SRA 7.10 A command-line argument consists of data included on the command line when the interpreter is invoked to execute the program. Command-line arguments are another way to provide input to a program. They are accessed using the array of strings that is passed into the `main` method as a parameter.

SRA 7.11 A Java method can be defined to accept a variable number of parameters by using an ellipsis (. . .) in the formal parameter list. When several values are passed to the method, they are automatically converted to an array. This allows the method to be written in terms of array processing without forcing the calling method to create the array.

SRA 7.12 A multidimensional array is implemented in Java as an array of array objects. The arrays that are elements of the outer array could also contain arrays as elements. This nesting process could continue for as many levels as needed.

8 Inheritance

CHAPTER OBJECTIVES

- Explore the derivation of new classes from existing ones.
- Define the concept and purpose of method overriding.
- Discuss the design of class hierarchies.
- Examine the purpose and use of abstract classes.
- Discuss the issue of visibility as it relates to inheritance.
- Discuss object-oriented design in the context of inheritance.

This chapter explains inheritance, a fundamental technique for organizing and creating classes. It is a simple but powerful idea that influences the way we design object-oriented software and enhances our ability to reuse classes in other situations and programs. In this chapter we explore the technique for creating subclasses and class hierarchies, and we discuss a technique for overriding the definition of an inherited method. We examine the `protected` modifier and discuss the effect that all visibility modifiers have on inherited attributes and methods.

8.1 Creating Subclasses

In our introduction to object-oriented concepts in Chapter 1 we presented the analogy that a class is to an object what a blueprint is to a house. In subsequent chapters we reinforced that idea, writing classes that define a set of similar objects. A class establishes the characteristics and behaviors of an object but reserves no memory space for variables (unless those variables are declared as `static`). Classes are the plan, and objects are the embodiment of that plan.

Many houses can be created from the same blueprint. They are essentially the same house in different locations with different people living in them. Now suppose you want a house that is similar to another but has some different or additional features. You want to start with the same basic blueprint but modify it to suit new, slightly different needs. Many housing developments are created this way. The houses in the development have the same core layout, but they have unique features. For instance, they might all be split-level homes with the same basic room configuration, but some have a fireplace or full basement, whereas others do not, and some have an upgraded gourmet kitchen instead of the standard version.

It's likely that the housing developer commissioned a master architect to create a single blueprint to establish the basic design of all houses in the development and then a series of new blueprints that include variations designed to appeal to different buyers. The act of creating the series of blueprints was simplified because they all begin with the same underlying structure, while the variations give them unique characteristics that may be important to the prospective owners.

KEY CONCEPT
Inheritance is the process of deriving a new class from an existing one.

Creating a new blueprint that is based on an existing blueprint is analogous to the object-oriented concept of *inheritance*, which is the process in which a new class is derived from an existing one. Inheritance is a powerful software development technique and a defining characteristic of object-oriented programming.

Via inheritance, the new class automatically contains the variables and methods in the original class. Then, to tailor the class as needed, the programmer can add new variables and methods to the derived class or modify the inherited ones.

KEY CONCEPT
One purpose of inheritance is to reuse existing software.

In general, creating new classes via inheritance is faster, easier, and cheaper than writing them from scratch. Inheritance is one way to support the idea of *software reuse*. By using existing software components to create new ones, we capitalize on the effort that went into the design, implementation, and testing of the existing software.

Keep in mind that the word *class* comes from the idea of classifying groups of objects with similar characteristics. Classification schemes often use levels of classes that are related to each other. For example, all mammals share certain characteristics, such as being warm-blooded. Now consider a subset of mammals, such as horses. All horses are mammals and have all of the characteristics of mammals, but they also have unique features that make them different from other mammals, such as dogs.

If we translate this idea into software terms, an existing class called `Mammal` would have certain variables and methods that describe the state and behavior of mammals. A `Horse` class could be derived from the existing `Mammal` class, automatically inheriting the variables and methods contained in `Mammal`. The `Horse` class can refer to the inherited variables and methods as if they had been declared locally in that class. New variables and methods can then be added to the derived class to distinguish a horse from other mammals.


VideoNote
Overview of inheritance

The original class that is used to derive a new one is called the *parent class*, *superclass*, or *base class*. The derived class is called a *child class*, or *subclass*. In UML, inheritance is represented by an arrow with an open arrowhead pointing from the child class to the parent, as shown in Figure 8.1.

The process of inheritance should establish an *is-a relationship* between two classes. That is, the child class should be a more specific version of the parent. For example, a horse is a mammal. Not all mammals are horses, but all horses are mammals. For any class X that is derived from class Y, you should be able to say that "X is a Y." If such a statement doesn't make sense, then that relationship is probably not an appropriate use of inheritance.

KEY CONCEPT
Inheritance creates an is-a relationship between the parent and child classes.

Let's look at an example. The `Words` program shown in Listing 8.1 instantiates an object of class `Dictionary`, which is derived from a class called `Book`. In the `main` method, three methods are invoked through the `Dictionary` object: two that were declared locally in the `Dictionary` class and one that was inherited from the `Book` class.

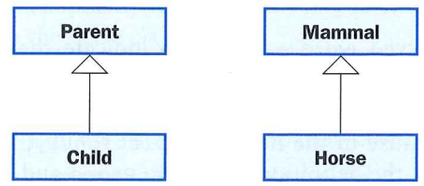

FIGURE 8.1 Inheritance relationships in UML

LISTING 8.1

```
//********************************************************************
//  Words.java       Java Foundations
//
//  Demonstrates the use of an inherited method.
//********************************************************************

public class Words
{
   //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Dictionary webster = new Dictionary();

      System.out.println("Number of pages: " + webster.getPages());

      System.out.println("Number of definitions: " +
                         webster.getDefinitions());

      System.out.println("Definitions per page: " +
                         webster.computeRatio());
   }
}
```

OUTPUT

```
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35.0
```

Java uses the reserved word `extends` to indicate that a new class is being derived from an existing class. The `Book` class (shown in Listing 8.2) is used to derive the `Dictionary` class (shown in Listing 8.3 on page 384) simply by using the `extends` clause in the header of `Dictionary`. The `Dictionary` class automatically inherits the definition of the `setPages` and `getPages` methods, as well as the `pages` variable. It is as if those methods and the `pages` variable were

LISTING 8.2

```
//********************************************************************
//  Book.java       Java Foundations
//
//  Represents a book. Used as the parent of a derived class to
//  demonstrate inheritance.
//********************************************************************

public class Book
{
   protected int pages = 1500;

   //-----------------------------------------------------------------
   //  Pages mutator.
   //-----------------------------------------------------------------
   public void setPages(int numPages)
   {
      pages = numPages;
   }

   //-----------------------------------------------------------------
   //  Pages accessor.
   //-----------------------------------------------------------------
   public int getPages()
   {
      return pages;
   }
}
```

declared inside the `Dictionary` class. Note that in the `Dictionary` class, the `computeRatio` method explicitly references the `pages` variable, even though that variable is declared in the `Book` class.

Also note that although the `Book` class is needed to create the definition of `Dictionary`, no `Book` object is ever instantiated in the program. An instance of a child class does not rely on an instance of the parent class.

Inheritance is a one-way street. The `Book` class cannot use variables or methods that are declared explicitly in the `Dictionary` class. For instance, if we created an

LISTING 8.3

```
//********************************************************************
//  Dictionary.java       Java Foundations
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  inheritance.
//********************************************************************

public class Dictionary extends Book
{
   private int definitions = 52500;

   //-----------------------------------------------------------------
   //  Prints a message using both local and inherited values.
   //-----------------------------------------------------------------
   public double computeRatio()
   {
      return definitions/pages;
   }

   //-----------------------------------------------------------------
   //  Definitions mutator.
   //-----------------------------------------------------------------
   public void setDefinitions(int numDefinitions)
   {
      definitions = numDefinitions;
   }

   //-----------------------------------------------------------------
   //  Definitions accessor.
   //-----------------------------------------------------------------
   public int getDefinitions()
   {
      return definitions;
   }
}
```

object from the `Book` class, it could not be used to invoke the `setDefinitions` method. This restriction makes sense because a child class is a more specific version of the parent class. A dictionary has pages because all books have pages, but even though a dictionary has definitions, not all books do.

Deriving a Class

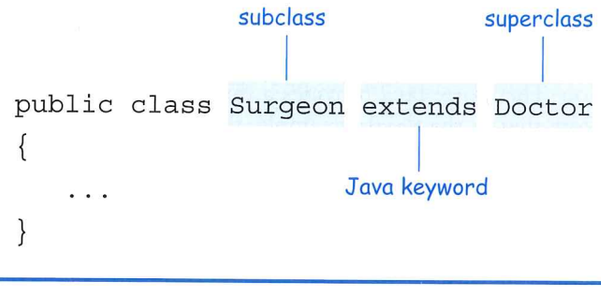

Figure 8.2 shows the inheritance relationship between the `Book` and `Dictionary` classes.

The `protected` Modifier

As we've seen, visibility modifiers are used to control access to the members of a class. Visibility plays an important role in the process of inheritance as well. Any public method or variable in a parent class can be explicitly referenced by name in

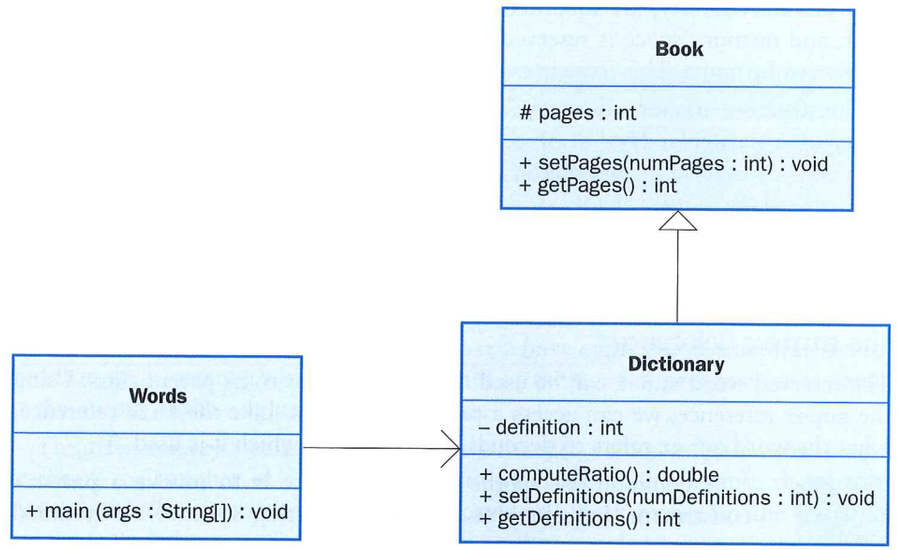

FIGURE 8.2 A UML class diagram for the `Words` program

the child class and through objects of that child class. On the other hand, private methods and variables of the parent class cannot be referenced in the child class or through an object of the child class.

This situation causes a dilemma. If we declare a variable with public visibility so that a derived class can reference it, we violate the principle of encapsulation. Therefore, Java provides a third visibility modifier: `protected`. Note that in the `Words` example, the variable `pages` is declared with protected visibility in the `Book` class. When a variable or method is declared with protected visibility, a derived class can reference it. And protected visibility allows the class to retain some encapsulation properties. The encapsulation with protected visibility is not as tight as it would be if the variable or method were declared private, but it is better than if it were declared public. Specifically, a variable or method declared with protected visibility may be accessed by any class in the same package. The relationships among all Java modifiers are explained completely in Appendix E.

KEY CONCEPT
Protected visibility provides the best possible encapsulation that permits inheritance.

In a UML diagram, protected visibility can be indicated by preceding the protected member with a hash mark (#). The `pages` variable of the `Book` class has this annotation in Figure 8.2.

Each variable or method retains the effect of its original visibility modifier. For example, the `setPages` method is still considered to be public in its inherited form in the `Dictionary` class.

Let's be clear about our terms. All methods and variables, even those declared with private visibility, are inherited by the child class. That is, their definitions exist, and memory space is reserved for the variables. It's just that they can't be referenced by name. This issue is explored in more detail in Section 8.4.

Constructors are not inherited. Constructors are special methods that are used to set up a particular type of object, so it doesn't make sense for a class called `Dictionary` to have a constructor called `Book`. But you can imagine that a child class may want to refer to the constructor of the parent class, which is one of the reasons for the `super` reference, described next.

The `super` Reference

The reserved word `super` can be used in a class to refer to its parent class. Using the `super` reference, we can access a parent's members. Like the `this` reference, what the word `super` refers to depends on the class in which it is used.

KEY CONCEPT
A parent's constructor can be invoked using the `super` reference.

A common use of the `super` reference is to invoke a parent's constructor. Let's look at an example. Listing 8.4 shows a modified version of the `Words` program, in which we use a class called `Book2` (shown in Listing 8.5 on page 388) as the parent of the derived class

LISTING 8.4

```
//********************************************************************
//  Words2.java       Java Foundations
//
//  Demonstrates the use of the super reference.
//********************************************************************

public class Words2
{
  //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Dictionary2 webster = new Dictionary2 (1500, 52500);

      System.out.println("Number of pages: " + webster.getPages());

      System.out.println("Number of definitions: " +
                         webster.getDefinitions());

      System.out.println("Definitions per page: " +
                         webster.computeRatio());
   }
}
```

OUTPUT

```
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35.0
```

`Dictionary2` (shown in Listing 8.6 on page 389). However, unlike earlier versions of these classes, `Book2` and `Dictionary2` have explicit constructors used to initialize their instance variables. The output of the `Words2` program is the same as the output of the original `Words` program.

The `Dictionary2` constructor takes two integer values as parameters, representing the number of pages and definitions in the book. Because the `Book2` class already has a constructor that performs the work to set up the parts of the dictionary that were inherited, we rely on that constructor to do that work. However, since the constructor is not inherited, we cannot invoke it directly, and

LISTING 8.5

```
//********************************************************************
//  Book2.java       Java Foundations
//
//  Represents a book. Used as the parent of a derived class to
//  demonstrate inheritance and the use of the super reference.
//********************************************************************

public class Book2
{
   protected int pages;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the book with the specified number of
   //  pages.
   //-----------------------------------------------------------------
   public Book2(int numPages)
   {
      pages = numPages;
   }

  //-----------------------------------------------------------------
   //  Pages mutator.
   //-----------------------------------------------------------------
   public void setPages(int numPages)
   {
      pages = numPages;
   }

   //-----------------------------------------------------------------
   //  Pages accessor.
   //-----------------------------------------------------------------
   public int getPages()
   {
      return pages;
   }
}
```

so we use the `super` reference to invoke it in the parent class. The `Dictionary2` constructor then proceeds to initialize its `definitions` variable.

In this example, it would have been just as easy to set the `pages` variable explicitly in the `Dictionary2` constructor instead of using `super` to call the `Book2` constructor. However, it is good practice to let each class "take care of itself." If

LISTING 8.6

```
//********************************************************************
//  Dictionary2.java       Java Foundations
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  the use of the super reference.
//********************************************************************

public class Dictionary2 extends Book2
{
   private int definitions;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the dictionary with the specified number
   //  of pages and definitions.
   //-----------------------------------------------------------------
   public Dictionary2(int numPages, int numDefinitions)
   {
      super(numPages);

      definitions = numDefinitions;
   }

   //-----------------------------------------------------------------
   //  Prints a message using both local and inherited values.
   //-----------------------------------------------------------------
   public double computeRatio()
   {
      return definitions/pages;
   }

   //-----------------------------------------------------------------
   //  Definitions mutator.
   //-----------------------------------------------------------------
   public void setDefinitions(int numDefinitions)
   {
      definitions = numDefinitions;
   }

   //-----------------------------------------------------------------
   //  Definitions accessor.
   //-----------------------------------------------------------------
   public int getDefinitions()
   {
      return definitions;
   }
}
```

we choose to change the way that the `Book2` constructor sets up its `pages` variable, we also have to remember to make that change in `Dictionary2`. When we use the `super` reference, a change made in `Book2` is automatically reflected in `Dictionary2`.

A child's constructor is responsible for calling its parent's constructor. Generally, the first line of a constructor should use the `super` reference call to a constructor of the parent class. If no such call exists, Java will automatically make a call to `super` with no parameters at the beginning of the constructor. This rule ensures that a parent class initializes its variables before the child class constructor begins to execute. Using the `super` reference to invoke a parent's constructor can be done only in the child's constructor, and if included, it must be the first line of the constructor.

The `super` reference can also be used to reference other variables and methods defined in the parent's class. We use this technique in later sections of this chapter.

Multiple Inheritance

Java's approach to inheritance is called *single inheritance*. This term means that a derived class can have only one parent. Some object-oriented languages allow a child class to have multiple parents. This approach, which is called *multiple inheritance*, is occasionally useful for describing objects that could share characteristics of more than one class. For example, suppose we had a class `Car` and a class `Truck` and we wanted to create a new class called `PickupTruck`. A pickup truck is somewhat like a car and somewhat like a truck. With single inheritance, we must decide whether it is better to derive the new class from `Car` or from `Truck`. With multiple inheritance, it can be derived from both, as shown in Figure 8.3.

Multiple inheritance works well in some situations, but it comes with a price. What if both `Truck` and `Car` have methods with the same name? Which method would `PickupTruck` inherit? The answer to this question is complex, and it depends on the rules of the language that supports multiple inheritance.

The designers of the Java language explicitly decided not to support multiple inheritance. Java *interfaces*, described in Chapter 9, provide the best features of multiple inheritance, without the added complexity.

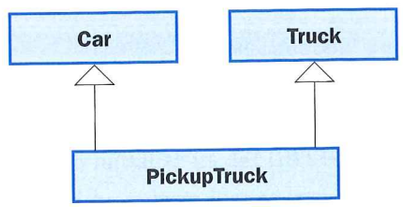

FIGURE 8.3 Multiple inheritance

8.2 Overriding Methods

When a child class defines a method with the same name and signature as a method in the parent class, we say that the child's version *overrides* the parent's version in favor of its own. The need for overriding occurs often in inheritance situations.

The program in Listing 8.7 provides a simple demonstration of method overriding in Java. The `Messages` class contains a `main` method that instantiates two objects: one from class `Thought` and one from class `Advice`. The `Thought` class is the parent of the `Advice` class.

KEY CONCEPT
A child class can override (redefine) the parent's definition of an inherited method.

Both the `Thought` class (shown in Listing 8.8 on page 392) and the `Advice` class (shown in Listing 8.9 on page 393) contain a definition

LISTING 8.7

```
//********************************************************************
//  Messages.java       Java Foundations
//
//  Demonstrates the use of an overridden method.
//********************************************************************

public class Messages
{
   //-----------------------------------------------------------------
   //  Creates two objects and invokes the message method in each.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Thought parked = new Thought();
      Advice dates = new Advice();

      parked.message();

      dates.message();   // overridden
   }
}
```

OUTPUT

```
I feel like I'm diagonally parked in a parallel universe.
Warning: Dates in calendar are closer than they appear.
I feel like I'm diagonally parked in a parallel universe.
```

LISTING 8.8

```
//********************************************************************
//  Thought.java       Java Foundations
//
//  Represents a stray thought. Used as the parent of a derived
//  class to demonstrate the use of an overridden method.
//********************************************************************

public class Thought
{
   //-----------------------------------------------------------------
   //  Prints a message.
   //-----------------------------------------------------------------
   public void message()
   {
      System.out.println("I feel like I'm diagonally parked in a " +
                         "parallel universe.");

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

for a method called `message`. The version of `message` defined in the `Thought` class is inherited by `Advice`, but `Advice` overrides it with an alternative version. The new version of the method prints out an entirely different message and then invokes the parent's version of the `message` method using the `super` reference.

The object that is used to invoke a method determines which version of the method is actually executed. When `message` is invoked using the `parked` object in the `main` method, the `Thought` version of `message` is executed. When `message` is invoked using the `dates` object, the `Advice` version of `message` is executed.

A method can be defined with the `final` modifier. A child class cannot override a final method. This technique is used to ensure that a derived class uses a particular definition of a method.

Method overriding is a key element in object-oriented design. It allows two objects that are related by inheritance to use the same naming conventions for methods that accomplish the same general task in different ways. Overriding becomes even more important when it comes to polymorphism, which is discussed in Chapter 9.

LISTING 8.9

```
//********************************************************************
//  Advice.java       Java Foundations
//
//  Represents some thoughtful advice. Used to demonstrate the use
//  of an overridden method.
//********************************************************************

public class Advice extends Thought
{
   //-----------------------------------------------------------------
   //  Prints a message. This method overrides the parent's version.
   //-----------------------------------------------------------------
   public void message()
   {
      System.out.println("Warning: Dates in calendar are closer " +
                         "than they appear.");

      System.out.println();

      super.message();   // explicitly invokes the parent's version
   }
}
```

COMMON ERROR

Don't confuse method overriding with method overloading. Recall from Chapter 5 that method overloading occurs when two or more methods with the same name have distinct signatures (parameter lists). With method overloading, you end up with multiple methods with the same name in the same class. With method overriding, you are replacing a method in the child class with a new definition. Method overriding occurs across classes and affects methods with the same signature.

A related problem occurs when you mean to override a method but instead create an overloaded version in the child class. The method defined in the child class must match the signature of the method you intend to override. If it doesn't, you end up with the inherited method plus an overloaded version of the method that is newly defined in the child. It's possible that you may want that situation to occur. Just be aware of the distinction.

Shadowing Variables

It is possible, although not recommended, for a child class to declare a variable with the same name as one that is inherited from the parent. Note the distinction between redeclaring a variable and simply giving an inherited variable a particular value. If a variable of the same name is declared in a child class, it is called a *shadow variable*. This is similar in concept to the process of overriding methods but creates confusing subtleties.

Because an inherited variable is already available to the child class, there is usually no good reason to redeclare it. Someone reading code with a shadowed variable will find two different declarations that seem to apply to a variable used in the child class. This confusion causes problems and serves no useful purpose. A redeclaration of a particular variable name could change its type, but that is usually unnecessary. In general, shadow variables should be avoided.

8.3 Class Hierarchies

A child class derived from one parent can be the parent of its own child class. Furthermore, multiple classes can be derived from a single parent. Therefore, inheritance relationships often develop into *class hierarchies*. The diagram in Figure 8.4 shows a class hierarchy that includes the inheritance relationship between the `Mammal` and `Horse` classes, discussed earlier.

There is no limit to the number of children a class can have or to the number of levels to which a class hierarchy can extend. Two children of the same parent are called *siblings*. Although siblings share the characteristics passed on by their common parent, they are not related by inheritance because one is not used to derive the other.

In class hierarchies, common features should be kept as high in the hierarchy as reasonably possible. That way, the only characteristics explicitly established in a child class are those that make the class distinct from its parent and from its siblings. This

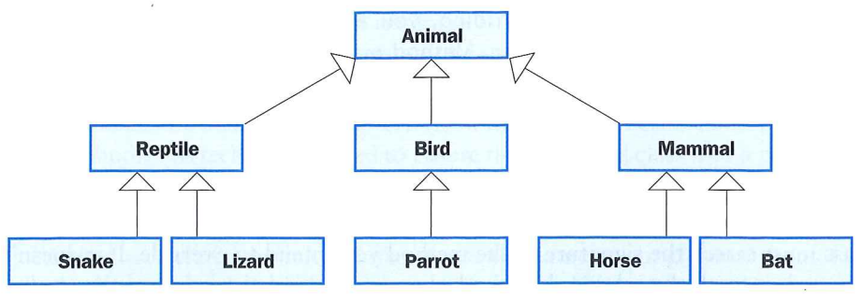

FIGURE 8.4 A class hierarchy

approach maximizes the potential to reuse classes. It also facilitates maintenance activities, because when changes are made to the parent, they are automatically reflected in the descendants. Always remember to maintain the is-a relationship when building class hierarchies.

KEY CONCEPT
The child of one class can be the parent of one or more other classes, creating a class hierarchy.

The inheritance mechanism is transitive. That is, a parent passes along a trait to a child class, and that child class passes it along to its children, and so on. An inherited feature might have originated in the immediate parent or, possibly, several levels higher in a more distant ancestor class.

KEY CONCEPT
Common features should be located as high in a class hierarchy as is reasonably possible.

There is no single best hierarchy organization for all situations. The decisions you make when you are designing a class hierarchy restrict and guide more detailed design decisions and implementation options, so you must make them carefully.

The class hierarchy shown in Figure 8.4 organizes animals by their major biological classifications, such as `Mammal`, `Bird`, and `Reptile`. In a different situation, however, it may be better to organize the same animals in a different way. For example, as shown in Figure 8.5, the class hierarchy might be organized around a function of the animals, such as their ability to fly. In this case, a `Parrot` class and a `Bat` class would be siblings derived from a general `FlyingAnimal` class. This class hierarchy is just as valid and reasonable as the original one. The goals of the programs that use the classes are the determining factor, guiding the programmer to a hierarchy design that is best for the situation.

The `Object` Class

In Java, all classes are derived ultimately from the `Object` class. If a class definition doesn't use the `extends` clause to derive itself explicitly from another class, then that class is automatically derived from the `Object` class by default. Therefore, this class definition:

```
class Thing
{
   // whatever
}
```

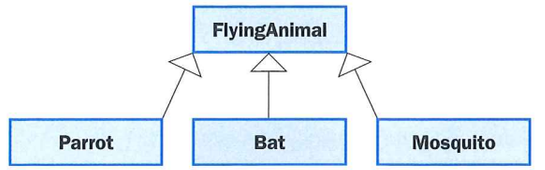

FIGURE 8.5 An alternative hierarchy for organizing animals

is equivalent to this one:

```
class Thing extends Object
{
   // whatever
}
```

Because all classes are derived from `Object`, all public methods of `Object` are inherited by every Java class. They can be invoked through any object created in any Java program. The `Object` class is defined in the `java.lang` package of the Java standard class library. Figure 8.6 lists some of the methods of the `Object` class.

KEY CONCEPT
All Java classes are derived, directly or indirectly, from the `Object` class.

As it turns out, we've been using `Object` methods quite often in our examples. The `toString` method, for instance, is defined in the `Object` class, so the `toString` method can be called on any object. As we've seen several times, when a `println` method is called with an object parameter, `toString` is called to determine what to print.

KEY CONCEPT
The `toString` and `equals` methods are inherited by every class in every Java program.

Thus, when we define a `toString` method in a class, we are actually overriding an inherited definition. The definition for `toString` that is provided by the `Object` class returns a string containing the object's class name followed by a numeric value that is unique for that object. Usually, we override the `Object` version of `toString` to fit our own needs. The `String` class has overridden the `toString` method so that it returns its stored string value.

VideoNote
Example using a class hierarchy

We are also overriding an inherited method when we define an `equals` method for a class. As we discussed in Chapter 4, the purpose of the `equals` method is to determine whether two objects are equal. The definition of the `equals` method provided by the `Object` class returns true if the two object references actually refer to the same object (that is, if they are aliases). Classes often override the inherited definition of the `equals` method in favor of a more appropriate definition.

```
boolean equals (Object obj)
    Returns true if this object is an alias of the specified object.

String toString ()
    Returns a string representation of this object.

Object clone ()
    Creates and returns a copy of this object.
```

FIGURE 8.6 Some methods of the `Object` class

For instance, the `String` class overrides `equals` so that it returns true only if both strings contain the same characters in the same order.

Abstract Classes

An *abstract class* represents a generic concept in a class hierarchy. As the name implies, an abstract class represents an abstract entity that is usually insufficiently defined to be useful by itself. Instead, an abstract class may contain a partial description that is inherited by all of its descendants in the class hierarchy. An abstract class is just like any other class, except that it may have some methods that have not been defined yet. Its children, which are more specific, fill in the gaps.

An abstract class cannot be instantiated and usually contains one or more *abstract methods*, which have no definition. That is, there is no body of code defined for an abstract method, and therefore it cannot be invoked. An abstract class might also contain methods that are not abstract, meaning that the method definition is provided as usual. And an abstract class can contain data declarations as usual.

KEY CONCEPT
An abstract class cannot be instantiated. It represents a concept on which other classes can build their definitions.

A class is declared as abstract by including the `abstract` modifier in the class header. Any class that contains one or more abstract methods must be declared as abstract. In abstract classes, the `abstract` modifier must be applied to each abstract method. A class declared as abstract does not have to contain abstract methods, however.

Consider the class hierarchy shown in Figure 8.7. The `Vehicle` class at the top of the hierarchy may be too generic for a particular application. Therefore, we may choose to implement it as an abstract class. In UML diagrams, the names of abstract classes and abstract methods are shown in italics.

Concepts that apply to all vehicles can be represented in the `Vehicle` class and are inherited by its descendants. That way, each of its descendants doesn't

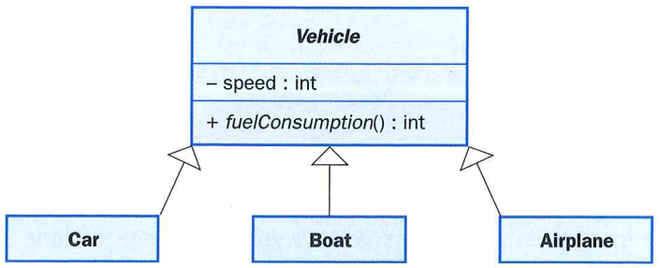

FIGURE 8.7 A vehicle class hierarchy

have to define the same concept redundantly (and perhaps inconsistently). For example, in Figure 8.7 we declare a variable called `speed` in the `Vehicle` class, and all specific vehicles below it in the hierarchy automatically have that variable because of inheritance. Any change we make to the representation of the speed of a vehicle is automatically reflected in all descendant classes. Similarly, in `Vehicle` we declare an abstract method called `fuelConsumption`, whose purpose is to calculate how quickly fuel is being consumed by a particular vehicle. The `Vehicle` class establishes that all vehicles consume fuel and provides a consistent method interface for computing that value. But implementation of the `fuelConsumption` method is left up to each subclass of `Vehicle`, which can tailor its method accordingly.

Some concepts don't apply to all vehicles, so we wouldn't represent those concepts at the `Vehicle` level. For instance, we wouldn't include a variable called `numberOfWheels` in the `Vehicle` class, because not all vehicles have wheels. The child classes for which wheels are appropriate can add that concept at the appropriate level in the hierarchy.

KEY CONCEPT
A class derived from an abstract parent must override all of its parent's abstract methods, or the derived class will also be considered abstract.

There are no restrictions on where in a class hierarchy an abstract class can be defined. Usually, abstract classes are located at the upper levels of a class hierarchy. However, it is possible to derive an abstract class from a nonabstract parent.

Usually, a child of an abstract class will provide a specific definition for an abstract method inherited from its parent. Note that this is just a specific case of overriding a method, giving a different definition from the one the parent provides. If a child of an abstract class does not give a definition for every abstract method that it inherits from its parent, then the child class is also considered abstract.

It would be a contradiction for an abstract method to be modified as `final` or `static`. Because a final method cannot be overridden in subclasses, an abstract final method would have no way of being given a definition in subclasses. A static method can be invoked using the class name without declaring an object of the class. Because abstract methods have no implementation, an abstract static method would make no sense.

DESIGN FOCUS

Choosing which classes and methods to make abstract is an important part of the design process. You should make such choices only after careful consideration. By using abstract classes wisely, you can create flexible, extensible software designs.

8.4 Visibility

As we mentioned earlier in this chapter, all variables and methods in a parent class, even those declared as private, are inherited by child classes. Private members exist for an object of a derived class, even though they can't be referenced directly. They can, however, be referenced indirectly.

Let's look at an example that demonstrates this situation. The program shown in Listing 8.10 contains a `main` method that instantiates a `Pizza` object and invokes a method to determine how many calories the pizza has per serving as a consequence of its fat content.

The `FoodItem` class shown in Listing 8.11 represents a generic type of food. The constructor of `FoodItem` accepts the number of grams of fat and the number of servings of that food. The `calories` method returns the number of calories due to fat, which the `caloriesPerServing` method invokes to help compute the number of fat calories per serving.

KEY CONCEPT
Private members are inherited by the child class but cannot be referenced directly by name. They may be used indirectly, however.

LISTING 8.10

```
//********************************************************************
//  FoodAnalyzer.java       Java Foundations
//
//  Demonstrates indirect access to inherited private members.
//********************************************************************

public class FoodAnalyzer
{
   //-----------------------------------------------------------------
   //  Instantiates a Pizza object and prints its calories per
   //  serving.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Pizza special = new Pizza(275);

      System.out.println("Calories per serving: " +
                         special.caloriesPerServing());
   }
}
```

OUTPUT

```
Calories per serving: 309
```

LISTING 8.11

```
//********************************************************************
//  FoodItem.java       Java Foundations
//
//  Represents an item of food. Used as the parent of a derived class
//  to demonstrate indirect referencing.
//********************************************************************

public class FoodItem
{
   final private int CALORIES_PER_GRAM = 9;
   private int fatGrams;
   protected int servings;

   //-----------------------------------------------------------------
   //  Sets up this food item with the specified number of fat grams
   //  and number of servings.
   //-----------------------------------------------------------------
   public FoodItem(int numFatGrams, int numServings)
   {
      fatGrams = numFatGrams;
      servings = numServings;
   }

   //-----------------------------------------------------------------
   //  Computes and returns the number of calories in this food item
   //  due to fat.
   //-----------------------------------------------------------------
   private int calories()
   {
      return fatGrams * CALORIES_PER_GRAM;
   }

   //-----------------------------------------------------------------
   //  Computes and returns the number of fat calories per serving.
   //-----------------------------------------------------------------
   public int caloriesPerServing()
   {
      return (calories() / servings);
   }
}
```


LISTING 8.12

```
//********************************************************************
//  Pizza.java        Java Foundations
//
//  Represents a pizza, which is a food item. Used to demonstrate
//  indirect referencing through inheritance.
//********************************************************************

public class Pizza extends FoodItem
{
   //-----------------------------------------------------------------
   //  Sets up a pizza with the specified amount of fat (assumes
   //  eight servings).
   //-----------------------------------------------------------------
   public Pizza(int fatGrams)
   {
      super(fatGrams, 8);
   }
}
```

The `Pizza` class, shown in Listing 8.12, is derived from the `FoodItem` class, but it adds no special functionality or data. Its constructor calls the constructor of `FoodItem` using the `super` reference, asserting that there are eight servings per pizza.

The `Pizza` object called `special` in the `main` method is used to invoke the method `caloriesPerServing`, which is defined as a public method of `FoodItem`. Note that `caloriesPerServing` calls `calories`, which is declared with private visibility. Furthermore, `calories` references the variable `fatGrams` and the constant `CALORIES_PER_GRAM`, which are also declared with private visibility.

Even though the `Pizza` class cannot explicitly reference `calories`, `fatGrams`, or `CALORIES_PER_GRAM`, these are available for use indirectly when the `Pizza` object needs them. A `Pizza` object cannot be used to invoke the `calories` method, but it can call a method that can be so used. Note that a `FoodItem` object was never created or needed.

8.5 Designing for Inheritance

As a major characteristic of object-oriented software, inheritance must be carefully and specifically addressed during software design. A little thought about inheritance relationships can lead to a far more elegant design, which pays huge dividends in the long term.

KEY CONCEPT
Software design must carefully and specifically address inheritance.

Throughout this chapter, several design issues have been addressed in the discussion of the nuts and bolts of inheritance in Java. The following list summarizes some of the inheritance issues that you should keep in mind during the program design stage:

- Every derivation should be an is-a relationship. The child should be a more specific version of the parent.
- Design a class hierarchy to capitalize on reuse, and on potential reuse in the future.
- As classes and objects are identified in the problem domain, find their commonality. Push common features as high in the class hierarchy as appropriate for consistency and ease of maintenance.
- Override methods as appropriate to tailor or change the functionality of a child.
- Add new variables to the child class as needed, but don't shadow (redefine) any inherited variables.
- Allow each class to manage its own data. Therefore, use the `super` reference to invoke a parent's constructor and to call overridden versions of methods if appropriate.
- Design a class hierarchy to fit the needs of the application, taking into account how it may be useful in the future.
- Even if there are no current uses for them, override general methods such as `toString` and `equals` appropriately in child classes so that the inherited versions don't inadvertently cause problems later.
- Use abstract classes to specify a common class interface for the concrete classes lower in the hierarchy.
- Use visibility modifiers carefully to provide the needed access in derived classes without violating encapsulation.

Restricting Inheritance

We've seen the `final` modifier used in declarations to create constants many times. The other uses of the `final` modifier involve inheritance and can have a significant influence on software design. Specifically, the `final` modifier can be used to curtail the abilities related to inheritance.

KEY CONCEPT
The `final` modifier can be used to restrict inheritance.

Earlier in this chapter, we mentioned that a method can be declared as `final`, which means it cannot be overridden in any classes that extend the one it is in. A final method is often used to insist that particular functionality be used in all child classes.

The `final` modifier can also be applied to an entire class. A final class cannot be extended at all. Consider the following declaration:

```
public final class Standards
{
    // whatever
}
```

Given this declaration, the `Standards` class cannot be used in the `extends` clause of another class. The compiler will generate an error message in such a case. The `Standards` class can be used normally, but it cannot be the parent of another class.

Using the `final` modifier to restrict inheritance abilities is a key design decision. It should be done in situations where a child class might otherwise be used to change functionality that you, as the designer, specifically want to be handled a certain way. This issue comes up again in the discussion of polymorphism in Chapter 9.

Summary of Key Concepts

- Inheritance is the process of deriving a new class from an existing one.
- One purpose of inheritance is to reuse existing software.
- Inheritance creates an is-a relationship between the parent and child classes.
- Protected visibility provides the best possible encapsulation that permits inheritance.
- A parent's constructor can be invoked using the `super` reference.
- A child class can override (redefine) the parent's definition of an inherited method.
- The child of one class can be the parent of one or more other classes, creating a class hierarchy.
- Common features should be located as high in a class hierarchy as is reasonably possible.
- All Java classes are derived, directly or indirectly, from the `Object` class.
- The `toString` and `equals` methods are inherited by every class in every Java program.
- An abstract class cannot be instantiated. It represents a concept on which other classes can build their definitions.
- A class derived from an abstract parent must override all of its parent's abstract methods, or the derived class will also be considered abstract.
- Private members are inherited by the child class but cannot be referenced directly by name. They may be used indirectly, however.
- Software design must carefully and specifically address inheritance.
- The `final` modifier can be used to restrict inheritance.

Summary of Terms

abstract class A class used to represent a generic concept in a class hierarchy. An abstract class cannot be instantiated.

abstract method A method header without a body, used to establish the existence of an operation before the implementation is available. A class that contains an abstract method is inherently abstract.

base class A class from which another is derived. Also called a parent class or superclass.

child class A class derived from a parent class. Also called a subclass.

class hierarchy The hierarchy formed by inheritance among multiple classes.

inheritance The process of deriving one class from another.

is-a relationship The relationship between two classes related by inheritance. The superclass is-a more specific version of the subclass.

multiple inheritance Inheritance in which a subclass can be derived from multiple parent classes. Java does not support multiple inheritance.

overriding Redefining a method that has been inherited from a parent class.

parent class A class from which another is derived. Also called a superclass or base class.

shadow variable An instance variable defined in a derived class that has the same name as a variable in the parent class.

single inheritance Inheritance in which a subclass can have only one parent. Java supports only single inheritance.

subclass A class derived from a superclass. Also called a child class.

superclass A class from which another is derived. Also called a parent class or base class.

Self-Review Questions

SR 8.1 Describe the relationship between a parent class and a child class.

SR 8.2 How does inheritance support software reuse?

SR 8.3 What relationship should every class derivation represent?

SR 8.4 What does the `protected` modifier accomplish?

SR 8.5 Why is the `super` reference important to a child class?

SR 8.6 What is the difference between single inheritance and multiple inheritance?

SR 8.7 Why would a child class override one or more of the methods of its parent class?

SR 8.8 What is the significance of the `Object` class?

SR 8.9 What is the role of an abstract class?

SR 8.10 Are all members of a parent class inherited by the child? Explain.

SR 8.11 How can the `final` modifier be used to restrict inheritance?

Exercises

EX 8.1 Draw a UML class diagram showing an inheritance hierarchy containing classes that represent different types of clocks. Show the variables and method names for two of these classes.

EX 8.2 Show an alternative diagram for the hierarchy in Exercise 8.1. Explain why it may be a better or a worse approach than the original.

EX 8.3 Draw a UML class diagram showing an inheritance hierarchy containing classes that represent different types of cars, organized first by manufacturer. Show some appropriate variables and method names for at least two of these classes.

EX 8.4 Show an alternative diagram for the hierarchy in Exercise 8.3 in which the cars are organized first by type (sports car, sedan, SUV, and so on). Show some appropriate variables and method names for at least two of these classes. Compare and contrast the two approaches.

EX 8.5 Draw a UML class diagram showing an inheritance hierarchy containing classes that represent different types of airplanes. Show some appropriate variables and method names for at least two of these classes.

EX 8.6 Draw a UML class diagram showing an inheritance hierarchy containing classes that represent different types of trees (oak, elm, and so on). Show some appropriate variables and method names for at least two of these classes.

EX 8.7 Draw a UML class diagram showing an inheritance hierarchy containing classes that represent different types of payment transactions at a store (cash, credit card, and so on). Show some appropriate variables and method names for at least two of these classes.

EX 8.8 Experiment with a simple derivation relationship between two classes. Put `println` statements in constructors of both the parent class and the child class. Do not explicitly call the constructor of the parent in the child. What happens? Why? Change the child's constructor to explicitly call the constructor of the parent. Now what happens?

Programming Projects

PP 8.1 Design and implement a class called `MonetaryCoin` that is derived from the `Coin` class presented in Chapter 5. Store a value in the monetary coin that represents its value, and add a method that returns its value. Create a driver class to instantiate and compute the sum of several `MonetaryCoin` objects. Demonstrate that a monetary coin inherits its parent's ability to be flipped.

PP 8.2 Design and implement a set of classes that define the employees of a hospital: doctor, nurse, administrator, surgeon, receptionist, janitor, and so on. Include methods in each class that are named according to the services provided by that person and that print an appropriate message. Create a driver class to instantiate and exercise several of the classes.

PP 8.3 Design and implement a set of classes that define various types of reading material: books, novels, magazines, technical journals, textbooks, and so on. Include data values that describe various attributes of the material, such as the number of pages and the names of the primary characters. Include methods that are named appropriately for each class and that print an appropriate message. Create a driver class to instantiate and exercise several of the classes.

PP 8.4 Design and implement a set of classes that keep track of various sports statistics. Have each low-level class represent a specific sport. Tailor the services of the classes to the sport in question, and move common attributes to the higher-level classes as appropriate. Create a driver class to instantiate and exercise several of the classes.

PP 8.5 Design and implement a set of classes that keep track of demographic information about a set of people, such as age, nationality, occupation, income, and so on. Design each class to focus on a particular aspect of data collection. Create a driver class to instantiate and exercise several of the classes.

PP 8.6 Design and implement a set of classes that define a series of three-dimensional geometric shapes. For each shape, store fundamental data about its size, and provide methods to access and modify the data. In addition, provide appropriate methods to compute each shape's circumference, area, and volume. In your design, consider how shapes are related and thus where inheritance can be implemented. Create a driver class to instantiate several shapes of differing types and exercise the behavior you provided.

PP 8.7 Design and implement a set of classes that define various types of electronics equipment (computers, cell phones, pagers, digital cameras, and so on). Include data values that describe various attributes of the electronics, such as the weight, cost, power usage, and name of the manufacturer. Include methods that are

named appropriately for each class and that print an appropriate message. Create a driver class to instantiate and exercise several of the classes.

PP 8.8 Design and implement a set of classes that define various courses in your curriculum. Include information about each course, such as its title, number, and description and the department that teaches the course. Consider the categories of classes that make up your curriculum when designing your inheritance structure. Create a driver class to instantiate and exercise several of the classes.

Answers to Self-Review Questions

SRA 8.1 A child class is derived from a parent class using inheritance. The methods and variables of the parent class automatically become a part of the child class, subject to the rules of the visibility modifiers used to declare them.

SRA 8.2 Because a new class can be derived from an existing class, the characteristics of the parent class can be reused without the error-prone process of copying and modifying code.

SRA 8.3 Each inheritance derivation should represent an is-a relationship: the child *is-a* more specific version of the parent. If this relationship does not hold, then inheritance is being used improperly.

SRA 8.4 The `protected` modifier establishes a visibility level (such as `public` and `private`) that takes inheritance into account. A variable or method declared with protected visibility can be referenced by name in the derived class, while retaining some level of encapsulation. Protected visibility allows access from any class in the same package.

SRA 8.5 The `super` reference can be used to call the parent's constructor, which cannot be invoked directly by name. It can also be used to invoke the parent's version of an overridden method.

SRA 8.6 With single inheritance, a class is derived from only one parent, whereas with multiple inheritance, a class can be derived from multiple parents, inheriting the properties of each. The problem with multiple inheritance is that collisions must be resolved in the cases when two or more parents contribute an attribute or method with the same name. Java supports only single inheritance.

SRA 8.7 A child class may prefer its own definition of a method in favor of the definition provided for it by its parent. In this case, the child overrides (redefines) the parent's definition with its own.

SRA 8.8 All classes in Java are derived, directly or indirectly, from the `Object` class. Therefore, all public methods of the `Object` class, such as `equals` and `toString`, are available to every object.

SRA 8.9 An abstract class is a representation of a general concept. Common characteristics and method signatures can be defined in an abstract class so that they are inherited by child classes derived from it.

SRA 8.10 A class member is not inherited if it has private visibility, meaning that it cannot be referenced by name in the child class. However, such members do exist for the child and can be referenced indirectly.

SRA 8.11 The `final` modifier can be applied to a particular method, which keeps that method from being overridden in a child class. It can also be applied to an entire class, which keeps that class from being extended at all.