CHAPTER 10

Introduction to Inheritance

In this chapter, you will:

- Learn about the concept of inheritance
- Extend classes
- Override superclass methods
- Call constructors during inheritance
- Access superclass methods
- Employ information hiding
- Learn which methods you cannot override

Learning About the Concept of Inheritance

In Java and all object-oriented languages, **inheritance** is a mechanism that enables one class to acquire all the behaviors and attributes of another class and then to expand on those features. Inheritance is the principle that allows you to apply your knowledge of a general category to more specific objects. A class can inherit all the attributes of an existing class, meaning that you can create a new class simply by indicating the ways in which it differs from a class that has already been developed and tested.

You are familiar with the concept of inheritance from all sorts of nonprogramming situations. When you use the term *inheritance*, you might think of genetic inheritance. You know from biology that your blood type and eye color are the product of inherited genes; you can say that many facts about you—your attributes, or "data fields"—are inherited. Similarly, you often can credit your behavior to inheritance. For example, your attitude toward saving money might be the same as your grandmother's, and the odd way that you pull on your ear when you are tired might match what your Uncle Steve does—thus, your methods are inherited, too.

You also might choose plants and animals based on inheritance. You plant impatiens next to your house because of your shady street location; you adopt a Doberman Pinscher because you need a watchdog. Every individual plant and pet has slightly different characteristics, but within a species, you can count on many consistent inherited attributes and behaviors. Similarly, the classes you create in object-oriented programming languages can inherit data and methods from existing classes. When you create a class by making it inherit from another class, you are provided with data fields and methods automatically.

Diagramming Inheritance Using the UML

Beginning with the first chapter of this book, you have been creating classes and instantiating objects that are members of those classes. Programmers and analysts sometimes use a graphical language to describe classes and object-oriented processes; this **Unified Modeling Language (UML)** consists of many types of diagrams. UML diagrams can help illustrate inheritance.

For example, consider the simple `Employee` class shown in Figure 10-1. The class contains two data fields, `empNum` and `empSal`, and four methods: a get and set method for each field. Figure 10-2 shows a UML class diagram for the `Employee` class. A **class diagram** is a visual tool that provides you with an overview of a class. It consists of a rectangle divided into three sections—the top section contains the name of the class, the middle section contains the names and data types of the attributes, and the bottom section contains the methods. Only the method return type, name, and arguments are provided in the diagram—the instructions that make up the method body are omitted.

```
public class Employee
{
   private int empNum;
   private double empSal;
   public int getEmpNum()
   {
      return empNum;
   }
   public double getEmpSal()
   {
      return empSal;
   }
   public void setEmpNum(int num)
   {
      empNum = num;
   }
   public void setEmpSal(double sal)
   {
      empSal = sal;
   }
}
```

Figure 10-1 The `Employee` class

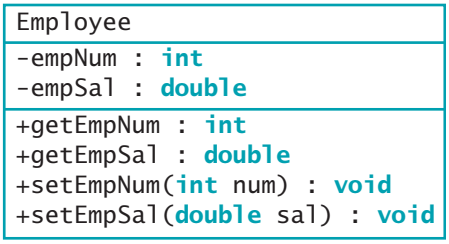

Figure 10-2 The `Employee` class diagram

By convention, a class diagram contains the data type following each attribute or method, as shown in Figure 10-2. A minus sign (–) is inserted in front of each private field or method, and a plus sign (+) is inserted in front of each public field or method.

Commonly, UML diagram creators refrain from using Java terminology such as `int` in a class diagram. Instead, they might use a more general term, such as integer. The `Employee` class is designed in natural language (English) and might be implemented in any programming language, and languages other than Java might use a different keyword to designate integer variables. Because you are studying Java, this book uses the Java keywords in diagrams. For more information on UML, you can go to the Object Management Group's Web site at *www.omg.org*. OMG is an international, nonprofit computer industry consortium.

After you create the `Employee` class, you can create specific `Employee` objects, such as the following:

```
Employee receptionist = new Employee();
Employee deliveryPerson = new Employee();
```

These `Employee` objects can eventually possess different numbers and salaries, but because they are `Employee` objects, you know that each `Employee` has *some* number and salary.

Suppose that you hire a new type of Employee named serviceRep, and that a serviceRep object requires not only an employee number and a salary but also a data field to indicate territory served. You can create a class with a name such as EmployeeWithTerritory and provide the class three fields (empNum, empSal, and empTerritory) and six methods (get and set methods for each of the three fields). However, when you do this, you are duplicating much of the work that you have already done for the Employee class. The wise, efficient alternative is to create the class EmployeeWithTerritory so it inherits all the attributes and methods of Employee. Then, you can add just the one field and two methods that are new within EmployeeWithTerritory objects. Figure 10-3 shows a class diagram of this relationship. In a UML diagram, an inheritance relationship is indicated with an arrow that points from the descendant class to the original class.

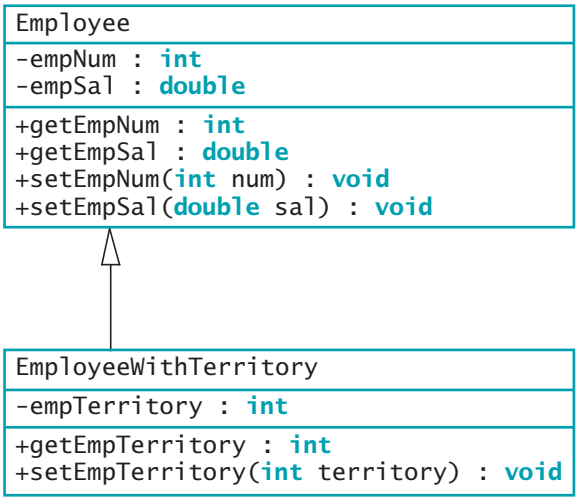

Figure 10-3 Class diagram showing the relationship between Employee and EmployeeWithTerritory

When you use inheritance to create the EmployeeWithTerritory class, you:

- Save time because the Employee fields and methods already exist
- Reduce errors because the Employee methods already have been used and tested
- Reduce the amount of new learning required to use the new class, because you have used the Employee methods on simpler objects and already understand how they work

The ability to use inheritance in Java makes programs easier to write, less error prone, and more quickly understood. Besides creating EmployeeWithTerritory, you also can create several other specific Employee classes (perhaps EmployeeEarningCommission, including a commission rate, or DismissedEmployee, including a reason for dismissal). By using inheritance, you can develop each new class correctly and more quickly. The concept of

inheritance is useful because it makes a class's code more easily reusable. Each defined data field and each method already written and tested in the original class becomes part of the new class that inherits it.

In Chapter 4, you learned about the `GregorianCalendar` class. It descends from a more general class named `Calendar`.

Inheritance Terminology

A class that is used as a basis for inheritance, such as `Employee`, is a **base class**. When you create a class that inherits from a base class (such as `EmployeeWithTerritory`), it is a **derived class**. When considering two classes that inherit from each other, you can tell which is the base class and which is the derived class by using the two classes in a sentence with the phrase "is a(n)." A derived class always "is a" case or example of the more general base class. For example, a `Tree` class can be a base class to an `Evergreen` class. An `Evergreen` "is a" `Tree`, so `Tree` is the base class; however, it is not true for all `Trees` that "a `Tree` is an `Evergreen`." Similarly, an `EmployeeWithTerritory` "is an" `Employee`—but not the other way around—so `Employee` is the base class.

Because a derived class object "is an" instance of the base class, too, you can assign a derived class object's reference to a base class reference. Similarly, if a method accepts a base class object reference, it also will accept references to its derived classes. The next chapter describes these concepts in greater detail.

Do not confuse "is a" situations with "has a" situations. For example, you might create a `Business` class that contains an array of `Department` objects; in turn, each `Department` object might contain an array of `Employee` objects. You would not say "A department is a business" but that "a business has departments." Therefore, this relationship is not inheritance; it is **composition**—the relationship in which a class contains one or more members of another class, when those members would not continue to exist without the object that contains them. (For example, if a `Business` closes, its `Departments` do, too.) Similarly, you would not say "an employee is a department" but that "a department has employees." This relationship is not inheritance either; it is a specific type of composition known as **aggregation**—the relationship in which a class contains one or more members of another class, when those members would continue to exist without the object that contains them. (For example, if a business or department closed, the employees would continue to exist.)

You can use the terms **superclass** and **subclass** as synonyms for base class and derived class, respectively. Thus, `Evergreen` can be called a subclass of the `Tree` superclass. You can also use the terms **parent class** and **child class**. An `EmployeeWithTerritory` is a child to the `Employee` parent. Use the pair of terms with which you are most comfortable; all of these terms are used interchangeably throughout this book.

As an alternative way to discover which of two classes is the base class or subclass, you can try saying the two class names together. When people say their names together, they state the more specific name before the all-encompassing family name, as in "Ginny Kroening." Similarly, with classes, the order that "makes more sense" is the child-parent order. "Evergreen Tree" makes more sense than "Tree Evergreen," so `Evergreen` is the child class.

Finally, you usually can distinguish superclasses from their subclasses by size. Although it is not required, in general a subclass is larger than a superclass because it usually has additional fields and methods. A subclass description might look small, but any subclass contains all the fields and methods of its superclass, as well as the new, more specific fields and methods you add to that subclass.

Watch the video *Inheritance*.

TWO TRUTHS & A LIE

Learning About the Concept of Inheritance

1. When you use inheritance in Java, you can create a new class that contains all the data and methods of an existing class.
2. When you use inheritance, you save time and reduce errors.
3. A class that is used as a basis for inheritance is called a subclass.

The false statement is #3. A class that is used as a basis for inheritance is called a superclass, base class, or parent class. A subclass is a class that inherits from a superclass.

Extending Classes

You use the keyword **`extends`** to achieve inheritance in Java. For example, the following class header creates a superclass-subclass relationship between `Employee` and `EmployeeWithTerritory`:

```
public class EmployeeWithTerritory extends Employee
```

Each `EmployeeWithTerritory` automatically receives the data fields and methods of the superclass `Employee`; you then add new fields and methods to the newly created subclass. Figure 10-4 shows an `EmployeeWithTerritory` class.

```
public class EmployeeWithTerritory extends Employee
{
   private int empTerritory;
   public int getEmpTerritory()
   {
      return empTerritory;
   }
   public void setEmpTerritory(int num)
   {
      empTerritory = num;
   }
}
```

Figure 10-4 The `EmployeeWithTerritory` class

You can write a statement that instantiates a derived class object, such as the following:

```
EmployeeWithTerritory northernRep = new EmployeeWithTerritory();
```

Then you can use any of the next statements to get field values for the `northernRep` object:

```
northernRep.getEmpNum();
northernRep.getEmpSal();
northernRep.getEmpTerritory();
```

The `northernRep` object has access to all three get methods—two methods that it inherits from `Employee` and one method that belongs to `EmployeeWithTerritory`.

Similarly, after the `northernRep` object is declared, any of the following statements are legal:

```
northernRep.setEmpNum(915);
northernRep.setEmpSal(210.00);
northernRep.setEmpTerritory(5);
```

The `northernRep` object has access to all the parent `Employee` class set methods, as well as its own class's new set method.

Inheritance is a one-way proposition; a child inherits from a parent, not the other way around. When you instantiate an `Employee` object, it does not have access to the `EmployeeWithTerritory` methods. It makes sense that a parent class object does not have access to its child's data and methods. When you create the parent class, you do not know how many future subclasses it might have or what their data or methods might look like.

In addition, subclasses are more specific than the superclass they extend. An `Orthodontist` class and `Periodontist` class are children of the `Dentist` parent class. You do not expect all members of the general parent class `Dentist` to have the `Orthodontist`'s `applyBraces()` method or the `Periodontist`'s `deepClean()` method. However, `Orthodontist` objects and `Periodontist` objects have access to the more general `Dentist` methods `conductExam()` and `billPatients()`.

You can use the **instanceof operator** to determine whether an object is a member or descendant of a class. For example, if `northernRep` is an `EmployeeWithTerritory` object, then the value of each of the following expressions is `true`:

```
northernRep instanceof EmployeeWithTerritory
northernRep instanceof Employee
```

If `aClerk` is an `Employee` object, then the following is `true`:

```
aClerk instanceof Employee
```

However, the following is `false`:

```
aClerk instanceof EmployeeWithTerritory
```

Programmers say that `instanceof` yields `true` if the operand on the left can be **upcast** to the operand on the right.

TWO TRUTHS & A LIE

Extending Classes

1. You use the keyword `inherits` to achieve inheritance in Java.
2. A derived class has access to all its parents' nonprivate methods.
3. Subclasses are more specific than the superclass they extend.

The false statement is #1. You use the keyword `extends` to achieve inheritance in Java.

You Do It

Demonstrating Inheritance

In this section, you create a working example of inheritance. To see the effects of inheritance, you create this example in four stages:

- First, you create a `Party` class that holds just one data field and three methods.
- After you create the general `Party` class, you write an application to demonstrate its use.
- Then, you create a more specific `DinnerParty` subclass that inherits the fields and methods of the `Party` class.
- Finally, you modify the demonstration application to add an example using the `DinnerParty` class.

(continues)

(continued)

Creating a Superclass and an Application to Use It

1. Open a new file in your text editor, and enter the following first few lines for a simple `Party` class. The class hosts one integer data field—the number of guests expected at the party:

```
public class Party
{
   private int guests;
```

2. Add the following methods that get and set the number of `guests`:

```
public int getGuests()
{
   return guests;
}
public void setGuests(int numGuests)
{
   guests = numGuests;
}
```

3. Add a method that displays a party invitation:

```
public void displayInvitation()
{
   System.out.println("Please come to my party!");
}
```

4. Add the closing curly brace for the class, and then save the file as **Party.java**. Compile the class; if necessary, correct any errors and compile again.

Writing an Application That Uses the `Party` Class

Now that you have created a class, you can use it in an application. A very simple application creates a `Party` object, prompts the user for the number of guests at the party, sets the data field, and displays the results.

1. Open a new file in your text editor.
2. Write a `UseParty` application that has one method—a `main()` method. Enter the beginning of the class, including the start of the `main()` method, which declares a variable for guests, a `Party` object, and a `Scanner` object to use for input:

(continues)

(continued)

```
import java.util.*;
public class UseParty
{
   public static void main(String[] args)
   {
      int guests;
      Party aParty = new Party();
      Scanner keyboard = new Scanner(System.in);
```

3. Continue the `main()` method by prompting the user for a number of guests and accepting the value from the keyboard. Set the number of guests in the `Party` object, and then display the value:

```
System.out.print("Enter number of guests for the party >> ");
guests = keyboard.nextInt();
aParty.setGuests(guests);
System.out.println("The party has " + aParty.getGuests() + " guests");
```

4. Add a statement to display the party invitation, and then add the closing curly braces for the `main()` method and for the class:

```
      aParty.displayInvitation();
   }
}
```

5. Save the file as **UseParty.java**, then compile and run the application. Figure 10-5 shows a typical execution.

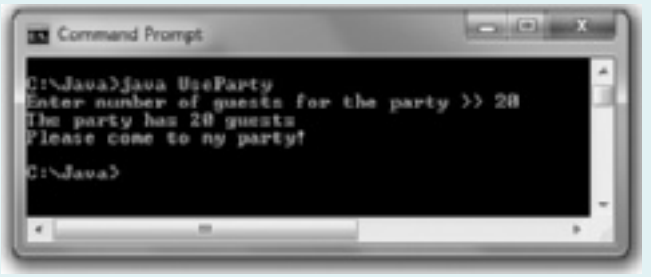

Figure 10-5 Execution of the `UseParty` application

Creating a Subclass from the `Party` Class

Next, you create a class named `DinnerParty`. A `DinnerParty` "is a" type of `Party` at which dinner is served, so `DinnerParty` is a child class of `Party`.

(continues)

(continued)

1. Open a new file in your text editor, and type the first few lines for the `DinnerParty` class:

```
public class DinnerParty extends Party
{
```

2. A `DinnerParty` contains a number of guests, but you do not have to define the variable here. The variable is already defined in `Party`, which is the superclass of this class. You need to add only any variables that are particular to a `DinnerParty`. Enter the following code to add an integer code for the dinner menu choice:

```
private int dinnerChoice;
```

3. The `Party` class already contains methods to get and set the number of guests, so `DinnerParty` needs methods only to get and set the `dinnerChoice` variable as follows:

```
public int getDinnerChoice()
{
   return dinnerChoice;
}
public void setDinnerChoice(int choice)
{
   dinnerChoice = choice;
}
```

4. Save the file as **DinnerParty.java**, and then compile it.

Creating an Application That Uses the `DinnerParty` Class

Now you can modify the `UseParty` application so that it creates a `DinnerParty` as well as a plain `Party`.

1. Open the **UseParty.java** file in your text editor. Change the class name to **`UseDinnerParty`**. Immediately save the file as **UseDinnerParty.java**.
2. Include a new variable that holds the dinner choice for a `DinnerParty`:

```
int choice;
```

3. After the statement that constructs the `Party` object, type the following `DinnerParty` object declaration:

```
DinnerParty aDinnerParty = new DinnerParty();
```

(continues)

(continued)

4. At the end of the `main()` method, after the `Party` object data and invitation are displayed, add a prompt for the number of guests for the `DinnerParty`. Accept the value the user enters and assign it to the object. Even though the `DinnerParty` class does not contain a `setGuests()` method, its parent class does, so `aDinnerParty` can use the method.

```
System.out.print("Enter number of guests for the dinner party >> ");
guests = keyboard.nextInt();
aDinnerParty.setGuests(guests);
```

5. Next, prompt the user for a dinner choice. To keep this example simple, the program provides only two choices and does not provide range checking. Accept a response from the user, assign it to the object, and then display all the data for the `DinnerParty`. Even though the `DinnerParty` class does not contain a `getGuests()` method, its parent class does, so `aDinnerParty` can use the method. The `DinnerParty` class uses its own `setDinnerChoice()` and `getDinnerChoice()` methods.

```
System.out.print
   ("Enter the menu option -- 1 for chicken or 2 for beef >> ");
choice = keyboard.nextInt();
aDinnerParty.setDinnerChoice(choice);
System.out.println("The dinner party has " +
   aDinnerParty.getGuests() + " guests");
System.out.println("Menu option " +
   aDinnerParty.getDinnerChoice () + " will be served");
```

6. Add a statement to call the `displayInvitation()` method with the `DinnerParty` object. Even though the `DinnerParty` class does not contain a `displayInvitation()` method, its parent class does, so `aDinnerParty` can use the method.

```
aDinnerParty.displayInvitation();
```

7. Save the file, compile it, and run it using values of your choice. Figure 10-6 shows a typical execution. The `DinnerParty` object successfully uses the data field and methods of its superclass, as well as its own data field and methods.

(continues)

(continued)

Figure 10-6 Execution of the `UseDinnerParty` application

Overriding Superclass Methods

When you create a subclass by extending an existing class, the new subclass contains data and methods that were defined in the original superclass. In other words, any child class object has all the attributes of its parent. Sometimes, however, the superclass data fields and methods are not entirely appropriate for the subclass objects; in these cases, you want to override the parent class members.

When you use the English language, you often use the same method name to indicate diverse meanings. For example, if you think of `MusicalInstrument` as a class, you can think of `play()` as a method of that class. If you think of various subclasses such as `Guitar` and `Drum`, you know that you carry out the `play()` method quite differently for each subclass. Using the same method name to indicate different implementations is called **polymorphism**, a term meaning "many forms"—many different forms of action take place, even though you use the same word to describe the action. In other words, many forms of the same word exist, depending on the object associated with the word.

You first learned the term *polymorphism* in Chapter 1. Polymorphism is one of the basic principles of object-oriented programming. If a programming language does not support polymorphism, the language is not considered object oriented.

For example, suppose that you create an `Employee` superclass containing data fields such as `firstName`, `lastName`, `socialSecurityNumber`, `dateOfHire`, `rateOfPay`, and so on, and the methods contained in the `Employee` class include the usual collection of get and set

methods. If your usual time period for payment to each `Employee` object is weekly, your `displayRateOfPay()` method might include a statement such as:

```
System.out.println("Pay is " + rateOfPay + " per week ");
```


Imagine your company has a few `Employees` who are not paid weekly. Maybe some are paid by the hour, and others are `Employees` whose work is contracted on a job-to-job basis. Because each `Employee` type requires different paycheck-calculating procedures, you might want to create subclasses of `Employee`, such as `HourlyEmployee` and `ContractEmployee`.

When you call the `displayRateOfPay()` method for an `HourlyEmployee` object, you want the display to include the phrase "per hour", as in "Pay is $8.75 per hour." When you call the `displayRateOfPay()` method for a `ContractEmployee`, you want to include "per contract", as in "Pay is $2000 per contract." Each class—the `Employee` superclass and the two subclasses—requires its own `displayRateOfPay()` method. Fortunately, if you create separate `displayRateOfPay()` methods for each class, the objects of each class use the appropriate method for that class. When you create a method in a child class that has the same name and parameter list as a method in its parent class, you **override the method** in the parent class. When you use the method name with a child object, the child's version of the method is used.

It is important to note that each subclass method overrides any method in the parent class that has both the same name and parameter list. If the parent class method has the same name but a different parameter list, the subclass method does not over*ride* the parent class version; instead, the subclass method over*loads* the parent class method, and any subclass object has access to both versions. You learned about overloading methods in Chapter 4. You first saw the term *override* in Chapter 4, when you learned that a variable declared within a block overrides another variable with the same name declared outside the block.

If you could not override superclass methods, you could always create a unique name for each subclass method, such as `displayRateOfPayForHourly()`, but the classes you create are easier to write and understand if you use one reasonable name for methods that do essentially the same thing. Because you are attempting to display the rate of pay for each object, `displayRateOfPay()` is a clear and appropriate method name for all the object types.

A child class object can use an overridden parent's method by using the keyword `super`. You will learn about this word later in this chapter.

Object-oriented programmers use the term *polymorphism* when discussing any operation that has multiple meanings. For example, the plus sign (+) is polymorphic because it operates differently depending on its operands. You can use the plus sign to add integers or `doubles`, to concatenate strings, or to indicate a positive value. As another example, methods with the same name but different parameter lists are polymorphic because the method call operates differently depending on its arguments. When Java developers refer to polymorphism, they most often mean **subtype polymorphism**—the ability of one method name to work appropriately for different subclass objects of the same parent class.

Watch the video *Handling Methods and Inheritance*.

TWO TRUTHS & A LIE

Overriding Superclass Methods

1. Any child class object has all the attributes of its parent, but all of those attributes might not be directly accessible.
2. You override a parent class method by creating a child class method with the same identifier but a different parameter list or return type.
3. When a child class method overrides a parent class method, and you use the method name with a child class object, the child class method version executes.

The false statement is #2. You override a parent class method by creating a child class method with the same identifier and parameter list. The return type is not a factor in overloading.

You Do It

Overriding a Superclass Method

In the previous "You Do It" section, you created `Party` and `DinnerParty` classes. The `DinnerParty` class extends `Party` and so can use its `displayInvitation()` method. Suppose that you want a `DinnerParty` object to use a specialized invitation. In this section, you override the parent class method so that the same method name acts uniquely for the child class object.

1. Open the **DinnerParty.java** class in your text editor. Change the class name to **`DinnerParty2`**, and save the file as **DinnerParty2.java**.
2. Create a `displayInvitation()` method that overrides the parent class method with the same name as follows:

```
public void displayInvitation()
{
   System.out.println("Please come to my dinner party!");
}
```

3. Save the class and compile it.

(continues)

(continued)

4. Open the **UseDinnerParty.java** file. Change the class name to `UseDinnerParty2`, and immediately save the file as **UseDinnerParty2.java**.
5. Change the declaration of the `aDinnerParty` object so that it uses the `DinnerParty2` class as a data type and `DinnerParty2` as the constructor name.
6. Save the class, compile it, and execute it. Figure 10-7 shows a typical execution. Each type of object uses its own version of the `displayInvitation()` method.

Figure 10-7 Typical execution of the `UseDinnerParty2` program

Calling Constructors During Inheritance

When you create any object, as in the following statement, you are calling a class constructor method that has the same name as the class itself:

```
SomeClass anObject = new SomeClass();
```

When you instantiate an object that is a member of a subclass, you are actually calling at least two constructors: the constructor for the base class and the constructor for the extended, derived class. When you create any subclass object, the superclass constructor must execute first, and *then* the subclass constructor executes.

In the chapter *Advanced Inheritance Concepts*, you will learn that every Java object automatically is a child of a class named `Object`. So, when you instantiate any object, you call its constructor and `Object`'s constructor, and when you create parent and child classes of your own, the child classes use three constructors.

When a superclass contains a default constructor and you instantiate a subclass object, the execution of the superclass constructor is transparent—that is, nothing calls attention to the fact that the superclass constructor is executing. However, you should realize that when you create an object such as the following (where `HourlyEmployee` is a subclass of `Employee`), *both* the `Employee()` and `HourlyEmployee()` constructors execute:

```
HourlyEmployee clerk = new HourlyEmployee();
```

For example, Figure 10-8 shows three classes. The class named `ASuperClass` has a constructor that displays a message. The class named `ASubClass` descends from `ASuperClass`, and its constructor displays a different message. The `DemoConstructors` class contains just one statement that instantiates one object of type `ASubClass`.

```
public class ASuperClass
{
   public ASuperClass()
   {
      System.out.println("In superclass constructor");
   }
}
public class ASubClass extends ASuperClass
{
   public ASubClass()
   {
      System.out.println("In subclass constructor");
   }
}
public class DemoConstructors
{
   public static void main(String[] args)
   {
      ASubClass child = new ASubClass();
   }
}
```

Figure 10-8 Three classes that demonstrate constructor calling when a subclass object is instantiated

Figure 10-9 shows the output when `DemoConstructors` executes. You can see that when `DemoConstructors` instantiates the `ASubClass` object, the parent class constructor executes first, displaying its message, and then the child class constructor executes. Even though only one object is created, two constructors execute.

Figure 10-9 Output of the `DemoConstructors` application

Of course, most constructors perform many more tasks than displaying a message to inform you that they exist. When constructors initialize variables, you usually want the superclass constructor to take care of initializing the data fields that originate in the superclass. Usually, the subclass constructor only needs to initialize the data fields that are specific to the subclass.

Using Superclass Constructors That Require Arguments

When you create a class and do not provide a constructor, Java automatically supplies you with a default constructor—one that never requires arguments. When you write your own constructor, you replace the automatically supplied version. Depending on your needs, a constructor you create for a class might be a default constructor or might require arguments. When you use a class as a superclass and the class has only constructors that require arguments, you must be certain that any subclasses provide the superclass constructor with the arguments it needs.

Don't forget that a class can have many constructors. As soon as you create at least one constructor for a class, you can no longer use the automatically supplied version.

When a superclass has a default constructor, you can create a subclass with or without its own constructor. This is true whether the default superclass constructor is the automatically supplied one or one you have written. However, when a superclass contains only constructors that require arguments, you must include at least one constructor for each subclass you create. Your subclass constructors can contain any number of statements, but if all superclass constructors require arguments, then the first statement within each subclass constructor must call one of the superclass constructors. When a superclass requires constructor arguments upon object instantiation, even if you have no other reason to create a subclass constructor, you must write the subclass constructor so it can call its superclass's constructor.

If a superclass has multiple constructors but one is a default constructor, you do not have to create a subclass constructor unless you want to. If the subclass contains no constructor, all subclass objects use the superclass default constructor when they are instantiated.

The format of the statement that calls a superclass constructor from the subclass constructor is:

```
super(list of arguments);
```

The keyword **super** always refers to the superclass of the class in which you use it.

If a superclass contains only constructors that require arguments, you must create a subclass constructor, but the subclass constructor does not necessarily have to have parameters of its own. For example, suppose that you create an `Employee` class with a constructor that requires three arguments—a character, a `double`, and an integer—and you create an `HourlyEmployee` class that is a subclass of `Employee`. The following code shows a valid constructor for `HourlyEmployee`:

```
public HourlyEmployee()
{
    super('P', 12.35, 40);
    // Other statements can go here
}
```

This version of the `HourlyEmployee` constructor requires no arguments, but it passes three constant arguments to its superclass constructor. A different, overloaded version of the `HourlyEmployee` constructor can require arguments. It could then pass the appropriate arguments to the superclass constructor. For example:

```
public HourlyEmployee(char dept, double rate, int hours)
{
    super(dept, rate, hours);
    // Other statements can go here
}
```

Except for any comments, the `super()` statement must be the first statement in any subclass constructor that uses it. Not even data field definitions can precede it. Although it seems that you should be able to use the superclass constructor name to call the superclass constructor—for example, `Employee()`—Java does not allow this. You must use the keyword `super`.

In Chapter 4, you learned that you can call one constructor from another using `this()`. In this chapter, you learned that you can call a base class constructor from a derived class using `super()`. However, you cannot use both `this()` and `super()` in the same constructor because each is required to be the first statement in any constructor in which it appears.

Watch the video *Constructors and Inheritance*.

TWO TRUTHS & A LIE

Calling Constructors During Inheritance

1. When you create any subclass object, the subclass constructor executes first, and then the superclass constructor executes.
2. When constructors initialize variables, you usually want the superclass constructor to initialize the data fields that originate in the superclass and the subclass constructor to initialize the data fields that are specific to the subclass.
3. When a superclass contains only nondefault constructors, you must include at least one constructor for each subclass you create.

The false statement is #1. When you create any subclass object, the superclass constructor must execute first, and *then* the subclass constructor executes.

You Do It

Understanding the Role of Constructors in Inheritance

Next, you add a constructor to the `Party` class. When you instantiate a subclass object, the superclass constructor executes before the subclass constructor executes.

1. Open the **Party.java** file in your text editor, and save it as **PartyWithConstructor.java**. Change the class name to `PartyWithConstructor`.
2. Following the statement that declares the `guests` data field, type a constructor that does nothing other than display a message indicating it is working:

```
public PartyWithConstructor()
{
   System.out.println("Creating a Party");
}
```

3. Save the file and compile it.

(continues)

(continued)

4. In your text editor, open the **DinnerParty2.java** file, and change the class name to `DinnerPartyWithConstructor`. Change the class in the `extends` clause to `PartyWithConstructor`. Save the file as **DinnerPartyWithConstructor.java**, and compile it.

5. In your text editor, open a new file so you can write an application to demonstrate the use of the base class constructor with an extended class object. This application creates only one child class object:

```
public class UseDinnerPartyWithConstructor
{
   public static void main(String[] args)
   {
       DinnerPartyWithConstructor aDinnerParty =
          new DinnerPartyWithConstructor();
   }
}
```

6. Save the application as **UseDinnerPartyWithConstructor.java**, then compile and run it. The output is shown in Figure 10-10. Even though the application creates only one subclass object (and no superclass objects) and the subclass contains no constructor of its own, the superclass constructor executes.

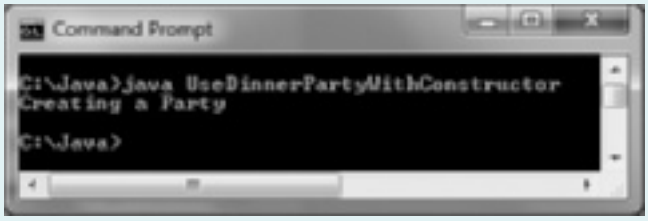

Figure 10-10 Output of the `UseDinnerPartyWithConstructor` application

Inheritance When the Superclass Requires Constructor Arguments

Next, you modify the `PartyWithConstructor` class so that its constructor requires an argument. Then, you observe that a subclass without a constructor cannot compile.

1. Open the **PartyWithConstructor.java** file in your text editor, and then change the class name to **PartyWithConstructor2**.

(continues)

(continued)

2. Replace the existing constructor with a new version using the new class name. This constructor requires an argument, which it uses to set the number of guests who will attend a party:

```
public PartyWithConstructor2(int numGuests)
{
   guests = numGuests;
}
```

3. Save the file as **PartyWithConstructor2.java**, and then compile it.
4. Open the **DinnerPartyWithConstructor.java** file in your text editor.
5. Change the class header as follows so that the name of the class is `DinnerPartyWithConstructor2` and it inherits from `PartyWithConstructor2`:

```
public class DinnerPartyWithConstructor2 extends
   PartyWithConstructor2
```

6. Save the file as **DinnerPartyWithConstructor2.java**, and then compile it. An error message appears, as shown in Figure 10-11. When you attempt to compile the subclass, no parameterless constructor can be found in the superclass, so the compile fails.

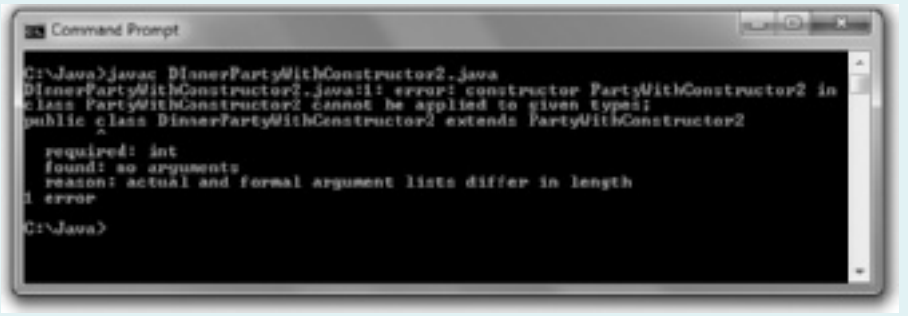

Figure 10-11 Error message generated when compiling the `DinnerPartyWithConstructor2` class

7. To correct the error, open the **DinnerPartyWithConstructor2.java** file in your text editor. Following the `dinnerChoice` field declaration, insert a constructor for the class as follows:

```
public DinnerPartyWithConstructor2(int numGuests)
{
   super(numGuests);
}
```

(continues)

(continued)

8. Save the file and compile it. This time, the compile is successful because the subclass calls its parent's constructor, passing along an integer value. Note that the `DinnerPartyWithConstructor2` subclass constructor is not required to receive an integer argument, although in this example it does. For example, it would be acceptable to create a subclass constructor that required no arguments but passed a constant (for example, 0) to its parent. Similarly, the subclass constructor could require several arguments and pass one of them to its parent. The requirement is not that the subclass constructor must have the same number or types of parameters as its parent; the only requirement is that the subclass constructor calls `super()` and passes to the parent what it needs to execute.

Accessing Superclass Methods

Earlier in this chapter, you learned that a subclass can contain a method with the same name and arguments (the same signature) as a method in its parent class. When this happens, using the subclass method overrides the superclass method. However, instead of overriding the superclass method, you might want to use it within a subclass. If so, you can use the keyword `super` to access the parent class method.

For example, examine the `Customer` class in Figure 10-12 and the `PreferredCustomer` class in Figure 10-13. A `Customer` has an `idNumber` and `balanceOwed`. In addition to these fields, a `PreferredCustomer` receives a `discountRate`. In the `PreferredCustomer display()` method, you want to display all three fields—`idNumber`, `balanceOwed`, and `discountRate`. Because two-thirds of the code to accomplish the display has already been written for the `Customer` class, it is convenient to have the `PreferredCustomer display()` method use its parent's version of the `display()` method before displaying its own discount rate. Figure 10-14 shows a brief application that displays one object of each class, and Figure 10-15 shows the output.

```
public class Customer
{
   private int idNumber;
   private double balanceOwed;
   public Customer(int id, double bal)
   {
      idNumber = id;
      balanceOwed = bal;
   }
   public void display()
   {
      System.out.println("Customer #" + idNumber +
        " Balance $" + balanceOwed);
   }
}
```

Figure 10-12 The `Customer` class

```
public class PreferredCustomer extends Customer
{
   double discountRate;
   public PreferredCustomer(int id, double bal, double rate)
   {
      super(id, bal);
      discountRate = rate;
   }
   public void display()
   {
      super.display();
      System.out.println("   Discount rate is " + discountRate);
   }
}
```

Figure 10-13 The `PreferredCustomer` class

```
public class TestCustomers
{
   public static void main(String[] args)
   {
      Customer oneCust = new Customer(124, 123.45);
      PreferredCustomer onePCust = new
         PreferredCustomer(125, 3456.78, 0.15);
      oneCust.display();
      onePCust.display();
   }
}
```

Figure 10-14 The `TestCustomers` application

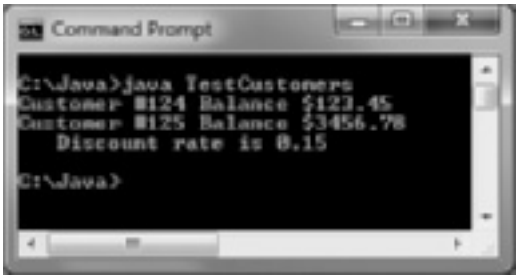

Figure 10-15 Output of the `TestCustomers` application

When you call a superclass constructor from a subclass constructor, the call must be the first statement in the constructor. However, when you call an ordinary superclass method within a subclass method, the call is not required to be the first statement in the method, although it can be, as shown in the `display()` method in Figure 10-13.

Comparing `this` and `super`

In a subclass, the keywords `this` and `super` sometimes refer to the same method, but sometimes they do not.

For example, if a subclass has overridden a superclass method named `someMethod()`, then within the subclass, `super.someMethod()` refers to the superclass version of the method, and both `someMethod()` and `this.someMethod()` refer to the subclass version.

On the other hand, if a subclass has *not* overridden a superclass method named `someMethod()`, the child can use the method name with `super` (because the method is a member of the superclass), with `this` (because the method is a member of the subclass by virtue of inheritance), or alone (again, because the method is a member of the subclass).

TWO TRUTHS & A LIE

Accessing Superclass Methods

1. You can use the keyword `this` from within a derived class method to access an overridden base class method.
2. You can use the keyword `super` from within a derived class method to access an overridden base class method.
3. You can use the keyword `super` from within a derived class method to access a base class method that has not been overridden.

The false statement is #1. You can use the keyword `super` from within a derived class method to access an overridden base class method; if you use the keyword `this`, then you will access the overriding subclass method.

Employing Information Hiding

The Student class shown in Figure 10-16 is an example of a typical Java class. Within the Student class, as with most Java classes, the keyword private precedes each data field, and the keyword public precedes each method. In fact, the four get and set methods are public within the Student class specifically because the data fields are private. Without the public get and set methods, there would be no way to access the private data fields.


```
public class Student
{
   private int idNum;
   private double gpa;
   public int getIdNum()
   {
      return idNum;
   }
   public double getGpa()
   {
      return gpa;
   }
   public void setIdNum(int num)
   {
      idNum = num;
   }
   public void setGpa(double gradePoint)
   {
      gpa = gradePoint;
   }
}
```

Figure 10-16 The Student class

When an application is a client of the Student class (that is, it instantiates a Student object), the client cannot directly alter the data in any private field. For example, suppose that you write a main() method that creates a Student as:

```
Student someStudent = new Student();
```

Then you cannot change the Student's idNum with a statement such as:

```
someStudent.idNum = 812;
```

Don't Do It
You cannot access a private data member of an object.

The idNum of the someStudent object is not accessible in the main() method that uses the Student object because idNum is private. Only methods that are part of the Student class itself are allowed to alter private Student data. To alter a Student's idNum, you must use a public method, as in the following:

```
someStudent.setIdNum(812);
```

The concept of keeping data private is known as **information hiding**. When you employ information hiding, your data can be altered only by the methods you choose and only in ways that you can control. For example, you might want the `setIdNum()` method to check to make certain the `idNum` is within a specific range of values. If a class other than the `Student` class could alter `idNum`, `idNum` could be assigned a value that the `Student` class couldn't control.

You first learned about information hiding and using the `public` and `private` keywords in Chapter 3. You might want to review these concepts.

When a class serves as a superclass to other classes you create, your subclasses inherit all the data and methods of the superclass. The methods in a subclass can use all of the data fields and methods that belong to its parent, with one exception: `private` members of the parent class are not accessible within a child class's methods. If a new class could simply extend your `Student` class and get to its data fields without going through the proper channels, information hiding would not be operating.

If the members of a base class don't have an explicit access specifier, their access is `package` by default. Such base class members cannot be accessed within a child class unless the two classes are in the same package. You will learn about packages in the next chapter.

Sometimes, you want to access parent class data from within a subclass. For example, suppose that you create two child classes—`PartTimeStudent` and `FullTimeStudent`—that extend the `Student` class. If you want the subclass methods to be able to directly access `idNum` and `gpa`, these data fields cannot be `private`. However, if you don't want other, nonchild classes to access these data fields, they cannot be `public`. To solve this problem, you can create the fields using the specifier `protected`. Using the keyword **`protected`** provides you with an intermediate level of security between `public` and `private` access. If you create a `protected` data field or method, it can be used within its own class or in any classes extended from that class, but it cannot be used by "outside" classes. In other words, `protected` members are those that can be used by a class and its descendants.

You seldom are required to make parent class fields `protected`. A child class can access its parent's `private` data fields by using `public` methods defined in the parent class, just as any other class can. You need to make parent class fields `protected` only if you want child classes to be able to access `private` data directly. (For example, perhaps you do not want a parent class to have a `get` method for a field, but you do want a child class to be able to access the field. As another example, perhaps a parent class set method enforces limits on a field's value, but a child class object should not have such limits.) Using the `protected` access specifier for a field can be convenient, and it also improves program performance because a child class can use an inherited field directly instead of "going through" methods to access the data. However, `protected` data members should be used sparingly. Whenever possible, the principle of information hiding should be observed, so even child classes should have to go through `public` methods to "get to" their parent's private data. When

child classes are allowed direct access to a parent's fields, the likelihood of future errors increases. Classes that directly use fields from parent classes are said to be **fragile** because they are prone to errors—that is, they are easy to "break."

TWO TRUTHS & A LIE

Employing Information Hiding

1. Information hiding describes the concept of keeping data private.
2. A subclass inherits all the data and methods of its superclass, except the `private` ones.
3. If a data field is defined as protected, then a method in a child class can use it directly.

The false statement is #2. A subclass inherits all the data and methods of its superclass, but it cannot access the `private` ones directly.

Methods You Cannot Override

Sometimes when you create a class, you might choose not to allow subclasses to override some of the superclass methods. For example, an `Employee` class might contain a method that calculates each `Employee`'s ID number based on specific `Employee` attributes, and you might not want any derived classes to be able to provide their own versions of this method. As another example, perhaps a class contains a statement that displays legal restrictions to using the class. You might decide that no derived class should be able to display a different version of the statement.

The three types of methods that you cannot override in a subclass are:

- `static` methods
- `final` methods
- Methods within `final` classes

A Subclass Cannot Override `static` Methods in Its Superclass

A subclass cannot override methods that are declared `static` in the superclass. In other words, a subclass cannot override a class method—a method you use without instantiating an object. A subclass can *hide* a `static` method in the superclass by declaring a `static` method in the subclass with the same signature as the `static` method in the superclass;

then, you can call the new `static` method from within the subclass or in another class by using a subclass object. However, this `static` method that hides the superclass `static` method cannot access the parent method using the `super` object.

Figure 10-17 shows a `BaseballPlayer` class that contains a single `static` method named `showOrigins()`. Figure 10-18 shows a `ProfessionalBaseballPlayer` class that extends the `BaseballPlayer` class to provide a salary. Within the `ProfessionalBaseballPlayer` class, an attempt is made to create a nonstatic method that overrides the `static showOrigins()` method to display the general Abner Doubleday message about baseball from the parent class as well as the more specific message about professional baseball. However, the compiler returns the error message shown in Figure 10-19—you cannot override a `static` method with a nonstatic method.

```
public class BaseballPlayer
{
   private int jerseyNumber;
   private double battingAvg;
   public static void showOrigins()
   {
      System.out.println("Abner Doubleday is often " +
         "credited with inventing baseball");
   }
}
```

Figure 10-17 The `BaseballPlayer` class

```
public class ProfessionalBaseballPlayer extends BaseballPlayer
{
   double salary;
   public void showOrigins()
   {
      super.showOrigins();
      System.out.println("The first professional " +
         "major league baseball game was played in 1871");
   }
}
```

Don't Do It
A nonstatic method cannot override a `static` member of a parent class.

Figure 10-18 The `ProfessionalBaseballPlayer` class attempting to override the parent's `static` method

Figure 10-19 Error message when compiling the `ProfessionalBaseballPlayer` class in Figure 10-18

Figure 10-20 shows a second version of the `ProfessionalBaseballPlayer` class. In this version, the `showOrigins()` method has been changed to `static` in an attempt to fix the problem in Figure 10-19. Figure 10-21 shows the error message that appears when this class is compiled. Because this method version is `static`, the method is not used with an object and does not receive a `this` reference. The keyword `super` can be used in child class, nonstatic, and instance methods and constructors, but not in child class `static` methods.

```
public class ProfessionalBaseballPlayer extends BaseballPlayer
{
   double salary;
   public static void showOrigins()
   {
      super.showOrigins();
      System.out.println("The first professional " +
         "major league baseball game was played in 1871");
   }
}
```

Don't Do It
You cannot refer to `super` in a static method.

Figure 10-20 The `ProfessionalBaseballPlayer` class with a static method that attempts to reference `super`

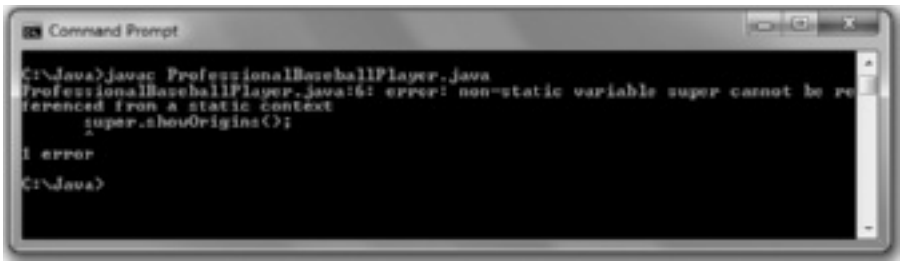

Figure 10-21 Error message when compiling the `ProfessionalBaseballPlayer` class in Figure 10-20

Finally, Figure 10-22 shows a `ProfessionalBaseballPlayer` class that compiles without error. The class extends `BaseballPlayer`, and its `showOrigins()` method is `static`. Because this method has the same name as the parent class method, when you use the name with a child class object, this method hides the original. However, it cannot use the `super` keyword to access the Abner Doubleday method. If you want the `ProfessionalBaseballPlayer` class to display information about baseball in general as well as professional baseball in particular, you can do either of the following:

- You can repeat the parent class message within the child class using a `println()` statement.
- You can use the parent class name, a dot, and the method name. Although a child class cannot inherit its parent's `static` methods, it can access its parent's `static` methods the same way any other class can. The shaded statement in Figure 10-22 uses this approach.

Figure 10-23 shows a class that creates a `ProfessionalBaseballPlayer` and tests the method; Figure 10-24 shows the output.

```
public class ProfessionalBaseballPlayer extends BaseballPlayer
{
   double salary;
   public static void showOrigins()
   {
      BaseballPlayer.showOrigins();
      System.out.println("The first professional " +
         "major league baseball game was played in 1871");
   }
}
```

Figure 10-22 The `ProfessionalBaseballPlayer` class

```
public class TestProPlayer
{
   public static void main(String[] args)
   {
      ProfessionalBaseballPlayer aYankee =
         new ProfessionalBaseballPlayer();
      aYankee.showOrigins();
   }
}
```

Figure 10-23 The `TestProPlayer` class

Figure 10-24 Output of the `TestProPlayer` application

A Subclass Cannot Override `final` Methods in Its Superclass

A subclass cannot override methods that are declared `final` in the superclass. For example, consider the `BasketballPlayer` and `ProfessionalBasketballPlayer` classes in Figures 10-25 and 10-26, respectively. When you attempt to compile the `ProfessionalBasketballPlayer` class, you receive the error message in Figure 10-27, because the class cannot override the `final displayMessage()` method in the parent class.

```
public class BasketballPlayer
{
   private int jerseyNumber;
   public final void displayMessage()
   {
      System.out.println("Michael Jordan is the " +
         "greatest basketball player - and that is final");
   }
}
```

Figure 10-25 The `BasketballPlayer` class

```
public class ProfessionalBasketballPlayer extends BasketballPlayer
{
   double salary;
   public void displayMessage()
   {
      System.out.println("I have nothing to say");
   }
}
```

Don't Do It
A child class method cannot override a `final` parent class method.

Figure 10-26 The `ProfessionalBasketballPlayer` class that attempts to override a `final` method

Figure 10-27 Error message when compiling the `ProfessionalBasketballPlayer` class in Figure 10-26

If you make the `displayMessage()` method `final` in the `ProfessionalBasketballPlayer` class in Figure 10-26, you receive the same compiler error message as shown in Figure 10-27. If you make the `displayMessage()` method `static` in the `ProfessionalBasketballPlayer` class, the class does not compile, but you do receive an additional error message.

In Chapter 2, you learned that you can use the keyword `final` when you want to create a constant, as in `final double TAXRATE = 0.065;`. You can also use the `final` modifier with methods when you don't want the method to be overridden—that is, when you want every child class to use the original parent class version of a method.

In Java, all instance method calls are **virtual method calls** by default—that is, the method used is determined when the program runs because the type of the object used might not be known until the method executes. For example, with the following method you can pass in a `BasketballPlayer` object, or any object that is a child of `BasketballPlayer`, so the "actual" type of the argument `bbplayer`, and which version of `displayMessage()` to use, is not known until the method executes.

```
public void display(BasketballPlayer bbplayer)
{
   bbplayer.displayMessage();
}
```

In other words, the version of the method used is not determined when the program is compiled; it is determined when the method call is made. Determining the correct method takes a small amount of time. An advantage to making a method `final` is that the compiler knows there is only one version of the method—the parent class version. Therefore, the compiler *does* know which method version to use—the only version—and the program is more efficient.

Because a `final` method's definition can never change—that is, can never be overridden with a modified version—the compiler can optimize a program's performance by removing the calls to `final` methods and replacing them with the expanded code of their definitions at each method call location. This process is called **inlining** the code. When a program executes, you are never aware that inlining is taking place; the compiler chooses to use this procedure to save the overhead of calling a method, and the program runs faster. The compiler chooses to inline a `final` method only if it is a small method that contains just one or two lines of code.

A Subclass Cannot Override Methods in a `final` Superclass

You can declare a class to be `final`. When you do, all of its methods are `final`, regardless of which access specifier precedes the method name. A `final` class cannot be a parent. Figure 10-28 shows two classes: a `HideAndGoSeekPlayer` class that is a `final` class because of the word `final` in the class header, and a `ProfessionalHideAndGoSeekPlayer` class that attempts to extend the `final` class, adding a salary field. Figure 10-29 shows the error message generated when you try to compile the `ProfessionalHideAndGoSeekPlayer` class.


```
public final class HideAndGoSeekPlayer
{
   private int count;
   public void displayRules()
   {
      System.out.println("You have to count to " + count +
         " before you start looking for hiders");
   }
}
public final class ProfessionalHideAndGoSeekPlayer
   extends HideAndGoSeekPlayer
{
   private double salary;
}
```

Don't Do It
You cannot extend a `final` class.

Figure 10-28 The `HideAndGoSeekPlayer` and `ProfessionalHideAndGoSeekPlayer` classes

Figure 10-29 Error message when compiling the `ProfessionalHideAndGoSeekPlayer` class in Figure 10-28

Java's `Math` class, which you learned about in Chapter 4, is an example of a `final` class.

TWO TRUTHS & A LIE

Methods You Cannot Override

1. A subclass cannot override methods that are declared `static` in the superclass.
2. A subclass cannot override methods that are declared `final` in the superclass.
3. A subclass cannot override methods that are declared `private` in the superclass.

The false statement is #3. A subclass can override `private` methods as well as `public` or `protected` ones.

Don't Do It

- Don't capitalize the *o* in the `instanceof` operator. Although the second word in an identifier frequently is capitalized in Java, `instanceof` is an exception.
- Don't try to directly access private superclass members from a subclass.
- Don't forget to call a superclass constructor from within a subclass constructor if the superclass does not contain a default constructor.
- Don't try to override a `final` method in an extended class.
- Don't try to extend a `final` class.

Key Terms

Inheritance is a mechanism that enables one class to inherit, or assume, both the behavior and the attributes of another class.

The **Unified Modeling Language (UML)** is a graphical language used by programmers and analysts to describe classes and object-oriented processes.

A **class diagram** is a visual tool that provides you with an overview of a class. It consists of a rectangle divided into three sections—the top section contains the name of the class, the middle section contains the names and data types of the attributes, and the bottom section contains the methods.

A **base class** is a class that is used as a basis for inheritance.

A **derived class** is a class that inherits from a base class.

Composition is the relationship in which one class contains one or more members of another class that would not continue to exist without the object that contains them.

Aggregation is a type of composition in which a class contains one or more members of another class that would continue to exist without the object that contains them.

Superclass and **subclass** are synonyms for base class and derived class.

Parent class and **child class** are synonyms for base class and derived class.

The keyword **`extends`** is used to achieve inheritance in Java.

The **`instanceof` operator** determines whether an object that is the operand on the left is a member or descendant of the class that is the operand on the right.

To **upcast** an object is to change it to an object of a class higher in the object's inheritance hierarchy.

Polymorphism is the technique of using the same method name to indicate different implementations.

To **override a method** in a parent class is to create a method in a child class that has the same name and parameter list as a method in its parent class.

Subtype polymorphism is the ability of one method name to work appropriately for different subclasses of a parent class.

The keyword **`super`** refers to the parent or superclass of the class in which you use it.

Information hiding is the concept of keeping data private.

The keyword **`protected`** provides you with an intermediate level of security between `public` and `private` access. `Protected` members are those that can be used by a class and its descendants.

Fragile classes are those that are prone to errors.

Virtual method calls are those in which the method used is determined when the program runs, because the type of the object used might not be known until the method executes. In Java, all instance method calls are virtual calls by default.

Inlining the code is an automatic process that optimizes performance. Because a `final` method's definition can never be overridden, the compiler can optimize a program's performance by removing the calls to `final` methods and replacing them with the expanded code of their definitions at each method call location.

Chapter Summary

- In Java, inheritance is a mechanism that enables one class to inherit both the behavior and the attributes of another class. Using inheritance saves time because the original fields and methods already exist, have been tested, and are familiar to users. A class that is used as a basis for inheritance is a base class. A class you create that inherits from a base class is called a derived class. You can use the terms *superclass* and *subclass* as synonyms for base class and derived class; you can also use the terms *parent class* and *child class*.
- You use the keyword `extends` to achieve inheritance in Java. A parent class object does not have access to its child's data and methods, but when you create a subclass by extending an existing class, the new subclass contains data and methods that were defined in the original superclass.
- Sometimes, superclass data fields and methods are not entirely appropriate for the subclass objects. Polymorphism is the act of using the same method name to indicate different implementations. You use polymorphism when you override a superclass method in a subclass by creating a method with the same name and parameter list.
- When you create any subclass object, the superclass constructor must execute first, and *then* the subclass constructor executes. When a superclass contains only constructors that require arguments, you must include at least one constructor for each subclass you create. Your subclass constructors can contain any number of statements, but the first statement within each constructor must call the superclass constructor. When a superclass requires parameters upon instantiation, even if you have no other reason to create a subclass constructor, you must write the subclass constructor so it can call its superclass's constructor. The format of the statement that calls a superclass constructor is `super(list of arguments);`.
- If you want to use a superclass method within a subclass, you can use the keyword `super` to access the parent class method.
- Subclasses inherit all the data and methods of their superclasses, but `private` members of the parent class are not accessible with a child class's methods. However, if you create a `protected` data field or method, it can be used within its own class or in any classes extended from that class, but it cannot be used by "outside" classes. A subclass cannot override methods that are declared `static` in the superclass. A subclass can *hide* a `static` method in the superclass by declaring a `static` method in the subclass with the same signature as the `static` method in the superclass. A subclass cannot override methods that are declared `final` in the superclass or methods declared within a `final` class.

Review Questions

1. A way to discover which of two classes is the base class and which is the subclass is to _____.

   a. look at the class size
   b. try saying the two class names together
   c. use polymorphism
   d. Both a and b are correct.

2. Employing inheritance reduces errors because _____.

   a. the new classes have access to fewer data fields
   b. the new classes have access to fewer methods
   c. you can copy methods that you already created
   d. many of the methods you need have already been used and tested

3. A base class can also be called a _____.

   a. child class
   b. subclass
   c. derived class
   d. superclass

4. Which of the following choices is the best example of a parent class/child class relationship?

   a. `Rose/Flower`
   b. `Present/Gift`
   c. `Dog/Poodle`
   d. `Sparrow/Bird`

5. The Java keyword that creates inheritance is _____.

   a. `static`
   b. `enlarge`
   c. `extends`
   d. `inherits`

6. A class named `Building` has a `public`, nonstatic method named `getFloors()`. If `School` is a child class of `Building`, and `modelHigh` is an object of type `School`, which of the following statements is valid?

   a. `Building.getFloors();`
   b. `School.getFloors();`
   c. `modelHigh.getFloors();`
   d. All of the previous statements are valid.

7. Which of the following statements is true?
   a. A child class inherits from a parent class.
   b. A parent class inherits from a child class.
   c. Both of the preceding statements are true.
   d. Neither a nor b is true.

8. When a subclass method has the same name and argument types as a superclass method, the subclass method _____ the superclass method.
   a. overrides
   b. overuses
   c. overloads
   d. overcompensates

9. When you instantiate an object that is a member of a subclass, the _____ constructor executes first.
   a. subclass
   b. child class
   c. extended class
   d. parent class

10. The keyword `super` always refers to the _____ of the class in which you use it.
    a. child class
    b. derived class
    c. subclass
    d. parent class

11. If the only constructor in a superclass requires arguments, its subclass _____.
    a. must contain a constructor
    b. must not contain a constructor
    c. must contain a constructor that requires arguments
    d. must not contain a constructor that requires arguments

12. If a superclass constructor requires arguments, any constructor of its subclasses must call the superclass constructor _____.
    a. as the first statement
    b. as the last statement
    c. at some time
    d. multiple times if multiple arguments are involved

13. A child class `Motorcycle` extends a parent class `Vehicle`. Each class constructor requires one `String` argument. The `Motorcycle` class constructor can call the `Vehicle` class constructor with the statement _____.
    a. `Vehicle("Honda");`
    b. `Motorcycle("Harley");`
    c. `super("Suzuki");`
    d. none of the above

14. In Java, the concept of keeping data private is known as _____.

    a. polymorphism
    b. information hiding
    c. data deception
    d. concealing fields

15. If you create a data field or method that is _____, it can be used within its own class or in any classes extended from that class.

    a. `public`
    b. `protected`
    c. `private`
    d. both a and b

16. Within a subclass, you cannot override _____ methods.

    a. `public`
    b. `private`
    c. `static`
    d. constructor

17. You call a `static` method using _____.

    a. the name of its class, a dot, and the method name
    b. the name of the class's superclass, a dot, and the method name
    c. the name of an object in the same class, a dot, and the method name
    d. either a or b

18. You use a _____ method access specifier when you create methods for which you want to prevent overriding in extended classes.

    a. `public`
    b. `protected`
    c. `final`
    d. subclass

19. A compiler can decide to _____ a `final` method—that is, determine the code of the method call when the program is compiled.

    a. duplicate
    b. inline
    c. redline
    d. beeline

20. When a parent class contains a `static` method, child classes _____ override it.

    a. frequently
    b. seldom
    c. must
    d. cannot

Exercises

Programming Exercises

1. Create a class named `Horse` that contains data fields for the name, color, and birth year. Include get and set methods for these fields. Next, create a subclass named `RaceHorse`, which contains an additional field that holds the number of races in which the horse has competed and additional methods to get and set the new field. Write an application that demonstrates using objects of each class. Save the files as **Horse.java**, **RaceHorse.java**, and **DemoHorses.java**.

2. Mick's Wicks makes candles in various sizes. Create a class for the business named `Candle` that contains data fields for `color`, `height`, and `price`. Create get methods for all three fields. Create set methods for `color` and `height`, but not for `price`. Instead, when `height` is set, determine the `price` as $2 per inch. Create a child class named `ScentedCandle` that contains an additional data field named `scent` and methods to get and set it. In the child class, override the parent's `setHeight()` method to set the price of a `ScentedCandle` object at $3 per inch. Write an application that instantiates an object of each type and displays the details. Save the files as **Candle.java**, **ScentedCandle.java**, and **DemoCandles.java**.

3. Create a class named `TennisGame` that holds data about a single tennis game. The class has six fields: the names of the two players, the integer final scores for the players, and the `String` values of the final scores. Include a get method for each of the six fields. Also include a set method that accepts two players' names, and another set method that accepts the two integer final score values. The integer final score for a player is the number of points the player won during the game; this value should be in the range of 0 through 4. If either of the set method parameters for a score is not in the range of 0 through 4, assign 0 to both scores and assign "error" to the `String` scores. If both players' score parameters are 4, assign 0 to both scores and "error" to the `String` scores. The `String` score values are set by the method that sets the integer score values. The `String` final score for each player contains the traditional names for points in tennis: *love*, *15*, *30*, *40*, or *game*, respectively, for the values 0 through 4. Create a subclass named `DoublesTennisGame` that includes two additional fields for the names of the first two players' partners. Include get methods for the names. Override the parent class `setNames()` method to accept the names of all four players. Write an application named `DemoTennisGames` that instantiates several objects of each of these classes. Demonstrate that all the methods assign correct values. Save the files as **TennisGame.java**, **DoublesTennisGame.java**, and **DemoTennisGames.java**.

4. a. Create a class named `Year` that contains a data field that holds the number of days in a year. Include a get method that displays the number of days and a constructor that sets the number of days to 365. Create a subclass named `LeapYear`. `LeapYear`'s constructor overrides `Year`'s constructor and sets the number of days

to 366. Write an application named `UseYear` that instantiates one object of each class and displays their data. Save the files as **Year.java**, **LeapYear.java**, and **UseYear.java**.

b. Add a method named `daysElapsed()` to the `Year` class you created in Exercise 4a. The `daysElapsed()` method accepts two arguments representing a month and a day; it returns an integer indicating the number of days that have elapsed since January 1 of that year. For example, on March 3 in nonleap years, 61 days have elapsed (31 in January, 28 in February, and 2 in March). Create a `daysElapsed()` method for the `LeapYear` class that overrides the method in the `Year` class. For example, on March 3 in a `LeapYear`, 62 days have elapsed (31 in January, 29 in February, and 2 in March). Write an application named `UseYear2` that prompts the user for a month and day, and calculates the days elapsed in a `Year` and in a `LeapYear`. Save the files as **Year2.java**, **LeapYear2.java**, and **UseYear2.java**.

5. Every summer, Leeland Lakeside resort rents cabins by the week. Create a class named `CabinRental` that includes an integer field for the cabin number and a `double` field for the weekly rental rate. Include get methods for these fields and a constructor that requires an integer argument representing the cabin number. The constructor sets the weekly rate based on the cabin number; cabins numbered 1, 2, and 3 are $950 per week, and others are $1,100 per week. Create an extended class named `HolidayCabinRental` that is used for rentals during weeks that include summer holiday weekends. The constructor for this class requires a room number and adds a $150 surcharge to the regular rental rate. Write an application named `DemoCabinRental` that creates an object of each class, and demonstrate that all the methods work correctly. Save the files as **CabinRental.java**, **HolidayCabinRental.java**, and **DemoCabinRental.java**.

6. Create a class named `Package` with data fields for weight in ounces, shipping method, and shipping cost. The shipping method is a character: *A* for air, *T* for truck, or *M* for mail. The `Package` class contains a constructor that requires arguments for weight and shipping method. The constructor calls a `calculateCost()` method that determines the shipping cost, based on the following table:

Weight (oz.)	Air ($)	Truck ($)	Mail ($)
1 to 8	2.00	1.50	.50
9 to 16	3.00	2.35	1.50
17 and over	4.50	3.25	2.15

The `Package` class also contains a `display()` method that displays the values in all four fields. Create a subclass named `InsuredPackage` that adds an insurance cost to the shipping cost, based on the following table:

Shipping Cost Before Insurance ($)	Additional Cost ($)
0 to 1.00	2.45
1.01 to 3.00	3.95
3.01 and over	5.55

Write an application named `UsePackage` that instantiates at least three objects of each type (`Package` and `InsuredPackage`) using a variety of weights and shipping method codes. Display the results for each `Package` and `InsuredPackage`. Save the files as **Package.java**, **InsuredPackage.java**, and **UsePackage.java**.

7. Create a class named `CarRental` that contains fields that hold a renter's name, zip code, size of the car rented, daily rental fee, length of rental in days, and total rental fee. The class contains a constructor that requires all the rental data except the daily rate and total fee, which are calculated based on the size of the car: economy at $29.99 per day, midsize at $38.99 per day, or full size at $43.50 per day. The class also includes a `display()` method that displays all the rental data. Create a subclass named `LuxuryCarRental`. This class sets the rental fee at $79.99 per day and prompts the user to respond to the option of including a chauffeur at $200 more per day. Override the parent class `display()` method to include chauffeur fee information. Write an application named `UseCarRental` that prompts the user for the data needed for a rental and creates an object of the correct type. Display the total rental fee. Save the files as **CarRental.java**, **LuxuryCarRental.java**, and **UseCarRental.java**.

8. Create a class named `CollegeCourse` that includes data fields that hold the department (for example, ENG), the course number (for example, 101), the credits (for example, 3), and the fee for the course (for example, $360). All of the fields are required as arguments to the constructor, except for the fee, which is calculated at $120 per credit hour. Include a `display()` method that displays the course data. Create a subclass named `LabCourse` that adds $50 to the course fee. Override the parent class `display()` method to indicate that the course is a lab course and to display all the data. Write an application named `UseCourse` that prompts the user for course information. If the user enters a class in any of the following departments, create a `LabCourse`: BIO, CHM, CIS, or PHY. If the user enters any other department, create a `CollegeCourse` that does not include the lab fee. Then display the course data. Save the files as **CollegeCourse.java**, **LabCourse.java**, and **UseCourse.java**.

9. Create a class named `Vehicle` that acts as a superclass for vehicle types. The `Vehicle` class contains private variables for the number of wheels and the average number of miles per gallon. The `Vehicle` class also contains a constructor with integer arguments for the number of wheels and average miles per gallon, and a

`toString()` method that returns a `String` containing these values. Create two subclasses, `Car` and `MotorCycle`, that extend the `Vehicle` class. Each subclass contains a constructor that accepts the miles-per-gallon value as an argument and forces the number of wheels to the appropriate value—2 for a `MotorCycle` and 4 for a `Car`. Write a `UseVehicle` class to instantiate the two `Vehicle` objects and display the objects' values. Save the files as **Vehicle.java**, **Car.java**, **MotorCycle.java**, and **UseVehicle.java**.

10. Develop a set of classes for a college to use in various student service and personnel applications. Classes you need to design include the following:

    - `Person`—A `Person` contains a first name, last name, street address, zip code, and phone number. The class also includes a method that sets each data field, using a series of dialog boxes and a display method that displays all of a `Person`'s information on a single line at the command line on the screen.
    - `CollegeEmployee`—`CollegeEmployee` descends from `Person`. A `CollegeEmployee` also includes a Social Security number, an annual salary, and a department name, as well as methods that override the `Person` methods to accept and display all `CollegeEmployee` data.
    - `Faculty`—`Faculty` descends from `CollegeEmployee`. This class also includes a Boolean field that indicates whether the `Faculty` member is tenured, as well as methods that override the `CollegeEmployee` methods to accept and display this additional piece of information.
    - `Student`—`Student` descends from `Person`. In addition to the fields available in `Person`, a `Student` contains a major field of study and a grade point average as well as methods that override the `Person` methods to accept and display these additional facts.

    Write an application named `CollegeList` that declares an array of four "regular" `CollegeEmployees`, three `Faculty`, and seven `Students`. Prompt the user to specify which type of person's data will be entered (*C*, *F*, or *S*), or allow the user to quit (*Q*). While the user chooses to continue (that is, does not quit), accept data entry for the appropriate type of `Person`. If the user attempts to enter data for more than four `CollegeEmployees`, three `Faculty`, or seven `Students`, display an error message. When the user quits, display a report on the screen listing each group of `Persons` under the appropriate heading of College Employees, Faculty, or Students. If the user has not entered data for one or more types of `Persons` during a session, display an appropriate message under the appropriate heading.

    Save the files as **Person.java**, **CollegeEmployee.java**, **Faculty.java**, **Student.java**, and **CollegeList.java**.

Debugging Exercises

1. Each of the following files in the Chapter10 folder of your downloadable student files has syntax and/or logic errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fix*. For example, DebugTen1.java will become FixDebugTen1.java.
   a. DebugTen1.java
   b. DebugTen2.java
   c. DebugTen3.java
   d. DebugTen4.java
   e. Eight other Debug files in the Chapter10 folder; these files are used by the DebugTen exercises

Game Zone

1. a. Create an `Alien` class. Include at least three `protected` data members of your choice, such as the number of eyes the `Alien` has. Include a constructor that requires a value for each data field and a `toString()` method that returns a `String` containing a complete description of the `Alien`. Save the file as **Alien.java**.
   b. Create two classes—`Martian` and `Jupiterian`—that descend from `Alien`. Supply each with a constructor that sets the `Alien` data fields with values you choose. For example, you can decide that a `Martian` has four eyes but a `Jupiterian` has only two. Save the files as **Martian.java** and **Jupiterian.java**.
   c. Create an application that instantiates one `Martian` and one `Jupiterian`. Call the `toString()` method with each object and display the results. Save the application as **CreateAliens.java**.
2. a. In Chapter 4, you created a `Die` class that you can use to instantiate objects that hold one of six randomly selected values. Modify this class so its value field is `protected` instead of `private`. This will allow a child class to access the value. Save the file as **Die.java**.
   b. Create a `LoadedDie` class that can be used to give a player a slight advantage over the computer. A `LoadedDie` never rolls a 1; it only rolls values 2 through 6. Save the file as **LoadedDie.java**.
   c. Create a program that rolls two `Die` objects against each other 1,000 times and counts the number of times the first `Die` has a higher value than the other `Die`. Then roll a `Die` object against a `LoadedDie` object 1,000 times, and count the number of times the `Die` wins. Display the results. Save the application as **TestLoadedDie.java**. Figure 10-30 shows two typical executions.

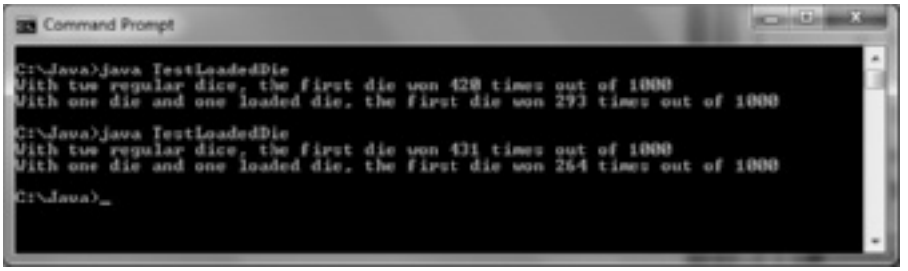

Figure 10-30 Two typical executions of the `TestLoadedDie` application

Case Problems

1. a. In Chapter 8, you created an `Event` class for Carly's Catering. Now extend the class to create a `DinnerEvent` class. In the extended class, include four new integer fields that represent numeric choices for an entrée, two side dishes, and a dessert for each `DinnerEvent` object. Also include three `final` arrays that contain `String` menu options for entrées, side dishes, and desserts, and store at least three choices in each array. Create a `DinnerEvent` constructor that requires arguments for an event number and number of guests, and integer menu choices for one entrée, two side dishes, and one dessert. Pass the first two parameters to the `Event` constructor, and assign the last four parameters to the appropriate local fields. Also include a `getMenu()` method that builds and returns a `String` including the `String`s for the four menu choices. Save the file as **DinnerEvent.java**.

   b. In Chapter 9, you created an `EventDemo` program for Carly's Catering. The program uses an array of `Event` objects and allows the user to sort `Event`s in ascending order by event number, number of guests, or event type. Now modify the program to use an array of four `DinnerEvent` objects. Prompt the user for all values for each object, and then allow the user to continuously sort the `DinnerEvent` descriptions by event number, number of guests, or event type. Save the file as **DinnerEventDemo.java**.

2. a. In Chapter 8, you created a `Rental` class for Sammy's Seashore Supplies. Now extend the class to create a `LessonWithRental` class. In the extended class, include a new Boolean field that indicates whether a lesson is required or optional for the type of equipment rented. Also include a `final` array that contains `String`s representing the names of the instructors for each of the eight equipment types, and store names that you choose in the array. Create a `LessonWithRental` constructor that requires arguments for an event number, minutes for the rental, and an integer equipment type. Pass the first two parameters to the `Rental` constructor, and assign the last parameter to the equipment type. For the first two equipment types (jet ski and pontoon boat),

set the Boolean lesson required field to true; otherwise, set it to false. Also include a `getInstructor()` method that builds and returns a `String` including the `String` for the equipment type, a message that indicates whether a lesson is required, and the instructor's name. Save the file as **LessonWithRental.java**.

b. In Chapter 9, you created a `RentalDemo` program for Sammy's Seashore Supplies. The program uses an array of `Rental` objects and allows the user to sort `Rentals` in ascending order by contract number, equipment type, or price. Now modify the program to use an array of four `LessonWithRental` objects. Prompt the user for all values for each object, and then allow the user to continuously sort the `LessonWithRental` descriptions by contract number, equipment type, or price. Save the file as **LessonWithRentalDemo.java**.