CHAPTER 11

Advanced Inheritance Concepts

In this chapter, you will:

- Create and use abstract classes
- Use dynamic method binding
- Create arrays of subclass objects
- Use the `Object` class and its methods
- Use inheritance to achieve good software design
- Create and use interfaces
- Create and use packages

Creating and Using Abstract Classes

Developing new classes is easier after you understand the concept of inheritance. When you use a class as a basis from which to create extended child classes, the child classes are more specific than their parent. When you create a child class, it inherits all the general attributes you need; thus, you must create only the new, more specific attributes. For example, a `SalariedEmployee` and an `HourlyEmployee` are more specific than an `Employee`. They can inherit general `Employee` attributes, such as an employee number, but they add specific attributes, such as pay-calculating methods.

Notice that a superclass contains the features that are shared by its subclasses. For example, the attributes of the `Dog` class are shared by every `Poodle` and `Spaniel`. The subclasses are more specific examples of the superclass type; they add more features to the shared, general features. Conversely, when you examine a subclass, you see that its parent is more general and less specific; for example, `Animal` is more general than `Dog`.

Recall from Chapter 10 that the terms *base class*, *superclass*, and *parent* are equivalent. Similarly, the terms *derived class*, *subclass*, and *child* are equivalent. Also recall that a child class contains all the members of its parent, whether those members are `public`, `protected`, or `private`. However, a child object cannot directly access a `private` member inherited from a parent.

A **concrete class** is one from which you can instantiate objects. Sometimes, a class is so general that you never intend to create any specific instances of the class. For example, you might never create an object that is "just" an `Employee`; each `Employee` is more specifically a `SalariedEmployee`, `HourlyEmployee`, or `ContractEmployee`. A class such as `Employee` that you create only to extend from is not a concrete class; it is an **abstract class**. In the last chapter, you learned that you can create `final` classes if you do not want other classes to be able to extend them. Classes that you declare to be `abstract` are the opposite; your only purpose in creating them is to enable other classes to extend them. If you attempt to instantiate an object from an abstract class, you receive an error message from the compiler that you have committed an `InstantiationError`. You use the keyword `abstract` when you declare an abstract class. (In other programming languages, such as C++, abstract classes are known as **virtual classes**.)

In the last chapter, you learned to create class diagrams. By convention, when you show abstract classes and methods in class diagrams, their names appear in italics.

In Chapter 4, you worked with the `GregorianCalendar` class. `GregorianCalendar` is a concrete class that extends the abstract class `Calendar`. In other words, there are no "plain" `Calendar` objects.

Programmers of an abstract class can include two method types:

- Nonabstract methods, like those you can create in any class, are implemented in the abstract class and are simply inherited by its children.
- **Abstract methods** have no body and must be implemented in child classes.

Abstract classes usually contain at least one abstract method. When you create an abstract method, you provide the keyword `abstract` and the rest of the method header, including the method type, name, and parameters. However, the declaration ends there: you do not provide curly braces or any statements within the method—just a semicolon at the end of the declaration. If you create an empty method within an abstract class, the method is an abstract method even if you do not explicitly use the keyword `abstract` when defining the method, but programmers often include the keyword for clarity. If you declare a class to be abstract, its methods can be abstract or not, but if you declare a method to be abstract, you must also declare its class to be abstract.

When you create a subclass that inherits an abstract method, you write a method with the same signature. You are required to code a subclass method to override every empty, abstract superclass method that is inherited. Either the child class method must itself be abstract, or you must provide a body, or implementation, for the inherited method.

Suppose that you want to create classes to represent different animals, such as `Dog` and `Cow`. You can create a generic abstract class named `Animal` so you can provide generic data fields, such as the animal's name, only once. An `Animal` is generic, but all specific `Animals` make a sound; the actual sound differs from `Animal` to `Animal`. If you code an empty `speak()` method in the abstract `Animal` class, you require all future `Animal` subclasses to code a `speak()` method that is specific to the subclass. Figure 11-1 shows an abstract `Animal` class containing a data field for the name, `getName()` and `setName()` methods, and an abstract `speak()` method.

```
public abstract class Animal
{
   private String name;
   public abstract void speak();
   public String getName()
   {
      return name;
   }
   public void setName(String animalName)
   {
      name = animalName;
   }
}
```

Figure 11-1 The abstract `Animal` class

The `Animal` class in Figure 11-1 is declared as `abstract`; the keyword is shaded. You cannot create a class in which you declare an `Animal` object with a statement such as `Animal myPet = new Animal("Murphy");`, because a class that attempts to instantiate an `Animal` object does not compile. `Animal` is an abstract class, so no `Animal` objects can exist.

You create an abstract class such as `Animal` only so you can extend it. For example, because a dog is an animal, you can create a `Dog` class as a child class of `Animal`. Figure 11-2 shows a `Dog` class that extends `Animal`.

```
public class Dog extends Animal
{
   public void speak()
   {
      System.out.println("Woof!");
   }
}
```

Figure 11-2 The Dog class

The `speak()` method within the `Dog` class is required because you want to create `Dog` objects and the abstract, parent `Animal` class contains an abstract `speak()` method (shaded in Figure 11-1). You can code any statements you want within the `Dog speak()` method, but the `speak()` method must exist. If you do not want to create `Dog` objects but want the `Dog` class to be a parent to further subclasses, then the `Dog` class must also be abstract. In that case, you can write code for the `speak()` method within the subclasses of `Dog`.

If `Animal` is an abstract class, you cannot instantiate an `Animal` object; however, if `Dog` is a concrete class, instantiating a `Dog` object is perfectly legal. When you code the following, you create a `Dog` object:

```
Dog myPet = new Dog("Murphy");
```

Then, when you code `myPet.speak();`, the correct `Dog speak()` method executes.

The classes in Figures 11-3 and 11-4 also inherit from the `Animal` class and implement `speak()` methods. Figure 11-5 contains a `UseAnimals` application.

```
public class Cow extends Animal
{
   public void speak()
   {
      System.out.println("Moo!");
   }
}
```

Figure 11-3 The Cow class

```java
public class Snake extends Animal
{
   public void speak()
   {
      System.out.println("Ssss!");
   }
}
```

Figure 11-4 The Snake class

```java
public class UseAnimals
{
   public static void main(String[] args)
   {
      Dog myDog = new Dog();
      Cow myCow = new Cow();
      Snake mySnake = new Snake();
      myDog.setName("My dog Murphy");
      myCow.setName("My cow Elsie");
      mySnake.setName("My snake Sammy");
      System.out.print(myDog.getName() + " says ");
      myDog.speak();
      System.out.print(myCow.getName() + " says ");
      myCow.speak();
      System.out.print(mySnake.getName() + " says ");
      mySnake.speak();
   }
}
```

Figure 11-5 The `UseAnimals` application

The output in Figure 11-6 shows that when you create `Dog`, `Cow`, and `Snake` objects, each is an `Animal` with access to the `Animal` class `getName()` and `setName()` methods, and each uses its own `speak()` method appropriately.

In Figure 11-6, notice how the `myDog.getName()` and `myDog.speak()` method calls produce different output from when the same method names are used with `myCow` and `mySnake`.

Recall that using the same method name to indicate different implementations is *polymorphism*. Using polymorphism, one method name causes different and appropriate actions for diverse types of objects.

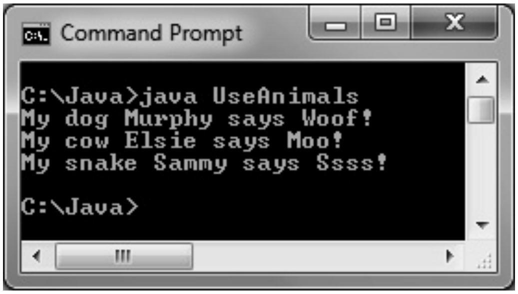

Figure 11-6 Output of the `UseAnimals` application

Watch the video *Abstract Classes*.

TWO TRUTHS & A LIE

Creating and Using Abstract Classes

1. An abstract class is one from which you cannot inherit, but from which you can create concrete objects.
2. Abstract classes usually have one or more empty abstract methods.
3. An abstract method has no body, curly braces, or statements.

The false statement is #1. An abstract class is one from which you cannot create any concrete objects, but from which you can inherit.

You Do It

Creating an Abstract Class

In this section, you create an abstract `Vehicle` class. The class includes fields for the power source, the number of wheels, and the price. `Vehicle` is an abstract class; there will never be a “plain” `Vehicle` object. Later, you will create two subclasses, `Sailboat` and `Bicycle`; these more specific classes include price limits for the vehicle type, as well as different methods for displaying data.

1. Open a new file in your text editor, and enter the following first few lines to begin creating an abstract `Vehicle` class:

```
public abstract class Vehicle
{
```

2. Declare the data fields that hold the power source, number of wheels, and price. Declare `price` as `protected` rather than `private`, because you want child classes to be able to access the field.

```
private String powerSource;
private int wheels;
protected int price;
```

3. The `Vehicle` constructor accepts three parameters and calls three methods. The first method accepts the `powerSource` parameter, the second

(continues)

(continued)

accepts the `wheels` parameter, and the third method prompts the user for a vehicle price.

```
public Vehicle(String powerSource, int wheels)
{
   setPowerSource(powerSource);
   setWheels(wheels);
   setPrice();
}
```

4. Include the following three get methods that return the values for the data fields:

```
public String getPowerSource()
{
   return powerSource;
}
public int getWheels()
{
   return wheels;
}
public int getPrice()
{
   return price;
}
```

5. Enter the following set methods, which assign values to the `powerSource` and `wheels` fields.

```
public void setPowerSource(String source)
{
   powerSource = source;
}
public void setWheels(int wls)
{
   wheels = wls;
}
```

6. The `setPrice()` method is an abstract method. Each subclass you eventually create that represents different vehicle types will have a unique prompt for the price and a different maximum allowed price. Type the abstract method definition and the closing curly brace for the class:

```
   public abstract void setPrice();
}
```

7. Save the file as **Vehicle.java**. At the command prompt, compile the file using the **javac** command.

(continues)

(continued)

Extending an Abstract Class

You just created an abstract class, but you cannot instantiate any objects from this class. Rather, you must extend this class to be able to create any `Vehicle`-related objects. Next, you create a `Sailboat` class that extends the `Vehicle` class. This new class is concrete; that is, you can create actual `Sailboat` class objects.

1. Open a new file in your text editor, and then type the following, including a header for a `Sailboat` class that extends the `Vehicle` class:

```
import javax.swing.*;
public class Sailboat extends Vehicle
{
```

2. Add the declaration of a length field that is specific to a `Sailboat` by typing the following code:

```
private int length;
```

3. The `Sailboat` constructor must call its parent's constructor and send two arguments to provide values for the `powerSource` and `wheels` values. It also calls the `setLength()` method that prompts the user for and sets the length of the `Sailboat` objects:

```
public Sailboat()
{
   super("wind", 0);
   setLength();
}
```

4. Enter the following `setLength()` and `getLength()` methods, which respectively ask for and return the `Sailboat`'s length:

```
public void setLength()
{
   String entry;
   entry = JOptionPane.showInputDialog
     (null, "Enter sailboat length in feet ");
   length = Integer.parseInt(entry);
}
public int getLength()
{
   return length;
}
```

5. The concrete `Sailboat` class must contain a `setPrice()` method because the method is abstract in the parent class. Assume that a `Sailboat` has a maximum price of $100,000. Add the following `setPrice()` method that

(continues)

(continued)

prompts the user for the price and forces it to the maximum value if the entered value is too high:

```
public void setPrice()
{
   String entry;
   final int MAX = 100000;
   entry = JOptionPane.showInputDialog
     (null, "Enter sailboat price ");
   price = Integer.parseInt(entry);
   if(price > MAX)
      price = MAX;
}
```

6. In Chapter 7, you first used the automatically included `Object` class `toString()` method that converts any object to a `String`. Now, you can override that method for this class by writing your own version as follows. When you finish, add a closing curly brace for the class.

```
   public String toString()
   {
      return("The " + getLength() +
         " foot sailboat is powered by " +
         getPowerSource() + "; it has " + getWheels() +
         " wheels and costs $" + getPrice());
   }
}
```

7. Save the file as **Sailboat.java**, and then compile the class.

Extending an Abstract Class with a Second Subclass

The `Bicycle` class inherits from `Vehicle`, just as the `Sailboat` class does. Whereas the `Sailboat` class requires a data field to hold the length of the boat, the `Bicycle` class does not. Other differences lie in the content of the `setPrice()` and `toString()` methods.

1. Open a new file in your text editor, and then type the following first lines of the `Bicycle` class:

```
import javax.swing.*;
public class Bicycle extends Vehicle
{
```

2. Enter the following `Bicycle` class constructor, which calls the parent constructor, sending it power source and wheel values:

```
public Bicycle()
{
    super("a person", 2);
}
```

(continues)

(continued)

3. Enter the following `setPrice()` method that forces a `Bicycle`'s price to be no greater than $4,000:

```
public void setPrice()
{
   String entry;
   final int MAX = 4000;
   entry = JOptionPane.showInputDialog
     (null, "Enter bicycle price ");
   price = Integer.parseInt(entry);
   if(price > MAX)
      price = MAX;
}
```

4. Enter the following `toString()` method, and add the closing curly brace for the class:

```
   public String toString()
   {
      return("The bicycle is powered by " + getPowerSource() +
         "; it has " + getWheels() + " wheels and costs $" +
         getPrice());
   }
}
```

5. Save the file as **Bicycle.java**, and then compile the class.

Instantiating Objects from Subclasses

Next, you create a program that instantiates concrete objects from each of the two child classes you just created.

1. Open a new file in your text editor, and then enter the start of the `DemoVehicles` class as follows:

```
import javax.swing.*;
public class DemoVehicles
{
   public static void main(String[] args)
   {
```

2. Enter the following statements that declare an object of each subclass type.

```
Sailboat aBoat = new Sailboat();
Bicycle aBike = new Bicycle();
```

3. Enter the following statement to display the contents of the two objects. Add the closing curly braces for the `main()` method and the class:

```
      JOptionPane.showMessageDialog(null,
        "\nVehicle descriptions:\n" +
        aBoat.toString() + "\n" + aBike.toString());
   }
}
```

(continues)

(continued)

4. Save the file as **DemoVehicles.java**, and then compile it. After you compile the class with no errors, run this application using the **java DemoVehicles** command. When the application prompts you, enter the length and price for a sailboat, and the price for a bicycle. Figure 11-7 shows output after typical user input.

Figure 11-7 Typical output of the `DemoVehicles` application

Using Dynamic Method Binding

When you create a superclass and one or more subclasses, each object of each subclass "is a" superclass object. Every `SalariedEmployee` "is an" `Employee`; every `Dog` "is an" `Animal`. (The opposite is not true. Superclass objects are not members of any of their subclasses. An `Employee` is not a `SalariedEmployee`. An `Animal` is not a `Dog`.) Because every subclass object "is a" superclass member, you can convert subclass objects to superclass objects.

As you are aware, when a superclass is abstract, you cannot instantiate objects of the superclass; however, you can indirectly create a reference to a superclass abstract object. A reference is not an object, but it points to a memory address. When you create a reference, you do not use the keyword `new` to create a concrete object; instead, you create a variable name in which you can hold the memory address of a concrete object. So, although a reference to an abstract superclass object is not concrete, you can store a concrete subclass object reference there.

You learned how to create a reference in Chapter 4. When you code `SomeClass someObject;`, you are creating a reference. If you later code the following statement, including the keyword `new` and the constructor name, then you actually set aside memory for `someObject`:

```
someObject = new SomeClass();
```

For example, if you create an `Animal` class, as shown previously in Figure 11-1, and various subclasses, such as `Dog`, `Cow`, and `Snake`, as shown in Figures 11-2 through 11-4, you can create an application containing a generic `Animal` reference variable into which you can assign any of the concrete `Animal` child objects. Figure 11-8 shows an `AnimalReference` application, and Figure 11-9 shows its output. The variable `animalRef` is a type of `Animal`. No superclass `Animal` object is created (none can be); instead, `Dog` and `Cow` objects are created using the `new` keyword. When the `Cow` object is assigned to the `Animal` reference, the `animalRef.speak()` method call results in "Moo!"; when the `Dog` object is assigned to the `Animal` reference, the method call results in "Woof!" Recall that assigning a variable or constant of one type to a variable of another type is called *promotion*, *implicit conversion*, or *upcasting*.

```
public class AnimalReference
{
   public static void main(String[] args)
   {
      Animal animalRef;
      animalRef = new Cow();
      animalRef.speak();
      animalRef = new Dog();
      animalRef.speak();
   }
}
```

Figure 11-8 The `AnimalReference` application

Figure 11-9 Output of the `AnimalReference` application

The application in Figure 11-8 shows that using a reference polymorphically allows you to extend a base class and use extended objects when a base class type is expected. For example, you could pass a `Dog` or a `Cow` to a method that expects an `Animal`. This means that all methods written to accept a superclass argument can also be used with its children—a feature that saves child-class creators a lot of work.

Recall from Chapter 10 that you can use the `instanceof` keyword to determine whether an object is an instance of any class in its hierarchy. For example, both of the following expressions are true if `myPoodle` is a `Dog` object and `Dog` is an `Animal` subclass:

```
myPoodle instanceof Animal
myPoodle instanceof Dog
```


The application in Figure 11-8 demonstrates polymorphic behavior. The same statement, `animalRef.speak();`, repeats after `animalRef` is assigned each new animal type. Each call to the `speak()` method results in different output. Each reference "chooses" the correct `speak()` method, based on the type of animal referenced. This flexible behavior is most useful when you pass references to methods; you will learn more about this in the next section. In the last chapter, you learned that in Java all instance method calls are virtual method calls by default—the method that is used is determined when the program runs, because the type of the object used might not be known until the method executes. An application's ability to select the correct subclass method depending on the argument type is known as **dynamic method binding**. When the application executes, the correct method is attached (or bound) to the application based on the current, changing (dynamic) context. Dynamic method binding is also called **late method binding**. The opposite of dynamic method binding is **static (fixed) method binding**. In Java, instance methods (those that receive a `this` reference) use dynamic binding; class methods use static method binding. Dynamic binding makes programs flexible; however, static binding operates more quickly.

In the example in this section, the objects using `speak()` happen to be related (`Cow` and `Dog` are both `Animal`s). Be aware that polymorphic behavior can apply to nonrelated classes as well. For example, a `DebateStudent` and a `VentriloquistsDummy` might also `speak()`. When polymorphic behavior depends on method overloading, it is called **ad-hoc polymorphism**; when it depends on using a superclass as a method parameter, it is called **pure polymorphism** or **inclusion polymorphism**.

Using a Superclass as a Method Parameter Type

Dynamic method binding is most useful when you want to create a method that has one or more parameters that might be one of several types. For example, the shaded header for the `talkingAnimal()` method in Figure 11-10 accepts any type of `Animal` argument. The method can be used in programs that contain `Dog` objects, `Cow` objects, or objects of any other class that descends from `Animal`. The application passes first a `Dog` and then a `Cow` to the method. The output in Figure 11-11 shows that the method works correctly no matter which type of `Animal` descendant it receives.

```
public class TalkingAnimalDemo
{
   public static void main(String[] args)
   {
      Dog dog = new Dog();
      Cow cow = new Cow();
      dog.setName("Ginger");
      cow.setName("Molly");
      talkingAnimal(dog);
      talkingAnimal(cow);
   }
   public static void talkingAnimal(Animal animal)
   {
      System.out.println("Come one. Come all.");
      System.out.println
         ("See the amazing talking animal!");
      System.out.println(animal.getName() +
         " says");
      animal.speak();
      System.out.println("***************");
   }
}
```

Figure 11-10 The `TalkingAnimalDemo` class

Figure 11-11 Output of the `TalkingAnimalDemo` application

TWO TRUTHS & A LIE

Using Dynamic Method Binding

1. If `Parent` is a parent class and `Child` is its child, then you can assign a `Child` object to a `Parent` reference.
2. If `Parent` is a parent class and `Child` is its child, then you can assign a `Parent` object to a `Child` reference.
3. Dynamic method binding refers to a program's ability to select the correct subclass method for a superclass reference while a program is running.

The false statement is #2. If `Parent` is a parent class and `Child` is its child, then you cannot assign a `Parent` object to a `Child` reference; you can assign a `Child` object only to a `Child` reference. However, you can assign a `Parent` object or a `Child` object to a `Parent` reference.

Creating Arrays of Subclass Objects

Recall that every array element must be the same data type, which can be a primitive, built-in type or a type based on a more complex class. When you create an array in Java, you are not constructing objects. Instead, you are creating space for references to objects. In other words, although it is convenient to refer to "an array of objects," every array of objects is really an array of object references. When you create an array of superclass references, it can hold subclass references. This is true whether the superclass in question is abstract or concrete.

For example, even though `Employee` is an abstract class, and every `Employee` object is either a `SalariedEmployee` or an `HourlyEmployee` subclass object, it can be convenient to create an array of generic `Employee` references. Likewise, an `Animal` array might contain individual elements that are `Dog`, `Cow`, or `Snake` objects. As long as every `Employee` subclass has access to a `calculatePay()` method, and every `Animal` subclass has access to a `speak()` method, you can manipulate an array of superclass objects and invoke the appropriate method for each subclass member.

The following statement creates an array of three `Animal` references:

```
Animal[] animalRef = new Animal[3];
```

The statement reserves enough computer memory for three `Animal` objects named `animalRef[0]`, `animalRef[1]`, and `animalRef[2]`. The statement does not actually instantiate `Animal`s; `Animal` is an abstract class and cannot be instantiated. The `Animal` array declaration simply reserves memory for three object references. If you instantiate objects from `Animal` subclasses, you can place references to those objects in the `Animal` array, as Figure 11-12 illustrates. Figure 11-13 shows the output of the `AnimalArrayDemo` application. The array of three references is used to access each appropriate `speak()` method.

```
public class AnimalArrayDemo
{
   public static void main(String[] args)
   {
      Animal[] animalRef = new Animal[3];
      animalRef[0] = new Dog();
      animalRef[1] = new Cow();
      animalRef[2] = new Snake();
      for(int x = 0; x < 3; ++x)
         animalRef[x].speak();
   }
}
```

Figure 11-12 The `AnimalArrayDemo` application

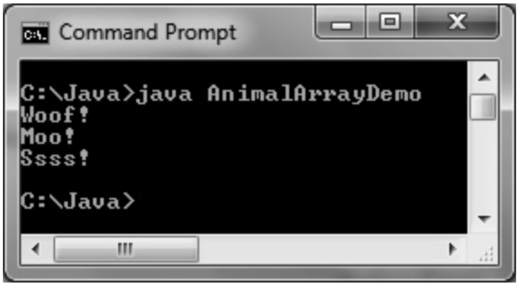

Figure 11-13 Output of the `AnimalArrayDemo` application

In the `AnimalArrayDemo` application in Figure 11-12, a reference to an instance of the `Dog` class is assigned to the first `Animal` reference, and then references to `Cow` and `Snake` objects are assigned to the second and third array elements. After the object references are in the array, you can manipulate them like any other array elements. The application in Figure 11-12 uses a `for` loop and a subscript to get each individual reference to `speak()`.

TWO TRUTHS & A LIE

Creating Arrays of Subclass Objects

1. You can assign a superclass reference to an array of its subclass type.
2. The following statement creates an array of 10 `Table` references:

   ```
   Table[] table = new Table[10];
   ```

3. You can assign subclass objects to an array that is their superclass type.

The false statement is #1. You can assign a subclass reference to an array of its superclass type, but not the other way around.

You Do It

Using Object References

Next, you write an application in which you create an array of `Vehicle` references. Within the application, you assign `Sailboat` objects and `Bicycle` objects to the same array. Then, because the different object types are stored in the same array, you can easily manipulate them by using a `for` loop.

1. Open a new file in your text editor, and then enter the following first few lines of the `VehicleDatabase` program:

   ```
   import javax.swing.*;
   public class VehicleDatabase
   {
      public static void main(String[] args)
      {
   ```

2. Create the following array of five `Vehicle` references and an integer subscript to use with the array:

   ```
   Vehicle[] vehicles = new Vehicle[5];
   int x;
   ```

3. Enter the following `for` loop that prompts you to select whether to enter a sailboat or a bicycle in the array. Based on user input, instantiate the appropriate object type.

(continues)

(continued)

```
for(x = 0; x < vehicles.length; ++x)
{
   String userEntry;
   int vehicleType;
   userEntry = JOptionPane.showInputDialog(null,
      "Please select the type of\n " +
      "vehicle you want to enter: \n1 - Sailboat\n" +
      " 2 - Bicycle");
   vehicleType = Integer.parseInt(userEntry);
   if(vehicleType == 1)
      vehicles[x] = new Sailboat();
   else
      vehicles[x] = new Bicycle();
}
```

4. After entering the information for each vehicle, display the array contents by typing the following code. First create a `StringBuffer` to hold the list of vehicles. Then, in a `for` loop, build an output `String` by repeatedly adding a newline character, a counter, and a vehicle from the array to the `StringBuffer` object. Display the constructed `StringBuffer` in a dialog box. Then type the closing curly braces for the `main()` method and the class:

```
      StringBuffer outString = new StringBuffer();
      for(x = 0; x < vehicles.length; ++x)
      {
         outString.append("\n#" + (x + 1) + " ");
         outString.append(vehicles[x].toString());
      }
      JOptionPane.showMessageDialog(null,
         "Our available Vehicles include:\n" +
         outString);
   }
}
```

5. Save the file as **VehicleDatabase.java**, and then compile it. Run the application, entering five objects of your choice. Figure 11-14 shows typical output after the user has entered data.

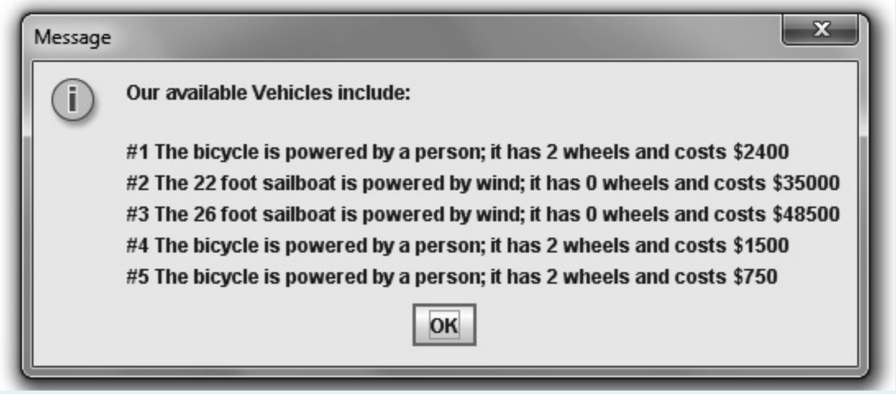

Figure 11-14 Output of the `VehicleDatabase` application

Using the Object Class and Its Methods

Every class in Java is actually a subclass, except one. When you define a class, if you do not explicitly extend another class, your class implicitly is an extension of the Object class. The **Object class** is defined in the java.lang package, which is imported automatically every time you write a program; in other words, the following two class declarations have identical outcomes:

```
public class Animal
{
}
public class Animal extends Object
{
}
```

When you declare a class that does not extend any other class, you always are extending the Object class. The Object class includes methods that descendant classes can use or override as you see fit. Table 11-1 describes the methods built into the Object class; every Object you create has access to these methods.

Method	Description
Object clone()	Creates and returns a copy of this object
boolean equals (Object obj)	Indicates whether some object is equal to the parameter object (this method is described in detail below)
void finalize()	Called by the garbage collector on an object when there are no more references to the object
Class<?> getClass()	Returns the class to which this object belongs at run time
int hashCode()	Returns a hash code value for the object (this method is described briefly below)
void notify()	Wakes up a single thread that is waiting on this object's monitor
void notifyAll()	Wakes up all threads that are waiting on this object's monitor
String toString()	Returns a string representation of the object (this method is described in detail below)
void wait (long timeout)	Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed
void wait (long timeout, int nanos)	Causes the current thread to wait until another thread invokes the notify() or notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed

Table 11-1 Object class methods

Table 11-1 refers to *threads* in several locations. In Chapter 7, you learned about threads in reference to the `StringBuffer` class. Threads of execution are units of processing that are scheduled by an operating system and that can be used to create multiple paths of control during program execution.

Using the `toString()` Method

The `Object` class **`toString()` method** converts an `Object` into a `String` that contains information about the `Object`. Within a class, if you do not create a `toString()` method that overrides the version in the `Object` class, you can use the superclass version of the `toString()` method. For example, examine the `Dog` class originally shown in Figure 11-2 and repeated in Figure 11-15. Notice that it does not contain a `toString()` method and that it extends the `Animal` class.

```
public abstract class Animal
{
   private String name;
   public abstract void speak();
   public String getName()
   {
      return name;
   }
   public void setName(String animalName)
   {
      name = animalName;
   }
}

public class Dog extends Animal
{
   public void speak()
   {
      System.out.println("Woof!");
   }
}

public class DisplayDog
{
   public static void main(String[] args)
   {
      Dog myDog = new Dog();
      String dogString = myDog.toString();
      System.out.println(dogString);
   }
}
```

Figure 11-15 The `Animal` and `Dog` classes and the `DisplayDog` application

Notice that neither the Animal class nor the Dog class in Figure 11-15 defines a toString() method. Yet, when you write the DisplayDog application in Figure 11-15, it uses a toString() method with a Dog object in the shaded statement. The class compiles correctly, converts the Dog object to a String, and produces the output shown in Figure 11-16 because Dog inherits toString() from Object.

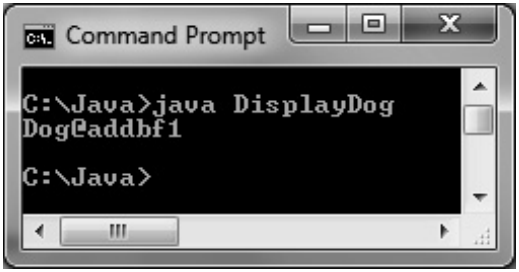

Figure 11-16 Output of the DisplayDog application

The output of the DisplayDog application in Figure 11-16 is not very useful. It consists of the class name of which the object is an instance (Dog), the at sign (@), and a hexadecimal (base 16) number that represents a unique identifier for every object in the current application. The hexadecimal number that is part of the String returned by the toString() method (*addbf1* in Figure 11-16) is an example of a **hash code**—a calculated number used to identify an object. Later in this chapter, you learn about the equals() method, which also uses a hash code.

Instead of using the automatic toString() method with your classes, it is usually more useful to write your own overloaded version that displays some or all of the data field values for the object with which you use it. A good toString() method can be very useful in debugging a program; if you do not understand why a class is behaving as it is, you can display the toString() value and examine its contents. For example, Figure 11-17 shows a BankAccount class that contains a mistake in the shaded line—the BankAccount balance value is set to the account number instead of the balance amount. Of course, if you made such a mistake within one of your own classes, there would be no shading or comment to help you find the mistake. In addition, a useful BankAccount class would be much larger, so the mistake would be more difficult to locate. However, when you ran programs containing BankAccount objects, you would notice that the balances of your BankAccounts were incorrect. To help you discover why, you could create a short application like the TestBankAccount class in Figure 11-18. This application uses the BankAccount class toString() method to display the relevant details of a BankAccount object. The output of the TestBankAccount application appears in Figure 11-19.

```
public class BankAccount
{
   private int acctNum;
   private double balance;
   public BankAccount(int num, double bal)
   {
      acctNum = num;
      balance = num;
   }
   public String toString()
   {
      String info = "BankAccount acctNum = " + acctNum +
         "   Balance = $" + balance;
      return info;
   }
}
```

Don't Do It
The `bal` parameter should be assigned to `balance`, not the `num` parameter.

Figure 11-17 The `BankAccount` class

```
public class TestBankAccount
{
   public static void main(String[] args)
   {
      BankAccount myAccount = new BankAccount(123, 4567.89);
      System.out.println(myAccount.toString());
   }
}
```

Figure 11-18 The `TestBankAccount` application

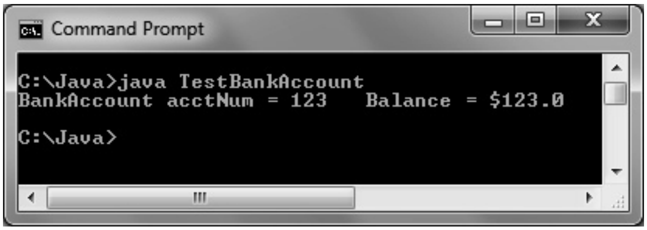

Figure 11-19 Output of the `TestBankAccount` application

From the output in Figure 11-19, you can see that the account number and balance have the same value, and this knowledge might help you to pin down the location of the incorrect statement in the `BankAccount` class. Of course, you do not have to use a method named `toString()` to discover a `BankAccount`'s attributes. If the class had methods such as getAcctNum() and getBalance(), you could use them to create a similar application. The advantage of creating a toString() method for your classes is that toString() is Java's conventional name for a method that converts an object's relevant details into String format. Because toString() originates in the Object class, you can be assured that toString() compiles with any object whose details you want to see, even if the method has not been rewritten for the subclass in question. In addition, as you write your own applications and use classes written by others, you can hope that those programmers have overridden toString() to provide useful information. You don't have to search documentation to discover a useful method—instead you can rely on the likely usefulness of toString(). In Chapter 7, you learned that you can use the toString() method to convert any object to a String. Now you understand why this works—the String class overloads the Object class toString() method.

Using the equals() Method

The Object class also contains an **equals() method** that takes a single argument, which must be the same type as the type of the invoking object, as in the following example:

```
if(someObject.equals(someOtherObjectOfTheSameType))
   System.out.println("The objects are equal");
```

> Other classes, such as the String class, also have their own equals() methods that overload the Object class method. You first used the equals() method to compare String objects in Chapter 7. Two String objects are considered equal only if their String contents are identical.

The Object class equals() method returns a boolean value indicating whether the objects are equal. This equals() method considers two objects of the same class to be equal only if they have the same hash code; in other words, they are equal only if one is a reference to the other. For example, two BankAccount objects named myAccount and yourAccount are not automatically equal, even if they have the same account numbers and balances; they are equal only if they have the same memory address. If you want to consider two objects to be equal only when one is a reference to the other, you can use the built-in Object class equals() method. However, if you want to consider objects to be equal based on their contents, you must write your own equals() method for your classes.

> Java's Object class contains a public method named hashCode() that returns an integer representing the hash code. (Discovering this number is of little use to you. The default hash code is the internal JVM memory address of the object.) However, whenever you override the equals() method in a professional class, you generally want to override the hashCode() method as well, because equal objects should have equal hash codes, particularly if the objects will be used in hash-based methods. See the documentation at the Java Web site for more details.

The application shown in Figure 11-20 instantiates two `BankAccount` objects using the `BankAccount` class in Figure 11-17. The `BankAccount` class does not include its own `equals()` method, so it does not override the `Object equals()` method. Thus, the application in Figure 11-20 produces the output in Figure 11-21. Even though the two `BankAccount` objects have the same account numbers and balances, the `BankAccounts` are not considered equal because they do not have the same memory address.

```
public class CompareAccounts
{
   public static void main(String[] args)
   {
      BankAccount acct1 = new BankAccount(1234, 500.00);
      BankAccount acct2 = new BankAccount(1234, 500.00);
      if(acct1.equals(acct2))
         System.out.println("Accounts are equal");
      else
         System.out.println("Accounts are not equal");
   }
}
```

Figure 11-20 The `CompareAccounts` application

Figure 11-21 Output of the `CompareAccounts` application

If your intention is that within applications, two `BankAccount` objects with the same account number and balance are equal, and you want to use the `equals()` method to make the comparison, you must write your own `equals()` method within the `BankAccount` class. For example, Figure 11-22 shows a new version of the `BankAccount` class containing a shaded `equals()` method. When you reexecute the `CompareAccounts` application in Figure 11-20, the result appears as in Figure 11-23.

```
public class BankAccount
{
   private int acctNum;
   private double balance;
   public BankAccount(int num, double bal)
   {
      acctNum = num;
      balance = bal;
   }
   public String toString()
   {
      String info = "BankAccount acctNum = " + acctNum +
        "   Balance = $" + balance;
      return info;
   }
   public boolean equals(BankAccount secondAcct)
   {
      boolean result;
      if(acctNum == secondAcct.acctNum && balance == secondAcct.balance)
        result = true;
      else
        result = false;
      return result;
   }
}
```

Figure 11-22 The BankAccount class containing its own equals() method

Figure 11-23 Output of the CompareAccounts application after adding an overloaded equals() method to the BankAccount class

The two BankAccount objects described in the output in Figure 11-23 are equal because their account numbers and balances match. Because the equals() method in Figure 11-22 is part of the BankAccount class, and because equals() is a nonstatic method, the object that calls the method is held by the this reference within the method. That is, in the application in Figure 11-22, acct1 becomes the this reference in the equals() method, so the fields acctNum and balance refer to acct1 object values. In the CompareAccounts application, acct2 is the argument to the equals() method, so within the equals() method, acct2 becomes secondAcct, and secondAcct.acctNum and secondAcct.balance refer to acct2's values.

Your organization might consider two `BankAccount` objects equal if their account numbers match, disregarding their balances. If so, you simply change the `if` clause in the `equals()` method. Or, you might decide accounts are equal based on some other criteria. You can implement the `equals()` method in any way that suits your needs. When you want to compare the contents of two objects, you do not have to overload the `Object` class `equals()` method. Instead, you could write a method with a unique name, such as `areTheyEqual()` or `areContentsSame()`. However, as with the `toString()` method, users of your classes will appreciate that you use the expected, usual, and conventional identifiers for your methods.

If you change a class (such as changing `BankAccount` by adding a new method), not only must you recompile the class, you must also recompile any client applications (such as `CompareAccounts`) so the newly updated class can be relinked to the application and so the clients include the new features of the altered class. If you execute the `CompareAccounts` application but do not recompile `BankAccount`, the application continues to use the previously compiled version of the class.

Watch the video *The Object Class*.

TWO TRUTHS & A LIE

Using the `Object` Class and Its Methods

1. When you define a class, if you do not explicitly extend another class, your class is an extension of the `Object` class.
2. The `Object` class is defined in the `java.lang` package that is imported automatically every time you write a program.
3. The `Object` class `toString()` and `equals()` methods are abstract.

The false statement is #3. The `toString()` and `equals()` methods are not abstract—you are not required to override them in a subclass.

Using Inheritance to Achieve Good Software Design

When an automobile company designs a new car model, the company does not build every component of the new car from scratch. The company might design a new feature completely from scratch; for example, at some point someone designed the first air bag. However, many of a new car's features are simply modifications of existing features. The manufacturer might create a larger gas tank or more comfortable seats, but even these new features still possess many properties of their predecessors in the older models. Most features of new car models are not even modified; instead, existing components, such as air filters and windshield wipers, are included on the new model without any changes.

Similarly, you can create powerful computer programs more easily if many of their components are used either "as is" or with slight modifications. Inheritance does not give you the ability to write programs that you could not write otherwise. If Java did not allow you to extend classes, you *could* create every part of a program from scratch. Inheritance simply makes your job easier. Professional programmers constantly create new class libraries for use with Java programs. Having these classes available makes programming large systems more manageable.

You have already used many "as is" classes, such as `System` and `String`. In these cases, your programs were easier to write than if you had to write these classes yourself. Now that you have learned about inheritance, you have gained the ability to modify existing classes. When you create a useful, extendable superclass, you and other future programmers gain several advantages:

- Subclass creators save development time because much of the code needed for the class has already been written.
- Subclass creators save testing time because the superclass code has already been tested and probably used in a variety of situations. In other words, the superclass code is reliable.
- Programmers who create or use new subclasses already understand how the superclass works, so the time it takes to learn the new class features is reduced.
- When you create a new subclass in Java, neither the superclass source code nor the superclass bytecode is changed. The superclass maintains its integrity.

When you consider classes, you must think about the commonalities among them; then you can create superclasses from which to inherit. You might be rewarded professionally when you see your own superclasses extended by others in the future.

TWO TRUTHS & A LIE

Using Inheritance to Achieve Good Software Design

1. If object-oriented programs did not support inheritance, programs could still be written, but they would be harder to write.
2. When you create a useful, extendable superclass, you save development and testing time.
3. When you create a new subclass in Java, you must remember to revise and recompile the superclass code.

The false statement is #3. When you create a new subclass in Java, neither the superclass source code nor the superclass bytecode is changed.

Creating and Using Interfaces

Some object-oriented programming languages, such as C++, allow a subclass to inherit from more than one parent class. For example, you might create an `InsuredItem` class that contains data fields pertaining to each possession for which you have insurance. Data fields might include the name of the item, its value, the insurance policy type, and so on. You might also create an `Automobile` class that contains data fields such as vehicle identification number, make, model, and year. When you create an `InsuredAutomobile` class for a car rental agency, you might want to include `InsuredItem` information and methods, as well as `Automobile` information and methods. It would be convenient to inherit from both the `InsuredItem` and `Automobile` classes. The capability to inherit from more than one class is called **multiple inheritance**.

Many programmers consider multiple inheritance to be a difficult concept, and when inexperienced programmers use it they encounter many problems. Programmers have to deal with the possibility that variables and methods in the parent classes might have identical names, which creates conflict when the child class uses one of the names. Also, you have already learned that a child class constructor must call its parent class constructor. When there are two or more parents, this task becomes more complicated—to which class should `super()` refer when a child class has multiple parents? For all of these reasons, multiple inheritance is prohibited in Java. A class can inherit from a superclass that has inherited from another superclass—this represents single inheritance with multiple generations. However, Java does not allow a class to inherit directly from two or more parents.

Java, however, does provide an alternative to multiple inheritance—an interface. An **interface** looks much like a class, except that all of its methods (if any) are implicitly `public` and `abstract`, and all of its data items (if any) are implicitly `public`, `static`, and `final`. An interface is a description of what a class does but not how it is done; it declares method headers but not the instructions within those methods. When you create a class that uses an interface, you include the keyword `implements` and the interface name in the class header. This notation requires class objects to include code for every method in the interface that has been implemented. Whereas using `extends` allows a subclass to use nonprivate, nonoverridden members of its parent's class, `implements` requires the subclass to implement its own version of each method.

> In English, an interface is a device or a system that unrelated entities use to interact. Within Java, an interface provides a way for unrelated objects to interact with each other. An interface is analogous to a protocol, which is an agreed-on behavior. In some respects, an `Automobile` can behave like an `InsuredItem`, and so can a `House`, a `TelevisionSet`, and a `JewelryPiece`.

As an example, recall the `Animal` and `Dog` classes from earlier in this chapter. Figure 11-24 shows these classes, with `Dog` inheriting from `Animal`.

```
public abstract class Animal
{
   private String name;
   public abstract void speak();
   public String getName()
   {
      return name;
   }
   public void setName(String animalName)
   {
      name = animalName;
   }
}
public class Dog extends Animal
{
   public void speak()
   {
      System.out.println("Woof!");
   }
}
```

Figure 11-24 The `Animal` and `Dog` classes

You can create a `Worker` interface, as shown in Figure 11-25. For simplicity, this example gives the `Worker` interface a single method named `work()`. When any class implements `Worker`, it must either include a `work()` method or the new class must be declared abstract, and then its descendants must implement the method.

```
public interface Worker
{
   public void work();
}
```

Figure 11-25 The `Worker` interface

The `WorkingDog` class in Figure 11-26 extends `Dog` and implements `Worker`. A `WorkingDog` contains a data field that a “regular” `Dog` does not—an integer that holds hours of training received. The `WorkingDog` class also contains get and set methods for this field. Because the `WorkingDog` class implements the `Worker` interface, it also must contain a `work()` method that calls the `Dog speak()` method, and then produces two more lines of output—a statement about working and the number of training hours.

```
public class WorkingDog extends Dog implements Worker
{
   private int hoursOfTraining;
   public void setHoursOfTraining(int hrs)
   {
      hoursOfTraining = hrs;
   }
   public int getHoursOfTraining()
   {
      return hoursOfTraining;
   }
   public void work()
   {
      speak();
      System.out.println("I am a dog who works");
      System.out.println("I have " + hoursOfTraining +
         " hours of professional training!");
   }
}
```

Figure 11-26 The `WorkingDog` class

> As you know from other classes you have seen, a class can extend another class without implementing any interfaces. A class can also implement an interface even though it does not extend any other class. When a class both extends and implements, like the `WorkingDog` class, by convention the `implements` clause follows the `extends` clause in the class header.

The `DemoWorkingDogs` application in Figure 11-27 instantiates two `WorkingDog` objects. Each object can use the following methods:

- The `setName()` and `getName()` methods that `WorkingDog` inherits from the `Animal` class
- The `speak()` method that `WorkingDog` inherits from the `Dog` class
- The `setHoursOfTraining()` and `getHoursOfTraining()` methods contained within the `WorkingDog` class
- The `work()` method that the `WorkingDog` class was required to contain when it used the phrase `implements Worker`; the `work()` method also calls the `speak()` method contained in the `Dog` class.

```
public class DemoWorkingDogs
{
   public static void main(String[] args)
   {
      WorkingDog aSheepHerder = new WorkingDog();
      WorkingDog aSeeingEyeDog = new WorkingDog();
      aSheepHerder.setName("Simon, the Border Collie");
      aSeeingEyeDog.setName("Sophie, the German Shepherd");
      aSheepHerder.setHoursOfTraining(40);
      aSeeingEyeDog.setHoursOfTraining(300);

      System.out.println(aSheepHerder.getName() + " says ");
      aSheepHerder.speak();
      aSheepHerder.work();
      System.out.println(); // outputs a blank line for readability

      System.out.println(aSeeingEyeDog.getName() + " says ");
      aSeeingEyeDog.speak();
      aSeeingEyeDog.work();
   }
}
```

Figure 11-27 The `DemoWorkingDogs` application

Figure 11-28 shows the output when the `DemoWorkingDogs` application executes. Each animal is introduced, then it "speaks," and then each animal "works," which includes speaking a second time. Each `Animal` can execute the `speak()` method implemented in its own class, and each can execute the `work()` method contained in the implemented interface. Of course, the `WorkingDog` class was not required to implement the `Worker` interface; instead, it could have just contained a `work()` method that all `WorkingDog` objects could use. If `WorkingDog` was the only class that would ever use `work()`, such an approach would probably be the best course of action. However, if many classes will be `Workers`—that is, require a `work()` method—they all can implement `work()`. If you are already familiar with the `Worker` interface and its method, when you glance at a class definition for a `WorkingHorse`, `WorkingBird`, or `Employee` and see that it implements `Worker`, you do not have to guess at the name of the method that shows the work the class objects perform. Notice that when a class implements an interface, it represents a situation similar to inheritance. Just as a `WorkingDog` "is a" `Dog` and "is an" `Animal`, so too it "is a" `Worker`.

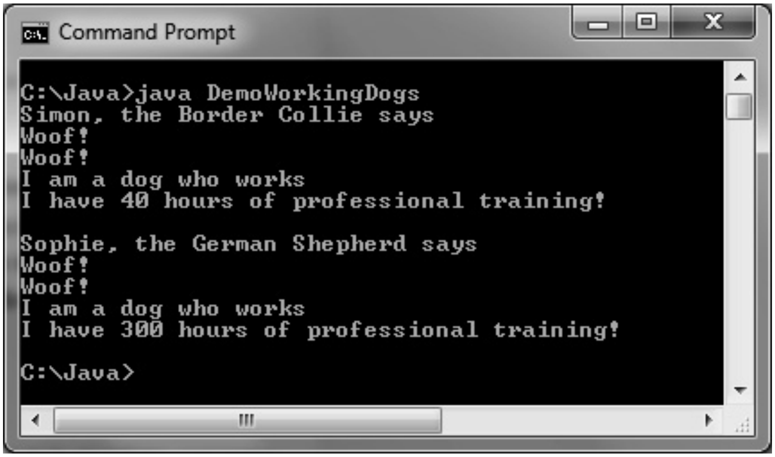

Figure 11-28 Output of the `DemoWorkingDogs` application

You can compare abstract classes and interfaces as follows:

- Abstract classes and interfaces are similar in that you cannot instantiate concrete objects from either one.
- Abstract classes differ from interfaces because abstract classes can contain nonabstract methods, but all methods within an interface must be abstract.
- A class can inherit from only one abstract superclass, but it can implement any number of interfaces.

Beginning programmers sometimes find it difficult to decide when to create an abstract superclass and when to create an interface. Remember, you create an abstract class when you want to provide data or methods that subclasses can inherit, but at the same time these subclasses maintain the ability to override the inherited methods.

Suppose that you create a `CardGame` class to use as a base class for different card games. It contains four methods named `shuffle()`, `deal()`, `displayRules()`, and `keepScore()`. The `shuffle()` method works the same way for every `CardGame`, so you write the statements for `shuffle()` within the superclass, and any `CardGame` objects you create later inherit `shuffle()`. The methods `deal()`, `displayRules()`, and `keepScore()` operate differently for every subclass (for example, for `TwoPlayerCardGames`, `FourPlayerCardGames`, `BettingCardGames`, and so on), so you force `CardGame` children to contain instructions for those methods by leaving them empty in the superclass. The `CardGame` class, therefore, should be an abstract superclass. When you write classes that extend the `CardGame` parent class, you inherit the `shuffle()` method, and write code within the `deal()`, `displayRules()`, and `keepScore()` methods for each specific child.

You create an interface when you know what actions you want to include, but you also want every user to separately define the behavior that must occur when the method executes. Suppose that you create a `MusicalInstrument` class to use as a base for different musical instrument object classes such as `Piano`, `Violin`, and `Drum`. The parent `MusicalInstrument` class contains methods such as `playNote()` and `outputSound()` that apply to every instrument, but you want to implement these methods differently for each type of instrument. By making `MusicalInstrument` an interface, you require every nonabstract subclass to code all the methods.

An interface specifies only the messages to which an object can respond; an abstract class can include methods that contain the actual behavior the object performs when those messages are received.

You also create an interface when you want a class to implement behavior from more than one parent. For example, suppose that you want to create an interactive `NameThatInstrument` card game in which you play an instrument sound from the computer speaker, and ask players to identify the instrument they hear by clicking one of several cards that display instrument images. This game class could not extend from two classes, but it could extend from `CardGame` and implement `MusicalInstrument`.

When you create a class and use the `implements` clause to implement an interface but fail to code one of the interface's methods, the compiler error generated indicates that you must declare your class to be `abstract`. If you want your class to be used only for extending, you can make it `abstract`. However, if your intention is to create a class from which you can instantiate objects, do not make it `abstract`. Instead, find out which methods from the interface you have failed to implement within your class and code those methods.

Java has many built-in interfaces with names such as `Serializable`, `Runnable`, `Externalizable`, and `Cloneable`. See the documentation at the Java Web site for more details.

Creating Interfaces to Store Related Constants

Interfaces can contain data fields, but they must be `public`, `static`, and `final`. It makes sense that interface data must not be `private` because interface methods cannot contain method bodies; without public method bodies, you have no way to retrieve `private` data. It also makes sense that the data fields in an interface are `static` because you cannot create interface objects. Finally, it makes sense that interface data fields are `final` because, without methods containing bodies, you have no way, other than at declaration, to set the data fields' values, and you have no way to change them.

Your purpose in creating an interface containing constants is to provide a set of data that a number of classes can use without having to redeclare the values. For example, the interface class in Figure 11-29 provides a number of constants for a pizzeria. Any class written for the pizzeria can implement this interface and use the permanent values. Figure 11-30 shows an example of one application that uses each value, and Figure 11-31 shows the output. The application in Figure 11-30 only needs a declaration for the current special price; all the constants, such as the name of the pizzeria, are retrieved from the interface.

```
public interface PizzaConstants
{
   public static final int SMALL_DIAMETER = 12;
   public static final int LARGE_DIAMETER = 16;
   public static final double TAX_RATE = 0.07;
   public static final String COMPANY = "Antonio's Pizzeria";
}
```

Figure 11-29 The `PizzaConstants` interface

```
public class PizzaDemo implements PizzaConstants
{
   public static void main(String[] args)
   {
      double specialPrice = 11.25;
      System.out.println("Welcome to " + COMPANY);
      System.out.println("We are having a special offer:\na " +
         SMALL_DIAMETER + " inch pizza with four toppings\nor a " +
         LARGE_DIAMETER +
         " inch pizza with one topping\nfor only $" + specialPrice);
      System.out.println("With tax, that is only $" +
         (specialPrice + specialPrice * TAX_RATE));
   }
}
```

Figure 11-30 The `PizzaDemo` application

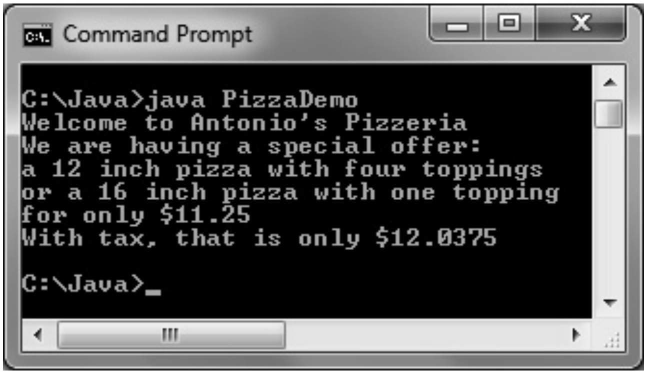

Figure 11-31 Output of the `PizzaDemo` application

Watch the video *Interfaces*.

TWO TRUTHS & A LIE

Creating and Using Interfaces

1. Java's capability to inherit from more than one class is called multiple inheritance.
2. All of the methods in an interface are implicitly `public` and `abstract`, and all of its data items (if any) are implicitly `public`, `static`, and `final`.
3. When a class inherits from another, the child class can use the nonprivate, nonoverridden members of its parent's class, but when a class uses an interface, it must implement its own version of each method.

The false statement is #1. The ability to inherit from more than one class is called multiple inheritance, but Java does not have that ability.

You Do It

Using an Interface

In this section, you create an `Insured` interface for use with classes that represent objects that can be insured. For example, you might use this interface with classes such as `Jewelry` or `House`. Also in this section, you extend `Vehicle` to create an `InsuredCar` class that implements the `Insured` interface, and then you write a short program that instantiates an `InsuredCar` object.

1. Open a new file in your text editor, and type the following `Insured` interface. A concrete class that implements `Insured` will be required to contain `setCoverage()` and `getCoverage()` methods.

```
public interface Insured
{
   public void setCoverage();
   public int getCoverage();
}
```

2. Save the file as **Insured.java** and compile it.
3. Open a new file in your text editor, and start the `InsuredCar` class that extends `Vehicle` and implements `Insured`:

```
import javax.swing.*;
public class InsuredCar extends Vehicle implements Insured
{
```

(continues)

(continued)

4. Add a variable to hold the amount covered by the insurance:

```
private int coverage;
```

5. Add a constructor that calls the `Vehicle` superclass constructor, passing arguments for the `InsuredCar`'s power source and number of wheels.

```
public InsuredCar()
{
   super("gas", 4);
   setCoverage();
}
```

6. Implement the `setPrice()` method required by the `Vehicle` class. The method accepts the car's price from the user and enforces a maximum value of $60,000.

```
public void setPrice()
{
   String entry;
   final int MAX = 60000;
   entry = JOptionPane.showInputDialog
     (null, "Enter car price ");
   price = Integer.parseInt(entry);
   if(price > MAX)
      price = MAX;
}
```

7. Implement the `setCoverage()` and `getCoverage()` methods required by the `Insured` class. The `setCoverage()` method sets the coverage value for an insured car to 90 percent of the car's price:

```
public void setCoverage()
{
   coverage = (int)(price * 0.9);
}
public int getCoverage()
{
   return coverage;
}
```

8. Create a `toString()` method, followed by a closing brace for the class:

```
   public String toString()
   {
      return("The insured car is powered by " + getPowerSource() +
         "; it has " + getWheels() + " wheels, costs $" +
         getPrice() + " and is insured for $" + getCoverage());
   }
}
```

(continues)

(continued)

9. Save the file as **InsuredCar.java** and compile it.
10. Create a demonstration program that instantiates an `InsuredCar` object and displays its values as follows:

```
import javax.swing.*;
public class InsuredCarDemo
{
   public static void main(String[] args)
   {
      InsuredCar myCar = new InsuredCar();
      JOptionPane.showMessageDialog(null,
        myCar.toString());
   }
}
```

11. Save the file as **InsuredCarDemo.java**. Compile and execute it. You will be prompted to enter the car's price. Figure 11-32 shows the output during a typical execution.

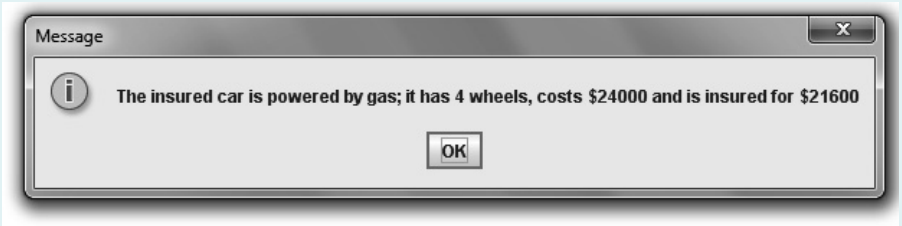

Figure 11-32 Output of the `InsuredCarDemo` program

Creating and Using Packages

Throughout most of this book, you have imported packages into your programs. As you learned in Chapter 4, a package is a named collection of classes; for example, the `java.lang` package contains fundamental classes and is automatically imported into every program you write. You also have created classes into which you explicitly imported optional packages such as `java.util` and `javax.swing`. When you create classes, you can place them in packages so that you or other programmers can easily import your related classes into new programs. Placing classes in packages for other programmers increases the classes' reusability. When you create a number of classes that inherit from each other, as well as multiple interfaces that you want to implement with these classes, you often will find it convenient to place these related classes in a package.

Creating packages encourages others to reuse software because it makes it convenient to import many related classes at once. In Chapter 3, you learned that if you do not use one of the three access specifiers `public`, `private`, or `protected` for a class, then it has default access, which means that the unmodified class is accessible to any other class in the same package.

When you create professional classes for others to use, you most often do not want to provide the users with your source code in the files that have .java extensions. You expend significant effort developing workable code for your programs, and you do not want other programmers to be able to copy your programs, make minor changes, and market the new product themselves. Rather, you want to provide users with the compiled files that have .class extensions. These are the files the user needs to run the program you have developed. Likewise, when other programmers use the classes you have developed, they need only the completed compiled code to import into their programs. The .class files are the files you place in a package so other programmers can import them.

In the Java programming language, a package or class library is often delivered to users as a **Java ARchive (JAR) file**. JAR files compress the data they store, which reduces the size of archived class files. The JAR format is based on the popular Zip file format.

If you do not specify a package for a class, it is placed in an unnamed **default package**. A class that will be placed in a nondefault package for others to use must be `public`. If a class is not `public`, it can be used only by other classes within the same package. To place a class in a package, you include a `package` declaration at the beginning of the source code file that indicates the folder into which the compiled code will be placed. When a file contains a `package` declaration, it must be the first statement in the file (excluding comments). If there are import declarations, they follow the `package` declaration. Within the file, the `package` statement must appear outside the class definition. The `package` statement, `import` statements, and comments are the only statements that appear outside class definitions in Java program files.

For example, the following statement indicates that the compiled file should be placed in a folder named com.course.animals:

```
package com.course.animals;
```

That is, the compiled file should be stored in the animals subfolder inside the course subfolder inside the com subfolder (or com\course\animals). The pathname can contain as many levels as you want.

When you compile a file that you want to place in a package, you can copy or move the compiled .class file to the appropriate folder. Alternatively, you can use a compiler option with the `javac` command. The `-d` (for *directory*) option indicates that you want to place the generated .class file in a folder. For example, the following command indicates that the compiled Animal.java file should be placed in the directory indicated by the `import` statement within the Animal.java file:

```
javac -d . Animal.java
```

The dot (period) in the compiler command indicates that the path shown in the `package` statement in the file should be created within the current directory.

If the `Animal` class file contains the statement `package com.course.animals;`, the Animal. class file is placed in C:\com\course\animals. If any of these subfolders do not exist, Java creates them. Similarly, if you package the compiled files for Dog.java, Cow.java, and so on, future programs need only use the following statements to be able to use all the related classes:

```
import com.course.animals.Dog;
import com.course.animals.Cow;
```

Because Java is used extensively on the Internet, it is important to give every package a unique name. The creators of Java have defined a package-naming convention that uses your Internet domain name in reverse order. For example, if your domain name is course.com, you begin all of your package names with com.course. Subsequently, you organize your packages into reasonable subfolders.

Creating packages using Java's naming convention helps avoid naming conflicts—different programmers might create classes with the same name, but they are contained in different packages. Class-naming conflicts are sometimes called **collisions**. Because of packages, you can create a class without worrying that its name already exists in Java or in packages distributed by another organization. For example, if your domain name is course.com, then you might want to create a class named `Scanner` and place it in a package named `com.course.input`. The fully qualified name of your `Scanner` class is `com.course.input.Scanner`, and the fully qualified name of the built-in `Scanner` class is `java.util.Scanner`.

TWO TRUTHS & A LIE

Creating and Using Packages

1. Typically, you place .class files in a package so other programmers can import them into their programs.
2. A class that will be placed in a package for others to use must be `protected` so that others cannot read your source code.
3. Java's creators have defined a package-naming convention in which you use your Internet domain name in reverse order.

The false statement is #2. A class that will be placed in a package for others to use must be `public`. If a class is not `public`, it can be used only by other classes within the same package. To prevent others from viewing your source code, you place compiled .class files in distributed packages.

You Do It

Creating a Package

Next, you place the `Vehicle` family of classes into a package. Assume you work for an organization that sponsors a Web site at *vehicleswesell.com*, so you name the package `com.vehicleswesell`. First, you must create a folder named VehiclePackage in which to store your project. You can use any technique that is familiar to you. For example, in Windows, you can double-click Computer, navigate to the device or folder where you want to store the package, right-click, click New, click Folder, replace "New Folder" with the new folder name (VehiclePackage), and press Enter. Alternatively, from the command prompt, you can navigate to the drive and folder where you want the new folder to reside by using the following commands:

- If the command prompt does not indicate the storage device you want, type the name of the drive and a colon to change the command prompt to a different device. For example, to change the command prompt to the F drive on your system, type `F:`.
- If the directory is not the one you want, type `cd\` to navigate to the root directory. The `cd` command stands for "change directory," and the backslash indicates the root directory. Then type `cd` followed by the name of the subdirectory you want. You can repeat this command as many times as necessary to get to the correct subdirectory if it resides many levels down the directory hierarchy.

Next, you can place three classes into a package.

1. Open the **Vehicle.java** file in your text editor.
2. As the first line in the file, insert the following statement:

   ```
   package com.vehicleswesell.vehicle;
   ```

3. Save the file as **Vehicle.java** in the **VehiclePackage** folder.
4. At the command line, at the prompt for the VehiclePackage folder, compile the file using the following command:

   ```
   javac -d . Vehicle.java
   ```

   Be certain that you type a space between each element in the command, including surrounding the dot. Java creates a folder named com\vehicleswesell\vehicle within the directory from which you compiled the program, and the compiled **Vehicle.class** file is placed in this folder.

(continues)

(continued)

If you see a list of compile options when you try to compile the file, you did not type the spaces within the command correctly. Repeat Step 4 to compile again.

The development tool GRASP generates software visualizations to make programs easier to understand. A copy of this tool is included with your downloadable student files. If you are using jGRASP to compile your Java programs, you also can use it to set compiler options. To set a compiler option to –d, do the following:

- Open a jGRASP project workspace. Click the **Settings** menu, point to **Compiler Settings**, and then click **Workspace**. The Settings for workspace dialog box appears.
- Under the FLAGS or ARGS section of the dialog box, click the dot inside the square next to the Compile option and enter the compiler option (**-d**). Then click the **Apply** button.
- Click the **OK** button to close the dialog box, and then compile your program as usual.

5. Examine the folders on your storage device, using any operating system program with which you are familiar. For example, if you are compiling at the DOS command line, type **dir** at the command-line prompt to view the folders stored in the current directory. You can see that Java created a folder named com. (If you have too many files and folders stored, it might be difficult to locate the com folder. If so, type **dir com*.*** to see all files and folders in the current folder that begin with "com".) Figure 11-33 shows the command to compile the `Vehicle` class and the results of the `dir` command, including the com folder.

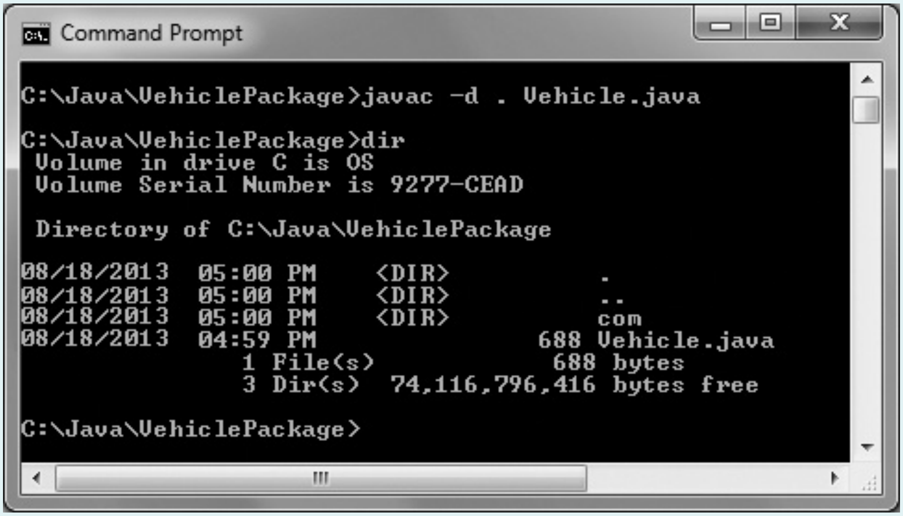

Figure 11-33 Compiling the Vehicle.java file in a package and viewing the results

(continues)

(continued)

Alternatively, to view the created folders in a Windows operating system, you can double-click **Computer**, double-click the appropriate storage device, and locate the com folder. Within the com folder is a vehicleswesell folder, and within vehicleswesell is a vehicle folder. The **Vehicle.class** file is within the vehicle subfolder and not in the same folder as the .java source file where it ordinarily would be placed.

If you cannot find the com folder on your storage device, you probably are not looking in the same folder where you compiled the class. Repeat Steps 4 and 5, but be certain that you first change to the command prompt for the directory where your source code file resides.

6. You could now delete the copy of the **Vehicle.java** file from the VehiclePackage folder (although you most likely want to retain a copy elsewhere). There is no further need for this source file in the folder you will distribute to users because the compiled .class file is stored in the com\vehicleswesell\vehicle folder. Don't delete the copy of your code from its original storage location; you might want to retain a copy of the code for modification later.

7. Open the **Sailboat.java** file in your text editor. For the first line in the file, insert the following statement:

```
package com.vehicleswesell.vehicle;
```

8. Save the file in the same directory as you saved **Vehicle.java**. At the command line, compile the file using the following command:

```
javac -d . Sailboat.java
```

   Then you can delete the **Sailboat.java** source file from the VehiclePackage folder (not from its original location—you want to retain a copy of your original code).

9. Repeat Steps 7 and 8 to perform the same operations using the **Bicycle.java** file.

10. Open the **VehicleDatabase.java** file in your text editor. Insert the following statements at the top of the file:

```
import com.vehicleswesell.vehicle.Vehicle;
import com.vehicleswesell.vehicle.Sailboat;
import com.vehicleswesell.vehicle.Bicycle;
```

11. Save the file as **VehiclePackage\VehicleDatabase.java**. Compile the file, and then run the program. The program's output should be the same as it was before you added the import statements. Placing the `Vehicle`-related class files in a package is not required for the `VehicleDatabase` program to execute correctly; you ran it in exactly the same manner before you learned about creating packages.

    Placing classes in packages gives you the ability to more easily isolate and distribute files.

Don't Do It

- Don't write a body for an abstract method.
- Don't forget to end an abstract method header with a semicolon.
- Don't forget to override any abstract methods in any subclasses you derive.
- Don't mistakenly overload an abstract method instead of overriding it; the subclass method must have the same parameter list as the parent's abstract method.
- Don't try to instantiate an abstract class object.
- Don't forget to override all the methods in an interface that you implement.
- When you create your own packages, don't try to use the wildcard format to import multiple classes. This technique works only with built-in packages.

Key Terms

Concrete classes are nonabstract classes from which objects can be instantiated.

An **abstract class** is one from which you cannot create any concrete objects but from which you can inherit.

Virtual classes is the name given to abstract classes in other programming languages, such as C++.

An **abstract method** is declared with the keyword `abstract`. It is a method with no body—no curly braces and no method statements—just a return type, a method name, an optional argument list, and a semicolon. You are required to code a subclass method to override the empty superclass method that is inherited.

Dynamic method binding is the ability of an application to select the correct method during program execution.

Late method binding is another term for dynamic method binding.

Static or **fixed method binding** is the opposite of dynamic method binding; it occurs when a method is selected when the program compiles rather than while it is running.

Ad-hoc polymorphism occurs when a single method name can be used with a variety of data types because various implementations exist; it is another name for method overloading.

Pure polymorphism or **inclusion polymorphism** occurs when a single method implementation can be used with a variety of related objects because they are objects of subclasses of the parameter type.

The **`Object` class** is defined in the `java.lang` package that is imported automatically every time you write a program; it includes methods that you can use or override. When you define a class, if you do not explicitly extend another class, your class is an extension of the `Object` class.

The `Object` class **`toString()` method** converts an `Object` into a `String` that contains information about the `Object`.

A **hash code** is a calculated number used to identify an object.

The `Object` class **`equals()` method** takes a single argument, which must be the same type as the type of the invoking object, and returns a Boolean value indicating whether two object references are equal.

Multiple inheritance is the capability to inherit from more than one class.

An **interface** looks much like a class, except that all of its methods must be abstract and all of its data (if any) must be `static final`; it declares method headers but not the instructions within those methods.

A **Java ARchive (JAR) file** compresses the stored data.

A **default package** is the unnamed one in which a class is placed if you do not specify a package for the class.

Collision is a term that describes a class-naming conflict.

Chapter Summary

- A class that you create only to extend from, but not to instantiate from, is an abstract class. Usually, abstract classes contain one or more abstract methods—methods with no method statements. You must code a subclass method to override any inherited abstract superclass method.
- When you create a superclass and one or more subclasses, each object of the subclass "is a" superclass object, so you can convert subclass objects to superclass objects. The ability of a program to select the correct method during execution based on argument type is known as dynamic method binding. You can create an array of superclass object references but store subclass instances in it.
- Every class in Java is an extension of the `Object` class, whether or not you explicitly extend it. Every class inherits several methods from `Object`, including `toString()`, which converts an `Object` into a `String`, and `equals()`, which returns a `boolean` value indicating whether one object is a reference to another. You can override these methods to make them more useful for your classes.

- When you create a useful, extendable superclass, you save development time because much of the code needed for the class has already been written. In addition, you save testing time and, because the superclass code is reliable, you reduce the time it takes to learn the new class features. You also maintain superclass integrity.
- An interface is similar to a class, but all of its methods are implicitly `public` and `abstract`, and all of its data (if any) is implicitly `public`, `static`, and `final`. When you create a class that uses an interface, you include the keyword `implements` and the interface name in the class header. This notation serves to require class objects to include code for all the methods in the interface.
- Abstract classes and interfaces are similar in that you cannot instantiate concrete objects from either. Abstract classes differ from interfaces because abstract classes can contain nonabstract methods, but all methods within an interface must be abstract. A class can inherit from only one abstract superclass, but it can implement any number of interfaces.
- You can place classes in packages so you or other programmers can easily import related classes into new classes. The convention for naming packages uses Internet domain names in reverse order to ensure that your package names do not conflict with those of any other Internet users.

Review Questions

1. Parent classes are ____________ than their child classes.
   a. less specific
   b. more specific
   c. easier to understand
   d. more cryptic
2. Abstract classes differ from other classes in that you ____________.
   a. must not code any methods within them
   b. must instantiate objects from them
   c. cannot instantiate objects from them
   d. cannot have data fields within them
3. Abstract classes can contain ____________.
   a. abstract methods
   b. nonabstract methods
   c. both of the above
   d. none of the above
4. An abstract class `Product` has two subclasses, `Perishable` and `NonPerishable`. None of the constructors for these classes requires any arguments. Which of the following statements is legal?
   a. `Product myProduct = new Product();`
   b. `Perishable myProduct = new Product();`
   c. `NonPerishable myProduct = new NonPerishable();`
   d. none of the above

5. An abstract class `Employee` has two subclasses, `Permanent` and `Temporary`. The `Employee` class contains an abstract method named `setType()`. Before you can instantiate `Permanent` and `Temporary` objects, which of the following statements must be true?
   a. You must code statements for the `setType()` method within the `Permanent` class.
   b. You must code statements for the `setType()` method within both the `Permanent` and `Temporary` classes.
   c. You must not code statements for the `setType()` method within either the `Permanent` or `Temporary` class.
   d. You can code statements for the `setType()` method within the `Permanent` class or the `Temporary` class, but not both.

6. When you create a superclass and one or more subclasses, each object of the subclass ____________ superclass object.
   a. overrides the
   b. "is a"
   c. "is not a"
   d. is a new

7. Which of the following statements is true?
   a. Superclass objects are members of their subclass.
   b. Superclasses can contain abstract methods.
   c. You can create an abstract class object using the `new` operator.
   d. An abstract class cannot contain an abstract method.

8. When you create a ____________ in Java, you create a variable name in which you can hold the memory address of an object.
   a. field
   b. pointer
   c. recommendation
   d. reference

9. An application's ability to select the correct subclass method to execute is known as ____________ method binding.
   a. polymorphic
   b. dynamic
   c. early
   d. intelligent

10. Which statement creates an array of five references to an abstract class named `Currency`?
   a. `Currency[] = new Currency[5];`
   b. `Currency[] currencyref = new Currency[5];`
   c. `Currency[5] currencyref = new Currency[5];`
   d. `Currency[5] = new Currency[5];`

11. You ______________ override the `toString()` method in any class you create.
    a. cannot
    b. can
    c. must
    d. must implement `StringListener` to

12. The `Object` class `equals()` method takes ______________.
    a. no arguments
    b. one argument
    c. two arguments
    d. as many arguments as you need

13. Assume the following statement appears in a working Java program:

    ```
    if(thing.equals(anotherThing)) x = 1;
    ```

    You know that ______________.
    a. `thing` is an object of the `Object` class
    b. `anotherThing` is the same type as `thing`
    c. Both of the above are correct.
    d. None of the above are correct.

14. The `Object` class `equals()` method considers two object references to be equal if they have the same ______________.
    a. value in all data fields
    b. value in any data field
    c. data type
    d. memory address

15. Java subclasses have the ability to inherit from ______________ parent class(es).
    a. one
    b. two
    c. multiple
    d. no

16. The alternative to multiple inheritance in Java is known as a(n) ______________.
    a. superobject
    b. abstract class
    c. interface
    d. none of the above

17. When you create a class that uses an interface, you include the keyword ______________ and the interface's name in the class header.
    a. `interface`
    b. `implements`
    c. `accouterments`
    d. `listener`

18. You can instantiate concrete objects from a(n) __________.

    a. abstract class
    b. interface
    c. either a or b
    d. neither a nor b

19. In Java, a class can __________.

    a. inherit from one abstract superclass at most
    b. implement one interface at most
    c. both a and b
    d. neither a nor b

20. When you want to provide some data or methods that subclasses can inherit, but you want the subclasses to override some specific methods, you should write a(n) __________.

    a. abstract class
    b. interface
    c. final superclass
    d. concrete object

Exercises

Programming Exercises

1. a. Create an abstract class named `Book`. Include a `String` field for the book's title and a `double` field for the book's price. Within the class, include a constructor that requires the book title, and add two get methods—one that returns the title and one that returns the price. Include an abstract method named `setPrice()`. Create two child classes of `Book`: `Fiction` and `NonFiction`. Each must include a `setPrice()` method that sets the price for all `Fiction Books` to $24.99 and for all `NonFiction Books` to $37.99. Write a constructor for each subclass, and include a call to `setPrice()` within each. Write an application demonstrating that you can create both a `Fiction` and a `NonFiction Book` and display their fields. Save the files as **Book.java**, **Fiction.java**, **NonFiction.java**, and **UseBook.java**.

   b. Write an application named `BookArray` in which you create an array that holds 10 `Books`, some `Fiction` and some `NonFiction`. Using a `for` loop, display details about all 10 books. Save the file as **BookArray.java**.

2. a. The Talk-A-Lot Cell Phone Company provides phone services for its customers. Create an abstract class named `PhoneCall` that includes a `String` field for a phone number and a `double` field for the price of the call. Also include a constructor that requires a phone number parameter and that sets the price to 0.0. Include a set method for the price. Also include three abstract get methods—one that returns the phone number, another that returns the price of the call, and a third that displays information about the call. Create two child classes of `PhoneCall`: `IncomingPhoneCall` and `OutgoingPhoneCall`. The `IncomingPhoneCall` constructor passes its phone number parameter to its parent's constructor and sets the price of the call to 0.02. The method that displays the phone call information displays the phone number, the rate, and the price of the call (which is the same as the rate). The `OutgoingPhoneCall` class includes an additional field that holds the time of the call in minutes. The constructor requires both a phone number and the time. The price is 0.04 per minute, and the display method shows the details of the call, including the phone number, the rate per minute, the number of minutes, and the total price. Write an application that demonstrates you can instantiate and display both `IncomingPhoneCall` and `OutgoingPhoneCall` objects. Save the files as **PhoneCall.java**, **IncomingPhoneCall.java**, **OutgoingPhoneCall.java**, and **DemoPhoneCalls.java**.

   b. Write an application in which you assign data to a mix of 10 `IncomingPhoneCall` and `OutgoingPhoneCall` objects into an array. Use a `for` loop to display the data. Save the file as **PhoneCallArray.java**.

3. Create an abstract `Auto` class with fields for the car make and price. Include get and set methods for these fields; the `setPrice()` method is abstract. Create two subclasses for individual automobile makers (for example, Ford or Chevy), and include appropriate `setPrice()` methods in each subclass (for example, $20,000 or $22,000). Finally, write an application that uses the `Auto` class and subclasses to display information about different cars. Save the files as **Auto.java**, **Ford.java**, **Chevy.java**, and **UseAuto.java**.

4. Create an abstract `Division` class with fields for a company's division name and account number, and an abstract `display()` method. Use a constructor in the superclass that requires values for both fields. Create two subclasses named `InternationalDivision` and `DomesticDivision`. The `InternationalDivision` includes a field for the country in which the division is located and a field for the language spoken; its constructor requires both. The `DomesticDivision` includes a field for the state in which the division is located; a value for this field is required by the constructor. Write an application named `UseDivision` that creates `InternationalDivision` and `DomesticDivision` objects for two different companies and displays information about them. Save the files as **Division.java**, **InternationalDivision.java**, **DomesticDivision.java**, and **UseDivision.java**.

5. Create an abstract class named `Element` that holds properties of elements, including their symbol, atomic number, and atomic weight. Include a constructor that requires values for all three properties and a get method for each value. (For example, the symbol for carbon is C, its atomic number is 6, and its atomic weight is 12.01. You can find these values by reading a periodic table in a chemistry reference or by searching the Web.) Also include an abstract method named `describeElement()`.

   Create two extended classes named `MetalElement` and `NonMetalElement`. Each contains a `describeElement()` method that displays the details of the element and a brief explanation of the properties of the element type. For example, metals are good conductors of electricity, while nonmetals are poor conductors. Write an application named `ElementArray` that creates and displays an array that holds at least two elements of each type. Save the files as **Element.java**, **MetalElement.java**, **NonMetalElement.java**, and **ElementArray.java**.

6. Create a class named `NewspaperSubscriber` with fields for a subscriber's street address and the subscription rate. Include get and set methods for the subscriber's street address, and include get and set methods for the subscription rate. The set method for the rate is abstract. Include an `equals()` method that indicates two `Subscribers` are equal if they have the same street address. Create child classes named `SevenDaySubscriber`, `WeekdaySubscriber`, and `WeekendSubscriber`. Each child class constructor sets the rate as follows: `SevenDaySubscribers` pay $4.50 per week, `WeekdaySubscribers` pay $3.50 per week, and `WeekendSubscribers` pay $2.00 per week. Each child class should include a `toString()` method that returns the street address, rate, and service type. Write an application named `Subscribers` that prompts the user for the subscriber's street address and requested service, and then creates the appropriate object based on the service type. Do not let the user enter more than one subscription type for any given street address. Save the files as **NewspaperSubscriber.java**, **WeekdaySubscriber.java**, **WeekendSubscriber.java**, **SevenDaySubscriber.java**, and **Subscribers.java**.

7. Picky Publishing House publishes stories in three categories and has strict requirements for page counts in each category. Create an abstract class named `Story` that includes a story title, an author name, a number of pages, and a `String` message. Include get and set methods for each field. The method that sets the number of pages is abstract. Also include constants for the page limits in each category. Create three `Story` subclasses named `Novel`, `Novella`, and `ShortStory`, each with a unique `setPages()` method. A `Novel` must have more than 100 pages, a `Novella` must have between 50 and 100 pages inclusive, and a `ShortStory` must have fewer than 50 pages. If the parameter passed to any of the set methods in the child class is out of range, set the page value but also create and store a message that indicates how many pages must be added or cut to satisfy the rules for the story type. Write an application named `StoryDemo` that creates an array of at least six objects to demonstrate how the methods work for objects created both with valid and invalid page counts for each story type. For each story, display the title, author, page count, and message if any was generated. Figure 11-34 shows a sample execution. Save the files as **Story.java**, **Novel.java**, **Novella.java**, **ShortStory.java**, and **StoryDemo.java**.

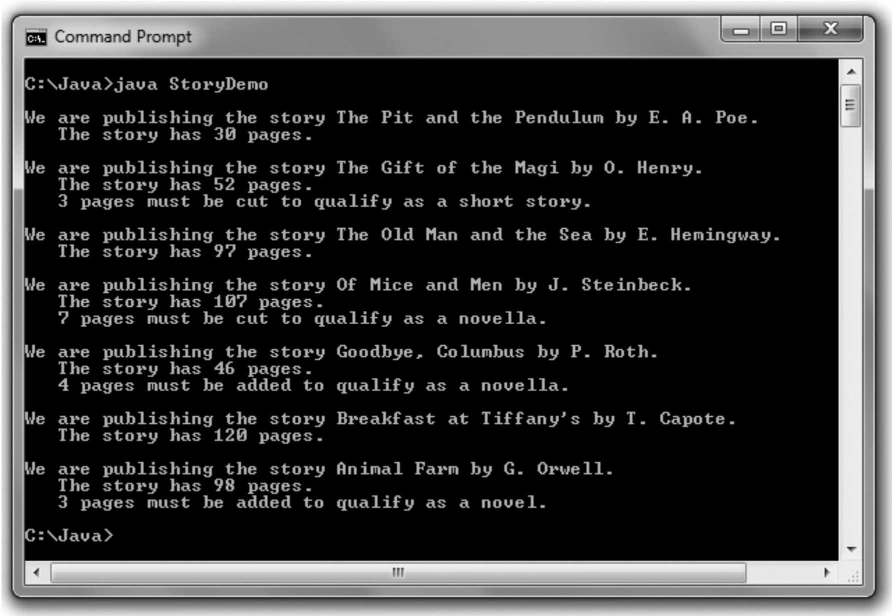

Figure 11-34 Typical execution of the `StoryDemo` application

8. a. Create an interface named `Turner`, with a single method named `turn()`. Create a class named `Leaf` that implements `turn()` to display "Changing colors". Create a class named `Page` that implements `turn()` to display "Going to the next page". Create a class named `Pancake` that implements `turn()` to display "Flipping". Write an application named `DemoTurners` that creates one object of each of these class types and demonstrates the `turn()` method for each class. Save the files as **Turner.java**, **Leaf.java**, **Page.java**, **Pancake.java**, and **DemoTurners.java**.

   b. Think of two more objects that use `turn()`, create classes for them, and then add objects to the `DemoTurners` application, renaming it **DemoTurners2.java**. Save the files, using the names of new objects that use `turn()`.

9. Write an application named `UseInsurance` that uses an abstract `Insurance` class and `Health` and `Life` subclasses to display different types of insurance policies and the cost per month. The `Insurance` class contains a `String` representing the type of insurance and a `double` that holds the monthly price. The `Insurance` class constructor requires a `String` argument indicating the type of insurance, but the `Life` and `Health` class constructors require no arguments. The `Insurance` class contains a get method for each field; it also contains two abstract methods named `setCost()` and `display()`. The `Life` class `setCost()` method sets the monthly fee to $36, and the `Health` class sets the monthly fee to $196. Write an application named `UseInsurance` that prompts the user for the type of insurance to be

displayed, and then create the appropriate object. Save the files as **Life.java**, **Health.java**, **Insurance.java**, and **UseInsurance.java**.

10. Create an abstract class called `GeometricFigure`. Each figure includes a height, a width, a figure type, and an area. Include an abstract method to determine the area of the figure. Create two subclasses called `Square` and `Triangle`. Create an application that demonstrates creating objects of both subclasses, and store them in an array. Save the files as **GeometricFigure.java**, **Square.java**, **Triangle.java**, and **UseGeometric.java**.

11. Modify Exercise 10, adding an interface called `SidedObject` that contains a method called `displaySides()`; this method displays the number of sides the object possesses. Modify the `GeometricFigure` subclasses to include the use of the interface to display the number of sides of the figure. Create an application that demonstrates the use of both subclasses. Save the files as **GeometricFigure2.java**, **Square2.java**, **Triangle2.java**, **SidedObject.java**, and **UseGeometric2.java**.

12. Create an interface called `Player`. The interface has an abstract method called `play()` that displays a message describing the meaning of "play" to the class. Create classes called `Child`, `Musician`, and `Actor` that all implement `Player`. Create an application that demonstrates the use of the classes. Save the files as **Player.java**, **Child.java**, **Actor.java**, **Musician.java**, and **UsePlayer.java**.

13. Create an abstract class called `Student`. The `Student` class includes a name and a Boolean value representing full-time status. Include an abstract method to determine the tuition, with full-time students paying a flat fee of $2,000 and part-time students paying $200 per credit hour. Create two subclasses called `FullTime` and `PartTime`. Create an application that demonstrates how to create objects of both subclasses. Save the files as **Student.java**, **FullTime.java**, **PartTime.java**, and **UseStudent.java**.

14. Create a `Building` class and two subclasses, `House` and `School`. The `Building` class contains fields for square footage and stories. The `House` class contains additional fields for number of bedrooms and baths. The `School` class contains additional fields for number of classrooms and grade level (for example, elementary or junior high). All the classes contain appropriate get and set methods. Place the `Building`, `House`, and `School` classes in a package named com.course.buildings. Create an application that declares objects of each type and uses the package. Save the necessary files as **Building.java**, **House.java**, **School.java**, and **CreateBuildings.java**.

15. Sanchez Construction Loan Co. makes loans of up to $100,000 for construction projects. There are two categories of `Loans`—those to businesses and those to individual applicants.

    Write an application that tracks all new construction loans. The application must also calculate the total amount owed at the due date (original loan amount + loan fee). The application should include the following classes:

- `Loan`—A public abstract class that implements the `LoanConstants` interface. A `Loan` includes a loan number, customer last name, amount of loan, interest rate, and term. The constructor requires data for each of the fields except interest rate. Do not allow loan amounts over $100,000. Force any loan term that is not one of the three defined in the `LoanConstants` class to a short-term, one-year loan. Create a `toString()` method that displays all the loan data.
- `LoanConstants`—A public interface class. `LoanConstants` includes constant values for short-term (one year), medium-term (three years), and long-term (five years) loans. It also contains constants for the company name and the maximum loan amount.
- `BusinessLoan`—A public class that extends `Loan`. The `BusinessLoan` constructor sets the interest rate to 1 percent over the current prime interest rate.
- `PersonalLoan`—A public class that extends `Loan`. The `PersonalLoan` constructor sets the interest rate to 2 percent over the current prime interest rate.
- `CreateLoans`—An application that creates an array of five `Loans`. Prompt the user for the current prime interest rate. Then, in a loop, prompt the user for a loan type and all relevant information for that loan. Store the created `Loan` objects in the array. When data entry is complete, display all the loans.

Save the files as **Loan.java**, **LoanConstants.java**, **BusinessLoan.java**, **PersonalLoan.java**, and **CreateLoans.java**.

Debugging Exercises

1. Each of the following files in the Chapter11 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, DebugEleven1.java will become FixDebugEleven1.java.

   a. DebugEleven1.java
   b. DebugEleven2.java
   c. DebugEleven3.java
   d. DebugEleven4.java
   e. Three other Debug files in the Chapter11 folder

Game Zone

1. In Chapter 10, you created an `Alien` class as well as two descendant classes, `Martian` and `Jupiterian`. Because you never create any “plain” `Alien` objects, alter the `Alien` class so it is abstract. Verify that the `Martian` and `Jupiterian` classes can still inherit from `Alien` and that the `CreateAliens` program still works correctly. Save the altered `Alien` file as **Alien.java**.

2. a. Create an abstract `CardGame` class similar to the one described in this chapter. The class contains a "deck" of 52 playing cards that uses a `Card` class to hold a suit and value for each `Card` object. It also contains an integer field that holds the number of cards dealt to a player in a particular game. The class contains a constructor that initializes the deck of cards with appropriate values (e.g., "King of Hearts"), and a `shuffle()` method that randomly arranges the positions of the `Cards` in the array. The class also contains two abstract methods: `displayDescription()`, which displays a brief description of the game in each of the child classes, and `deal()`, which deals the appropriate number of `Card` objects to one player of a game. Save the file as **CardGame.java**.

   b. Create two child classes that extend `CardGame`. You can choose any games you prefer. For example, you might create a `Poker` class or a `Bridge` class. Create a constructor for each child class that initializes the field that holds the number of cards dealt to the correct value. (For example, in standard poker, a player receives five cards, but in bridge, a player receives 13.) Create an appropriate `displayDescription()` and `deal()` method for each child class. Save each file using an appropriate name—for example, **Poker.java** or **Bridge.java**.

   c. Create an application that instantiates one object of each game type and demonstrates that the methods work correctly. Save the application as **PlayCardGames.java**.

Case Problems

1. a. In previous chapters, you have created several classes for Carly's Catering. Now, create a new abstract class named `Employee`. The class contains data fields for an employee's ID number, last name, first name, pay rate, and job title. The class contains get and set methods for each field; the set methods for pay rate and job title are abstract. Save the file as **Employee.java**.

   b. Create three classes that extend `Employee` named `Waitstaff`, `Bartender`, and `Coordinator`. The method that sets the pay rate in each class accepts a parameter and assigns it to the pay rate, but no `Waitstaff` employee can have a rate higher than 10.00, no `Bartender` can have a rate higher than 14.00, and no `Coordinator` can have a rate higher than 20.00. The method that sets the job title accepts no parameters—it simply assigns the string "waitstaff", "bartender", or "coordinator" to the object appropriately. Save the files as **Waitstaff.java**, **Bartender.java**, and **Coordinator.java**.

   c. In Chapter 10, you created a `DinnerEvent` class that holds event information, including menu choices. Modify the class to include an array of 15 `Employee` objects representing employees who might be assigned to work at a `DinnerEvent`. Include a method that accepts an `Employee` array parameter and assigns it to the `Employee` array field, and include a method that returns the `Employee` array. The filename is **DinnerEvent.java**.

d. Write an application that declares a `DinnerEvent` object, prompts the user for an event number, number of guests, menu options, and contact phone number, and then assigns them to the object. Also prompt the user to enter data for as many `Employees` as needed based on the number of guests. A `DinnerEvent` needs one `Waitstaff` `Employee` for every event, two if an event has 10 guests or more, three if an event has 20 guests or more, and so on. A `DinnerEvent` also needs one `Bartender` for every 25 guests and one `Coordinator` no matter how many guests attend. All of these `Employees` should be stored in the `Employee` array in the `DinnerEvent` object. (For many events, you will have empty `Employee` array positions.) After all the data values are entered, pass the `DinnerEvent` object to a method that displays all of the details for the event, including all the details about the `Employees` assigned to work. Save the program as **StaffDinnerEvent.java**.

2. a. In previous chapters, you have created several classes for Sammy's Seashore Supplies. Now, Sammy has decided to restructure his rates to include different fees for equipment types in addition to the fees based on rental length and to charge for required lessons for using certain equipment. Create an abstract class named `Equipment` that holds fields for a numeric equipment type, a `String` equipment name, and a fee for renting the equipment. Include a `final` array that holds the equipment names—jet ski, pontoon boat, rowboat, canoe, kayak, beach chair, umbrella, and other. Also include a `final` array that includes the surcharges for each equipment type—$50, $40, $15, $12, $10, $2, $1, and $0, respectively. Include a constructor that requires an equipment type and sets the field to the type unless it is out of range, in which case the type is set to the "other" code. Include get and set methods for each field, and include an abstract method that returns a `String` explaining the lesson policy for the type of equipment. Save the file as **Equipment.java**.

b. Create two classes that extend `Equipment`—`EquipmentWithoutLesson` and `EquipmentWithLesson`. The constructor for each class requires that the equipment type be in range—that is, jet skis, pontoon boats, rowboats, canoes, and kayaks are `EquipmentWithLesson` objects, but other equipment types are not. In both subclasses, the constructors set the equipment type to "other" if it is not in range. The constructors also set the equipment fee, as described in part 2a. Each subclass also includes a method that returns a message indicating whether a lesson is required, and the cost ($27) if it is. Save the files as **EquipmentWithoutLesson.java** and **EquipmentWithLesson.java**.

c. In Chapter 8, you created a `Rental` class. Now, modify it to contain an `Equipment` data field and an additional price field that holds a base price before equipment fees are added. Remove the array of equipment `Strings` from the `Rental` class as well as the method that returns an equipment string. Modify the `Rental` constructor so that it requires three parameters: contract number, minutes for the rental, and an equipment type. The method that sets the hours and minutes now sets a base price before equipment fees are included. Within the constructor, set the contract number and time as before, but add statements to create either an `EquipmentWithLesson` object or an `EquipmentWithoutLesson` object, and assign it to the `Equipment` data field. Assign the sum of the base price (based on time) and the equipment fee (based on the type of equipment) to the price field. Save the file as **Rental.java**.

d. In Chapter 8, you created a `RentalDemo` class that displays details for four `Rental` objects. Modify the class as necessary to use the revised `Rental` class that contains an `Equipment` field. Be sure to modify the method that displays details for the `Rental` to include all the pertinent data for the equipment. Figure 11-35 shows the output from a typical execution. Save the file as **RentalDemo.java**.

```
Command Prompt

Contract #A213
For a rental of 1 hours and 0 minutes,
    at a rate of $40 per hour and $1 per extra minute,
    the base price is $40.0
Contact phone number is: (273) 647-5869
Equipment rented is type #0 -- jet ski
This type of rental requires a lesson for $27.0.
The equipment fee with lesson is 77.0
The total price is 117.0
-------------------------------

Contract #B456
For a rental of 2 hours and 5 minutes,
    at a rate of $40 per hour and $1 per extra minute,
    the base price is $85.0
Contact phone number is: (345) 768-9088
Equipment rented is type #1 -- pontoon boat
This type of rental requires a lesson for $27.0.
The equipment fee with lesson is 67.0
The total price is 152.0
-------------------------------

Contract #C671
For a rental of 6 hours and 50 minutes,
    at a rate of $40 per hour and $1 per extra minute,
    the base price is $280.0
Contact phone number is: (465) 123-9088
Equipment rented is type #5 -- beach chair

This type of rental does not require a lesson.
The equipment fee with lesson is 2.0
The total price is 282.0
-------------------------------

Contract #D810
For a rental of 1 hours and 5 minutes,
    at a rate of $40 per hour and $1 per extra minute,
    the base price is $45.0
Contact phone number is: (812) 876-7667
Equipment rented is type #6 -- umbrella

This type of rental does not require a lesson.
The equipment fee with lesson is 1.0
The total price is 46.0
-------------------------------

C:\Java>
```

Figure 11-35 Output of typical `RentalDemo` execution