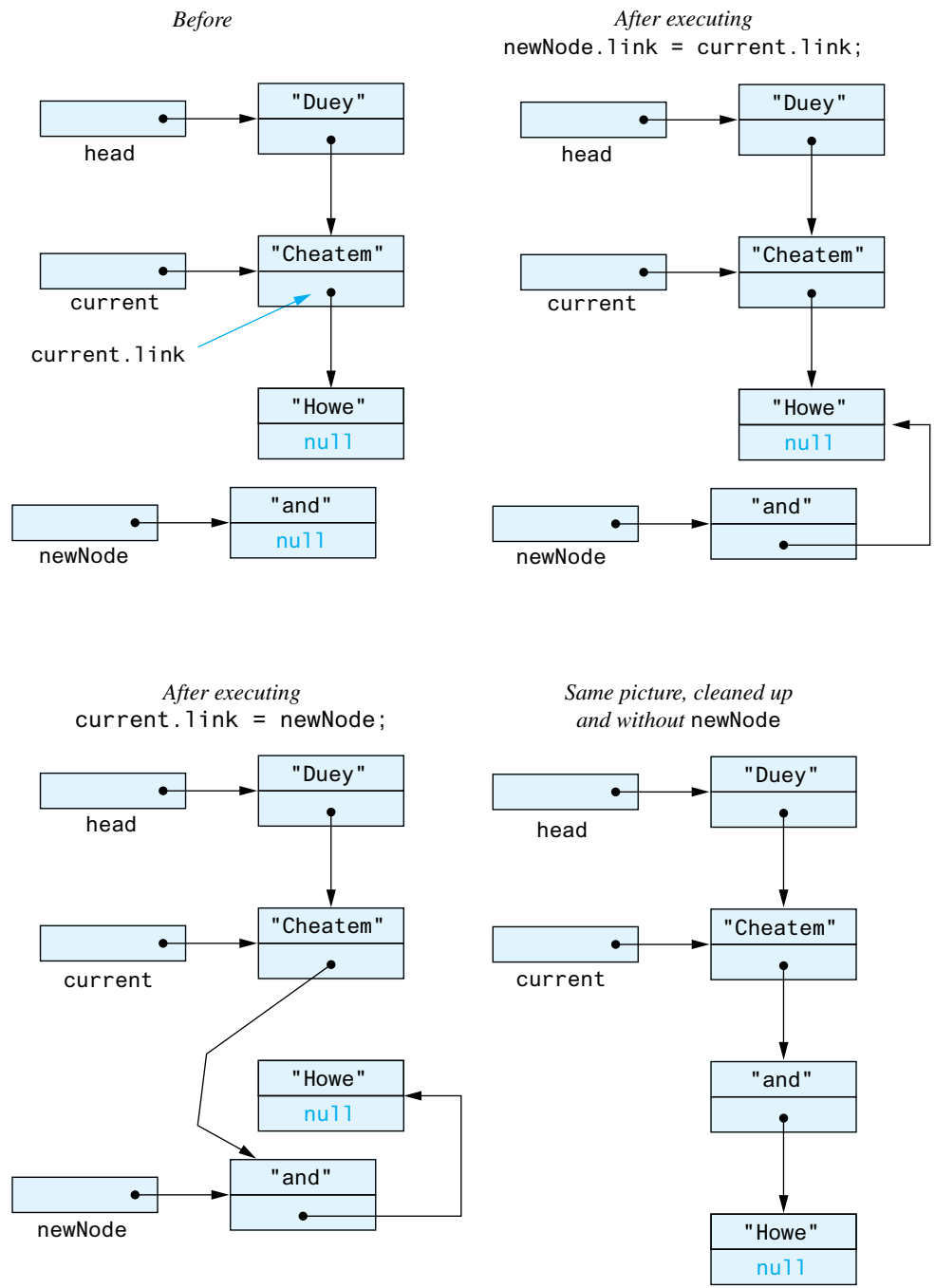
Inserting and  
deleting items  
within a list

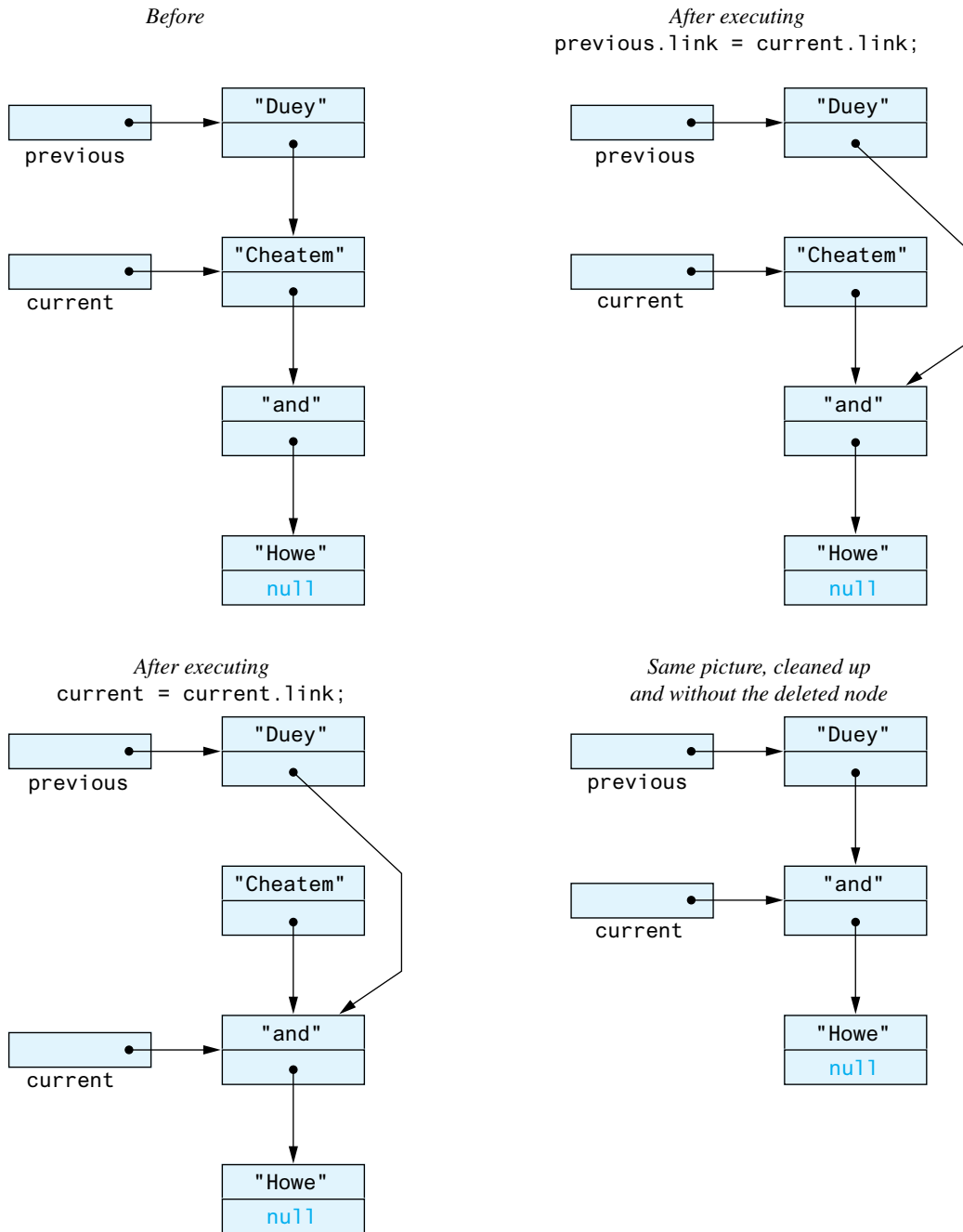
## SELF-TEST QUESTIONS

22. What is an inner class?
23. Why does the definition of the inner class `ListNode` in Listing 12.7 not have the accessor and mutator methods `getLink`, `setLink`, `getData`, and `setData`, as the class definition `ListNode` in Listing 12.4 does?
24. What is an iterator for a collection of items, such as an array or a linked list?



**FIGURE 12.8** Adding a Node to a Linked List, Using insertAfterIterator



**FIGURE 12.9** Deleting a Node

### The standard interface Iterator

## The Java Iterator Interface

Our introduction to iterators defined them as variables that enable you to step through a collection of items. However, Java formally considers an iterator to be an object, not simply a variable. The interface named `Iterator` in the package `java.util` stipulates how Java would like an iterator to behave. The interface specifies the following three methods:

- `hasNext`—returns true if the iteration has another element to return.
- `next`—returns the next element in the iteration.
- `remove`—removes from the collection the element returned most recently by the invocation `next()`.

Our iterators do not satisfy this interface, but it is easy to define classes that use our iterators and that do satisfy this interface. The `Iterator` interface also uses exception handling, but that use is not difficult to understand. Appendix 9, briefly describes the `Iterator` interface. Exercise 16 at the end of this chapter asks you to define a linked-list class that satisfies this interface.

## Exception Handling with Linked Lists

As you may have guessed, you need to have read Chapter 9 on exception handling before reading this section.

Consider the class `StringLinkedListWithIterator` in Listing 12.9. We defined the methods for that class such that whenever something went wrong, the method sent an error message to the screen and ended the program. However, it may not always be necessary to end the program when something unusual happens. To allow the programmer to provide an action in these unusual situations, it would make more sense to throw an exception and let the programmer decide how to handle the situation. The programmer can still decide to end the program but instead could do something else. For example, you could rewrite the method `goToNext` as follows:

```
public void goToNext() throws LinkedListException
{
    if (current != null)
    {
        previous = current;
        current = current.link;
    }
    else if (head != null)
        throw new LinkedListException("Iterated too many times"
                                       + " or uninitialized iteration.");
    else
        throw new LinkedListException("Iterating an empty "
                                       + "list.");
}
```

**LISTING 12.10 The LinkedListException Class**


---

```

public class LinkedListException extends Exception
{
    public LinkedListException()
    {
        super("Linked List Exception");
    }
    public LinkedListException(String message)
    {
        super(message);
    }
}

```

---

In this version, we have replaced each of the branches that end the program with a branch that throws an exception. The exception class `LinkedListException` can be a routine exception class, as shown in Listing 12.10.

You can use an exception thrown during an iteration for a number of different purposes. One possibility is to check for the end of a linked list. For example, suppose that the version of `StringLinkedListWithIterator` that throws exceptions is named `StringLinkedListWithIterator2`. The following code removes all nodes that contain a given string (`badString`) from a linked list, throwing an exception if it attempts to read past the end of the list:

```

StringLinkedListWithIterator2 list =
    new StringLinkedListWithIterator2();
String badString;
<Some code to fill the linked list and set the variable badString.>
list.resetIteration();
try
{
    while (list.length() >= 0)
    {
        if (badString.equals(list.getDataAtCurrent()))
            list.deleteCurrentNode();
        else list.goToNext();
    }
}
catch(LinkedListException e)
{
    if (e.getMessage().equals("Iterating an empty list.))
    {//This should never happen, but
      //the catch block is compulsory.
      System.out.println("Fatal Error.");
      System.exit(0);
    }
}
System.out.println("List cleaned of bad strings.");

```

This use of an exception to test for the end of a list may seem a bit strange at first, but similar uses of exceptions do occur in Java programming. For example, Java requires something like this when checking for the end of a binary file. Of course, there are many other uses for the `LinkedListException` class.

The self-test questions that follow ask you to rewrite more of the methods in `StringLinkedListWithIterator` so that they throw exceptions in unusual or error situations.

## SELF-TEST QUESTIONS

25. Redefine the method `getDataAtCurrent` in `StringLinkedListWithIterator` (Listing 12.9) so that it throws an exception instead of ending the program when something unusual happens.
26. Repeat Question 25 for the method `setDataAtCurrent`.
27. Repeat Question 25 for the method `insertNodeAfterCurrent`.
28. Repeat Question 25 for the method `deleteCurrentNode`.

## Variations on a Linked List

The study of data structures is a large topic, with many good books on the subject. In this book, we cannot introduce you to all aspects of programming, and therefore we cannot go into the topic of linked data structures in exhaustive detail. Instead, we will give you an informal introduction to a few important linked data structures, beginning with some other kinds of linked lists that link their nodes in different ways.

You can have a linked list of any kind of data. Just replace the type `String` in the definition of the node class (and other corresponding places) with the data type you want to use. You can even have a linked list of objects of different kinds by replacing the type `String` in the node definition (and other corresponding places) with the type `Object`, as shown in the following code:

```
private class ListNode
{
    private Object data;
    private ListNode link;
    public ListNode()
    {
        link = null;
        data = null;
    }
    public ListNode(Object newData, ListNode linkValue)
```

A list node having  
Object data

```

{
    data = newData;
    link = linkValue;
}

```

Because an object of any class type is also of type `Object`, you can store any kinds of objects in a linked list with nodes of this kind.

Instead of using `Object` as the data type of the list items, you can use a generic data type. Section 12.3 of this chapter will show you how. By using a generic data type, you restrict the objects in the linked list to having the same data type.

Sometimes it is handy to have a reference to the last node in a linked list. This last node is often called the **tail node** of the list, so the linked-list definition might begin as follows:

The last node in a linked list is its tail node

```

public class StringLinkedListWithTail
{
    private ListNode head;
    private ListNode tail;
    . . .
}

```

The constructors and methods must be modified to accommodate the new reference, `tail`, but the details of doing so are routine.

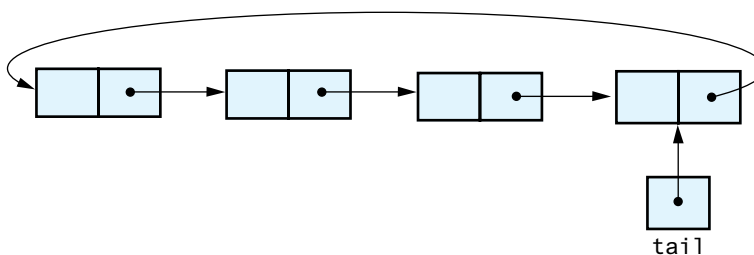
You have seen that the link portion of the last node in a linked list contains `null`. In a **circular linked list**, this link is instead a reference to the first node. Thus, no link in a circular linked list is `null`. Such a linked list still has a beginning and an end, as well as an external reference to either its first node or its last node. Figure 12.10 illustrates a circular linked list that has an external reference, `tail`, to its last node. Note that `tail.link` is then a reference to the list's first node.

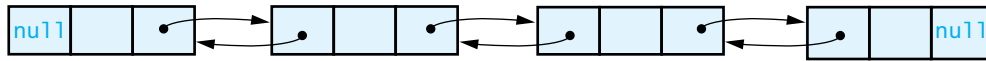
A linked list can be circular

An ordinary linked list allows you to follow its links and move down the list in only one direction. A **doubly linked list** has two links in each node: one link is a reference to the next node and one is a reference to the previous node. Figure 12.11 illustrates a doubly linked list.

A linked list can be doubly linked

**FIGURE 12.10** A Circular Linked List



**FIGURE 12.11 A Doubly Linked List**

The node class for a doubly linked list can begin as follows:

```
private class ListNode
{
    private Object data;
    private ListNode next;
    private ListNode previous;
    . . .
```

The constructors and some of the methods in the doubly linked list class will have somewhat different definitions than those in the singly linked case in order to accommodate the extra link.

A **circular doubly linked list** is also possible. In this structure, the previous link of the first node references the last node and the next link of the last node references the first node.

## Other Linked Data Structures

We have discussed a few kinds of linked lists as examples of linked data structures. Here are several more linked data structures that you will surely encounter in later studies.

### The ADT stack

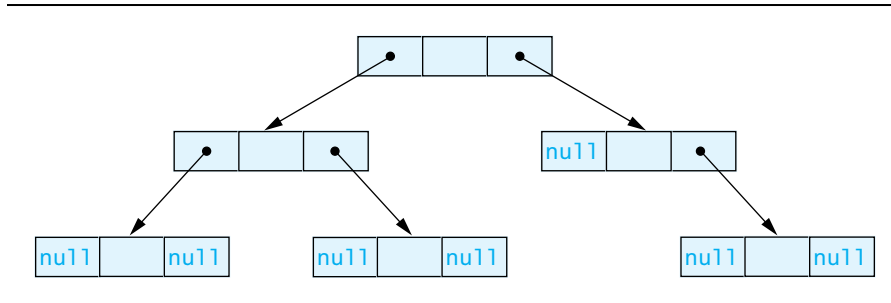
A **stack** is an ADT whose implementation is not necessarily a linked data structure, but it can be implemented as one. A stack organizes its data so that items are removed in the reverse of the order in which they were inserted. So if you insert "one", then "two", and then "three" into a stack and then remove them, they will come out in the order "three", then "two", and finally "one". A linked list that inserts and deletes only at the beginning of the list—such as the one in Listing 12.5—behaves like a stack, and so can be an implementation of a stack.

### A binary tree

Another common and powerful data structure is a **tree**. In a tree, each node leads to several other nodes. The most common form of a tree is a **binary tree**, in which each node has links to at most two other nodes. This tree has the same kind of nodes as a doubly linked list, but they are used in a very different way. A binary tree can be represented diagrammatically, as shown in Figure 12.12.

It is not an accident that we have no references leading back up the tree in our diagram of a binary tree. In a tree, the nodes must be arranged without any loops.

The top node in a binary tree is known as the **root node**, and there is normally a reference to this root node, just as there is a reference to the head node of a linked list. Every node can be reached from the root node by following suitable links.

**FIGURE 12.12** A Binary Tree

## 12.4 GENERICS

*There are no favorites in my office. I treat them all with the same general inconsideration.*

—LYNDON B. JOHNSON, 36th President of the United States.

As we mentioned earlier, beginning with version 5.0, Java allows class definitions to include parameters for data types. These parameters are called **generics**. This section gives a brief introduction to this topic. Programming with generics can be subtle and requires care. To do serious programming with generics, you may want to consult a more advanced text.

### The Basics

Classes and methods can use a **type parameter** instead of a specific data type. When Generics allows you to parameterize a data type using such a class or method, a programmer plugs in any class type for the type parameter to produce a specific class type or method for that particular instance. For example, Listing 12.11 shows a very simple class definition that uses a type parameter *T*. Notice the angle bracket notation around *T* in the class heading. You can use any nonkeyword identifier for the type parameter; you need not use *T*. However, by convention, type parameters start with an uppercase letter, and there seems to be a tradition of using a single letter for these parameters. Starting with an uppercase letter makes sense, since only a class type may be plugged in for the type parameter. The tradition of using a single letter is not as compelling.

Generics allows you to parameterize a data type

When writing a class or method that uses a type parameter, you can use the parameter almost anywhere that you can use a class type. For example, the class `Sample` in Listing 12.11 uses *T* as the data type of the instance variable `data`, the method parameter `newValue`, and the return type of the method `getData`. You cannot, however, use a type parameter to create a new object. So even though `data` is of type *T* in the class `Sample`, you cannot write

```
data = new T(); //Illegal!
```



**LISTING 12.11** A Class Definition That Uses a Type Parameter

---

```
public class Sample<T>
{
    private T data;

    public void setData(T newValue)
    {
        data = newValue;
    }
    public T getData()
    {
        return data;
    }
}
```

---

Moreover, you cannot use a type parameter when allocating memory for an array. Although you can declare an array by writing a statement such as

```
T[] anArray; //Valid declaration of an array
```

you cannot write

```
anArray = new T[20]; //Illegal!
```

**GOTCHA** You Cannot Use a Type Parameter Everywhere You Can Use a Type Name

Within the definition of a parameterized class definition, there are places where a type name is allowed but a type parameter is not. You cannot use a type parameter in simple expressions that use `new` to create a new object or to allocate memory for an array. For example, the following expressions are illegal within the definition of a parameterized class definition whose type parameter is `T`:

```
new T(); //Illegal!
new T[20]; //Illegal!
```

A class definition that uses a type parameter is stored in a file and compiled just like any other class. For example, the parameterized class shown in Listing 12.11 would be stored in a file named `Sample.java`.

Once the parameterized class is compiled, it can be used like any other class, except that when using it in your code, you must specify a class type to replace the type parameter. For example, the class `Sample` from Listing 12.11 could be used as follows:

```
Sample<String> sample1 = new Sample<String>();
sample1.setData("Hello");
Sample<Species> sample2 = new Sample<Species>();
Species creature = new Species();
. . . <Some code to set the data for the object creature>
sample1.setData(creature);
```

Notice the angle bracket notation for the actual class type that replaces the type parameter. The class `Species` could be as defined in Chapter 5, but the details do not matter; it could be any class you define.

You should now realize that the class `ArrayList`, which we discussed at the beginning of this chapter, is a parameterized class. Recall that we created an instance of `ArrayList` by writing, for example,

```
ArrayList<String> list = new ArrayList<String>(20);
```

Here `String` replaces the type parameter in the definition of the class.

### **RECAP** Class Definitions Having a Type Parameter

You can define classes that use a parameter instead of an actual class type. You write the type parameter within angle brackets right after the class name in the heading of the class definition. You can use any nonkeyword identifier for the type parameter, but by convention, the type parameter starts with an uppercase letter.

You use the type parameter within the class definition in the same way that you would use an actual class type, except that you cannot use it in conjunction with `new`.

#### **EXAMPLE**

See Listing 12.11.

### **RECAP** Using a Class Whose Definition Has a Type Parameter

You create an object of a parameterized class in the same way that you create an object of any other class, except that you specify an actual class type, rather than the type parameter, within angle brackets after the class name.

#### **EXAMPLE**

```
Sample<String> anObject = new Sample<String>();
anObject.setData("Hello");
```

**GOTCHA** You Cannot Plug in a Primitive Type for a Type Parameter

When you create an object of a class that has a type parameter, you cannot plug in a primitive type, such as `int`, `double`, or `char`. For example, the following statement will cause a syntax error:

```
ArrayList<int> aList = new ArrayList<int>(20); //Illegal!
```

The type plugged in for a type parameter must be a class type. ■

**PROGRAMMING TIP** Compile with the `-Xlint` Option

There are many pitfalls that you can encounter when using type parameters. If you compile with the `-Xlint` option, you will receive more informative diagnostics of any problems or potential problems in your code. For example, the class `Sample` in Listing 12.11 should be compiled as follows:

```
javac -Xlint Sample.java
```

If you are using an integrated development environment, or IDE, to compile your programs, check your documentation to see how to set a compiler option.

When compiling with the `-Xlint` option, you will get more warnings than you would otherwise get. A warning is not an error, and if the compiler gives only warnings and no error message, the class has compiled and can be used. However, in most cases be sure you understand the warning and feel confident that it does not indicate a problem, or else change your code to eliminate the warning. ■

**PROGRAMMING EXAMPLE****A Generic Linked List**


Let's revise the definition of the linked-list class that appears in Listing 12.7 to use a type parameter, `E`, instead of the type `String` as the type of data stored on the list. Listing 12.12 shows the result of this revision.

Notice that the constructor heading looks just as it would in a nonparameterized class. It does not include `<E>`, the type parameter within angle brackets, after the constructor's name. This is counterintuitive to many people. While a constructor—like any other method in the class—can use the type parameter within its body, its heading does not include the type parameter.

Likewise, the inner class `ListNode` does not have `<E>` after its name in its heading, but `ListNode` does use the type parameter `E` within its definition. These are correct as shown, however, because `ListNode` is an inner class and can access the type parameter of its outer class.

**LISTING 12.12 A Generic Linked-List Class (part 1 of 2)**

---

```
import java.util.ArrayList;
public class LinkedList2<E>
{
    private ListNode head;
    public LinkedList2( ) 
    {
        head = null;
    }
    public void showList( )
    {
        ListNode position = head;
        while (position != null)
        {
            System.out.println(position.getData( ));
            position = position.getLink( );
        }
    }
    public int length( )
    {
        int count = 0;
        ListNode position = head;
        while (position != null)
        {
            count++;
            position = position.getLink( );
        }
        return count;
    }
    public void addANodeToStart(E addData)
    {
        head = new ListNode(addData, head);
    }
    public void deleteHeadNode( )
    {
        if (head != null)
        {
            head = head.getLink( );
        }
        else
        {
            System.out.println("Deleting from an empty list.");
            System.exit(0);
        }
    }
    public boolean onList(E target)
    {
        return find(target) != null;
    }
}
```

*Constructor headings do not include the type parameter.*

(continued)

**LISTING 12.12 A Generic Linked-List Class** *(part 2 of 2)*

```

private ListNode find(E target)
{
    boolean found = false;
    ListNode position = head;
    while (position != null)
    {
        E dataAtPosition = position.getData();
        if (dataAtPosition.equals(target))
            found = true;
        else
            position = position.getLink();
    }
    return position;
}
private ArrayList<E> toArrayList()
{
    ArrayList<E> list = new ArrayList<E>(length());
    ListNode position = head;
    while (position != null)
    {
        list.add(position.data);
        position = position.link;
    }
    return list;
}
private class ListNode
{
    private E data;
    private ListNode link;
    public ListNode()
    {
        link = null;
        data = null;
    }
    public ListNode(E newData, ListNode linkValue)
    {
        data = newData;
        link = linkValue;
    }
    public E getData()
    {
        return data;
    }
    public ListNode getLink()
    {
        return link;
    }
}
}

```

The inner class heading has no type parameter.

However, the type parameter is used within the definition of the inner class.

The linked-list class in Listing 12.7 can have a method named `toArray`—shown in Listing 12.8—that returns an array containing the same data as the linked list. However, our generic version of a linked list cannot implement this same method. If we were to translate `toArray` in Listing 12.8 so that we could include it in our generic linked list in Listing 12.12, it would begin as follows:

```
public E[] toArray()
{
    E[] anArray = new E[length()]; //Illegal
    . . .
}
```

As the comment indicates, the expression `new E[length()]` is not allowed. You cannot include the type parameter in certain situations, and this, unfortunately, is one of them. Thus, you simply cannot have the method `toArray` in the generic linked list. Instead, Listing 12.12 defines the method `toArrayList`, which returns an instance of the class `ArrayList`. Note that its definition contains a loop like the one in `toArray`.

Listing 12.13 contains a simple example of using our parameterized linked list.



**VideoNote**  
Creating classes that use  
generics

### LISTING 12.13 Using the Generic Linked list

```
public class LinkedList2Demo
{
    public static void main(String[] args)
    {
        LinkedList2<String> stringList = new LinkedList2<String>( );
        stringList.addANodeToStart("Hello");
        stringList.addANodeToStart("Good-bye");
        stringList.showList( );
        LinkedList2<Integer> numberList = new LinkedList2<Integer>( );
        for (int i = 0; i < 5; i++)
            numberList.addANodeToStart(i);
        numberList.deleteHeadNode();
        numberList.showList( );
    }
}
```

#### Screen Output

```
Good-bye
Hello
3
2
1
0
```

**GOTCHA** A Constructor Name in a Generic Class Has No Type Parameter

The class name in a parameterized class definition has a type parameter attached—for example, `LinkedList<E>` in Listing 12.12. This can mislead you into thinking you need to use the type parameter in the heading of the constructor definition, but that is not the case. For example, you use

```
public LinkedList()
```

instead of

```
public LinkedList<E>() //Illegal
```

However, you can use a type parameter, such as `E`, within the body of the constructor's definition. ■

**FAQ** Can a class definition have more than one type parameter?

Yes, you can define a class with more than one type parameter. To do so, you write the type parameters one after the other, separated by commas, with one pair of angle brackets. For example, the class `Pair` could begin as follows:

```
public class Pair<S, T>
```

You would use `S` and `T` within the definition of `Pair` as you would actual data types. This class allows you to create pairs of objects that have different data types.

**REMEMBER** An Inner Class Has Access to the Type Parameter of Its Outer Class

An inner class can access the type parameter of its outer class. In Listing 12.12, for example, the definition of the inner class `ListNode` uses the type parameter `E`, which is the type parameter of its outer class `LinkedList`, even though `ListNode` does not have `<E>` after its name in its heading.

**PROGRAMMING TIP** Defining Type Parameters in an Inner Class

An inner class can define its own type parameters, but they must be distinct from any defined in the outer class. For example, we could define the outer and inner classes in Listing 12.12 as follows:

```

public class LinkedList<E>
{
    private ListNode<E> head;
    . . .
    private class ListNode<T>
    {
        private T data;
        private ListNode<T> link;
        . . .
        public ListNode (T newData, ListNode<T> linkValue)
        { . . .
        }
    }
}

```

Notice how the inner class `ListNode` uses `T` consistently as its type parameter and the outer class `LinkedList` has `E` as its type parameter. When `ListNode` is used as a data type within this outer class, it is written as `ListNode<E>`, not `ListNode<T>`. Also, note that we have no reason to give `ListNode` its own type parameter in this example, other than to show the syntax you would use if you wanted to do so. ■

## SELF-TEST QUESTIONS

29. Revise the definition of the class `ListNode` in Listing 12.4 so that it uses a type parameter instead of the type `String`. Note that the class is, and should remain, public. Rename the class to `ListNode2`.
30. Using the definition of the `ListNode2` class from the previous question, how would you create a `ListNode2` object named `node` that stores a value of type `Integer`?

## 12.5 GRAPHICS SUPPLEMENT

This section introduces the **Scene Builder**, a graphical program to help you build complex GUI applications.

### Building JavaFX Applications with the Scene Builder

Building complex interfaces can be tedious and difficult to visualize when directly coding the layout panes. To assist with UI development Oracle has released the JavaFX Scene Builder. If you are using an IDE then the Scene Builder may already be installed on your system. Oracle no longer releases binary executables of the Scene Builder (you have to build it from source code) but free binaries can be downloaded from Gluon Labs at <http://gluonhq.com/>



**VideoNote**  
Introduction to the Scene  
Builder