# CHAPTER 6

# Inheritance

## CHAPTER CONTENTS

Java gets richer through inheritance! Inheritance means that a class gets some fields and methods from another class. The advantage is that a new class can reuse code from existing classes, which saves time and effort for the programmer. In this chapter, we will discuss how a class can inherit from another class. In addition, we will use several code examples to illustrate how to include animation in programs—we will make cars move, airplanes fly, and other fun stuff!

## 6.1 What Is Inheritance?

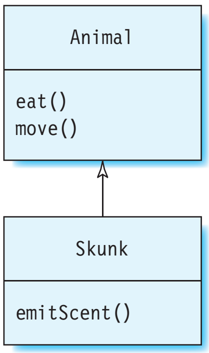
Consider a class `Animal` with two methods: `eat` and `move`. Suppose that we would like to create a class `Skunk` with the same methods as in `Animal`, and an additional method `emitScent`. With inheritance, `Skunk` can acquire the methods in `Animal`, removing the need to rewrite these methods in `Skunk`. Additionally, `Skunk` can be specialized with a new method `emitScent`, which highlights a well-known trait of skunks. `Skunk` is called the **child class** or **subclass**, and `Animal` is called the **parent class** or **superclass**. This inheritance relationship between `Animal` and `Skunk` is depicted in Figure 6–1 by using an arrowhead that is not filled in. The arrow means that `Skunk` reuses code in `Animal`. In other words, `Skunk` is *derived* from `Animal`.

## 6.2 When Is Inheritance Used?

A class `X` can inherit the fields and methods of another class `Y` when there is a special type of **is-a** relationship between these two classes. The "is-a" rela-

**Figure 6–1**

**Class `Skunk` inherits from the `Animal` class.**



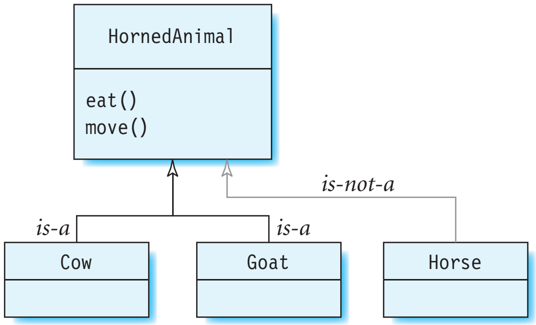



**Figure 6–2**
**Cow and Goat can inherit from HornedAnimal, but Horse should not.**

tionship means that X is a specialization of Y. For example, Skunk is-a Animal is true.

Assume that we have written a class called Vehicle and another called Car. Class Car can inherit from Vehicle because Car is-a Vehicle. Note that the reverse is not true, because not all vehicles are cars. Therefore, Vehicle cannot inherit from Car. In addition, if a third class called Door exists, it may seem plausible that Car should inherit from Door. However, Car is-a Door and Door is-a Car are both not true, so neither should inherit from the other. The derived class is a specialized form of the parent class. Therefore, it is important that this "is-a" relationship holds true whenever you write a derived class.

Figure 6–2 shows that the Cow and Goat classes can inherit from the HornedAnimal class. Next, we want to write a class Horse that also has two methods: eat and move. You may want to make Horse a subclass of HornedAnimal merely to reuse the methods available in the latter, but this is a bad idea—Horse is not a HornedAnimal, and so it should not inherit from HornedAnimal.

## 6.3 The extends Keyword

The keyword extends is used to denote that one class inherits from another class. The code for class Skunk that inherits from Animal is shown here:

```
public class Skunk extends Animal {
  public void emitScent() {
    System.out.println("Emit scent");
  }
}
```

The `Animal` class implements the methods `eat` and `move`, which are inherited by `Skunk`:

```
public class Animal {
  public void eat() {
    System.out.println("Eat");
  }
  public void move() {
    System.out.println("Move");
  }
}
```

As discussed previously, the advantage of using inheritance is that `Skunk` can reuse the code in `Animal`. Otherwise, `Skunk` would need to define explicitly all the fields and methods, as shown in the `NotSoSmartSkunk` class (see Figure 6–3):

```
//Class NotSoSmartSkunk does not use inheritance
public class NotSoSmartSkunk {
  public void eat() {
    System.out.println("Eat");
  }
  public void move() {
    System.out.println("Move");
  }
  public void emitScent() {
    System.out.println("Emit scent");
  }
}
```

The `Skunk` class is the same as the `NotSoSmartSkunk` class. However, less code (and effort) is required to write the `Skunk` class using inheritance. Inheritance is especially useful when the superclass is large and complex, or when there are many classes that share a portion of code.

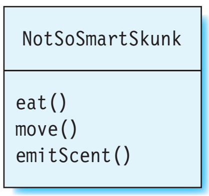



**Figure 6–3**

**The `NotSoSmartSkunk` class must implement all three methods because it does not use inheritance.**

## 6.4 Types of Inheritance

**Single-level inheritance** is a simple form of inheritance in which there are only two classes involved: one superclass and a subclass. Its form is shown in Figure 6–4. Here, class `BoardGame` inherits from class `Game`. Therefore, `Game` is a superclass and `BoardGame` is a subclass of `Game`.

In **multilevel inheritance,** a class that acts as both a superclass and a subclass is present. For example, let us add a third class `Chess` that inherits from `BoardGame`, as shown in Figure 6–5. Here, `BoardGame` is a superclass of `Chess` and a subclass of `Game`. Now, `Chess` inherits the fields and methods of both classes `Game` and `BoardGame`. There can be any number of classes in the multilevel inheritance path; for example, class `V` inherits from class `U`, class `W` inherits from `V`, and so on.

**Hierarchical inheritance** is a form of inheritance in which many classes inherit from the same superclass. An example is shown in Figure 6–6. In the figure, `BoardGame`, `TableGame`, and `CourtGame` inherit fields and methods from the same superclass `Game`. However, some of the other fields and methods in these three subclasses will differ.

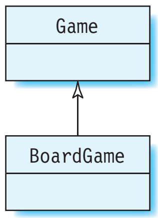



**Figure 6–4**
**Single-level inheritance.**

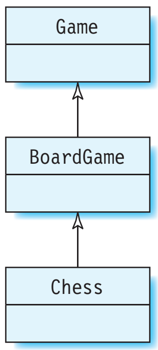



**Figure 6–5**
**Multilevel inheritance.**

**Figure 6–6**
**Hierarchical inheritance.**

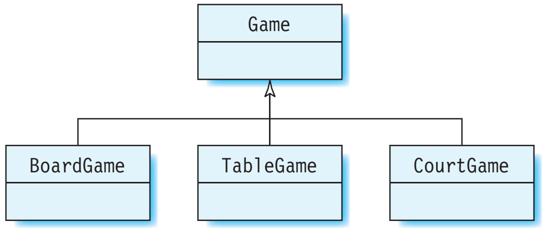
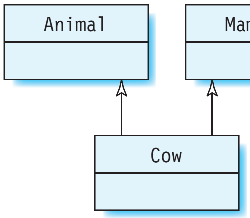



**Figure 6–7**
**Multiple inheritance is not supported in Java.**

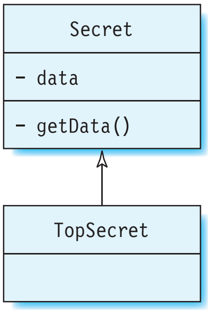



In **multiple inheritance**, a class can have more than one superclass. This form of inheritance is not supported in Java, although it is supported in other languages, such as C++ and Python. Figure 6–7 shows an example of multiple inheritance in which class `Cow` has two superclasses, `Animal` and `Mammal`.

## 6.5 Access Modifiers

The subclass can access the methods and fields of a superclass just as it would access its own methods and fields. Note that constructors are not inherited. In addition, some restrictions are imposed by the access modifiers used. In Chapter 5, we discussed the four types of access identifiers: `private`, `public`, *default*, and `protected`. We also described how these modifiers affect the ability of a class to access the fields and methods in another (unrelated) class. Now, we extend this discussion to the case when the classes are related; that is, one class is a subclass of another. These access modifiers decide which fields and methods can be inherited by a subclass:

- `public`: All `public` fields and methods of the parent class are inherited by its child class. For example, the `Skunk` class inherits all of the public methods in `Animal`.
- `private`: Fields and methods that are declared `private` cannot be inherited by a subclass. This structure is shown in Figure 6–8.



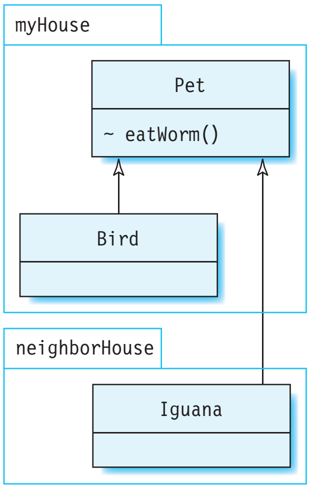

```
//code segment
TopSecret t = new Top Secret();
t.getData();      // error!
int v = t.data;   // error!
```

**Figure 6–8**

**Private fields and methods of `Secret` cannot be accessed by `TopSecret`.**

- *Package-private* or *default* access modifier: Fields and methods that have a package-private access modifier are inherited by a subclass *only* if it is in the same package as the superclass. This structure is shown in Figure 6–9. Both `Bird` and `Iguana` are subclasses of `Pet`; however, only `Pet` and `Bird` are in the same package `myHouse`. Therefore, only `Bird` (and not `Iguana`) can access the package-private method `eatWorm` in `Pet`.
- `protected`: Fields and methods declared as `protected` are inherited by a subclass even if it is in a different package from its superclass. Therefore, if the `eatWorm` method in `Pet` were instead declared as `protected`, `Iguana` would be able to access it.

When a class is designed, a different access modifier can be selected for each of its fields and methods, depending upon the amount of visibility required




```
//code segment
Bird earlyBird = new Bird();
earlyBird.eatWorm(); // okay

Iguana lizard = new Iguana();
lizard.eatWorm();    // error!
```

**Figure 6–9**

**`Bird` can access the package-private method `eatWorm`, but `Iguana` cannot.**

for that field or method. If a field or method is to be used only within the class, it should be declared as `private`. On the other hand, if a field or method is to be used within the class and by its subclasses, it should be declared as `protected`.

**The `Vehicle` Class** We are going to create a new class called `Vehicle`, and we will build upon it through the rest of the chapter to explain new concepts:

```
package inheritance;
import java.awt.*;

public class Vehicle {
   // method to draw shape of Vehicle
   protected void drawShape(Graphics2D myGraphics) {
   }
}
```

This class contains a single empty method called `drawShape`. For convenience, we put this and all of the other classes in this chapter in a package called `inheritance` in the `src` directory. You should add this statement to all the classes in this chapter:

```
package inheritance;
```

Next, we create two subclasses of `Vehicle`, called `Car` and `Airplane`. `Car` and `Airplane` are types of `Vehicle`; that is, they have an "is-a" relationship with `Vehicle`, and so they can inherit the code in `Vehicle`. In addition, these classes will be developed further by adding new methods to them. This arrangement is a form of hierarchical inheritance and is shown in Figure 6–10.

The class declaration for `Car`, which is a subclass of `Vehicle`, is shown here:

```
package inheritance;

public class Car extends Vehicle {
}
```

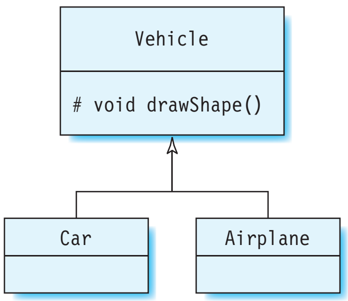



**Figure 6–10**
**Class `Vehicle` and its subclasses, `Car` and `Airplane`.**

The class declaration for Airplane, which is also a subclass of Vehicle, is shown here:

```
package inheritance;

public class Airplane extends Vehicle {
}
```

We will develop these two classes further later in this chapter.

## 6.6 Overriding Methods

A subclass can contain a method with the same *signature* and *return type* as in its superclass. In this case, the method in the superclass is not inherited. The method of the subclass is said to **override** the method of the superclass.

An **overriding** method has:

- the same signature (method name, number of parameters, and parameter types) as a method M in the superclass, and
- the same return type as M, and
- a different body from M.

Consider a Cat class with subclasses HouseCat and Lion. The Cat class contains the method vocalize:

```
// vocalize method in Cat
public void vocalize() {
  System.out.println("Meow");
}
```

With method overriding, the Lion class can define its own vocalize method instead of inheriting it from Cat:

```
// vocalize method in Lion
public void vocalize() {
  System.out.println("ROAR!");
}
```

The signature and return type of the overriding method in the subclass should match those of the overridden method in the superclass exactly. Let us add a main method to this class:

```
public static void main(String[] args) {
  Lion lion = new Lion();
  lion.vocalize();
  HouseCat housecat = new HouseCat();
  housecat.vocalize();
}
```

The program output is shown here:

```
ROAR!
Meow
```

The `vocalize` method of class `Lion`, and not class `Cat`, is called. On the other hand, the `HouseCat` class inherits the `vocalize` method of `Cat` because it does not have an overriding method. Method overriding is shown in Figure 6–11.

Fields that are inherited from the superclass do not have to be overridden because instances can select different values for the same field. The `type` field of `Cat` is inherited by `Lion` and `HouseCat`, and instances of both classes can assign different values to this field. However, if the subclass contains a field with the same *name* as a field inherited from the superclass, the superclass field is not visible to the subclass even if the types of the two fields are different. In this case, the superclass field is said to be **hidden** by the subclass field.

It is not a good idea to add the field `type` to `Lion` and `HouseCat` because this would hide the field inherited from `Cat`.

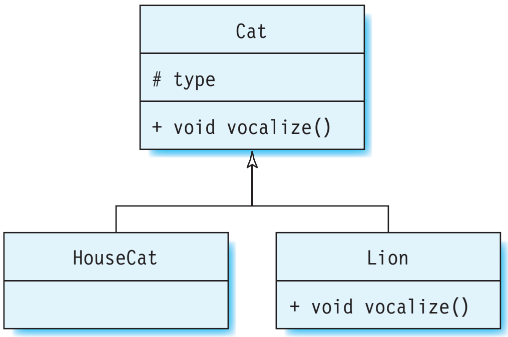
> Fields should not be hidden because doing so makes the code confusing.

### 6.6.1 Polymorphism

**Polymorphism** refers to an object's ability to select a particular behavior based on its type while the program is executing. To understand how polymorphism works, consider the following statement:

```
Cat cat = new Lion();
```

**Figure 6–11**
**The `vocalize` method in the `Lion` class overrides the `vocalize` method in `Cat`.**



An object of type `Lion` is assigned to the reference variable `cat` of type `Cat`. This is different from the way in which we have been constructing objects up until now, where the reference variable and the object belong to the same class, as in the following example:

```
Lion cat1 = new Lion();
```

However, by inheritance, `Lion` is-a `Cat`, and so we can assign an object of type `Lion` to a reference variable of type `Cat`. In other words, *an object can be assigned to a reference variable of its superclass*. This process is known as **upcasting**. Similarly, we can also write the following:

```
Cat cat2 = new HouseCat();
```

The reverse assignment cannot be made; that is, we cannot assign an object of `Cat` to a reference variable of type `HouseCat`:

```
HouseCat cat3 = new Cat(); // error!
```

In this case, an explicit cast must be used:

```
HouseCat cat3 = (HouseCat) new Cat(); // okay as cast is used
```

The cast consists of a class type within parentheses that is placed in front of the object being cast. This is similar to casting primitive types, in which a larger type is converted to a narrower type using a cast. However, a cast cannot be used with two unrelated classes:

```
Vehicle v = (Vehicle) new Animal();  // error
```

It simply does not make sense to cast an object of type `Animal` to type `Vehicle`, because these classes are not related. Doing so will cause a compiler to issue an error message that the cast from `Animal` to `Vehicle` is not allowed.

Furthermore, suppose that the method `vocalize` is called as follows:

```
Cat cat = new Lion();
cat.vocalize();
```

Which `vocalize` method is called now—that of `Cat` or `Lion`? To find out, add this `main` method to the `Lion` or `HouseCat` class:

```
public static void main(String[] args) {
  Cat lion = new Lion();
  lion.vocalize();
  Cat housecat = new HouseCat();
  housecat.vocalize();
}
```

The program output is:

```
ROAR!
Meow
```

The correct `vocalize` method of a particular object is called, even though the reference variable is of type `Cat` in both cases. `lion` references an object of type `Lion`; as a result, it calls the `vocalize` method of `Lion`, and not `Cat`. Thus, the method called by an object depends on its type determined at run time—a process known as **polymorphism**. Polymorphism is achieved via *method overriding* in this example.

### 6.6.2 Access Modifiers for Overriding Methods

The access modifier of the superclass method decides what the access modifier of the overriding subclass method can be. Table 6–1 shows the relationship between the access modifier of a superclass method and the corresponding overriding method in the subclass. If the access modifier of the superclass method is `protected`, the overriding method in the subclass can only be `protected` or `public`—it cannot be `private`. The reason for this is that an overriding method in the subclass cannot be *less* visible than the corresponding method in the superclass. The relationships for the package-private and `public` access modifiers are also shown in the table. If the access modifier of the superclass method is `private`, the method is not inherited by the subclass, and thus it cannot be overridden in the subclass.

The following example explains, in more detail, how the modifiers can be used.

**TABLE 6–1 Access Modifiers for Overriding Methods**

| Access Modifier of a Superclass Method | Access Modifier of an Overriding Subclass Method |
|---|---|
| `protected` | `protected` or `public` |
| package-private | package-private, `protected`, or `public` |
| `public` | `public` |

### Example 1

Consider two classes Bank and OnlineBank, where OnlineBank is a subclass of Bank. Only the method declarations in these two classes are shown here:

```
class Bank {
      public void deposit(float amount) { }
      protected void withdraw(float amount) { }
      void name() { };
      private void update() { };
}

class OnlineBank extends Bank {
      private void deposit(float amount) { }
      private void withdraw(float amount) { }
      protected void name() { }
      private void update() { }
}
```

Pick out the overriding methods in OnlineBank. Using Table 6–1, explain which access modifiers in OnlineBank are incorrect and result in an error.

***Solution:***

```
class OnlineBank extends Bank {
   private void deposit(float amount); // error!
   private void withdraw(float amount); // error!
   protected void name(); // overrides method name in Bank
   private void update(); // does not override method update in Bank
}
```

The method deposit cannot have an access modifier of private. The reason is that, according to Table 6–1, if a method is declared as public in a superclass, the overriding method should also be public in the subclass. Therefore, deposit should be made public. Similarly, withdraw also has an incorrect access modifier of private. This method is declared as protected in Bank, and thus it can have access modifiers of protected or public only. The method update is declared as private in Bank, and it is not visible to OnlineBank. Therefore, update in OnlineBank does not override update in Bank. ■

### 6.6.3 Covariant Return Types

Recall that an overriding method has the same return type as the method in its superclass. However, an exception exists when the return types are different. These return types are called **covariant return types** and they differ in that the return type of the overriding method can be a *subclass* of the return type of its superclass method. We explain this with an example.

Consider four classes `Animal`, `Goat`, `Zoo`, and `PettingZoo`, where `Goat` is derived from `Animal`, and `PettingZoo` is derived from `Zoo`. The class `Zoo` has a method called `getAnimal` with the return type `Animal`:

```
public class Zoo {
  private Animal myAnimal;

  public Animal getAnimal() {
    return myAnimal;
  }
}
```

The `getAnimal` method in `PettingZoo` has a return type of `Goat`:

```
public class PettingZoo extends Zoo{
  private Goat billy;
  public PettingZoo() {
    billy = new Goat();
  }
  // A return type of Goat is allowed
  // since Goat is a subclass of Animal.
  public Goat getAnimal() {
    return billy;
  }
}
```

The return type of `getAnimal` is `Goat`, which does not match the return type of `Animal` in the superclass method. However, no error exists, because `Goat` is a subclass of `Animal`. Therefore, the `getAnimal` method of `PettingZoo` overrides that of `Zoo` correctly. On the other hand, if the return type of the subclass method were different and not a subclass of the return type of the superclass method, it would be flagged as a compilation error.

**Adding an Overriding Method to `Car`** Here, we will add a method called `drawShape` to the `Car` class to draw the shape of a car. This method overrides the empty method `drawShape` in `Vehicle`.

It is important to note that all drawing is done relative to a point on the vehicles. Any point on the object can be chosen as this **reference point**. For the car, we select the upper-left corner of the trunk as the reference point ($x$, $y$). Recall that the reason for using this reference point is so that we can draw this object at another position in the frame by merely changing the values of *x* and *y*, which will make it easy to animate the objects. Thus, the position of all graphics objects, such as ellipses, lines, and rectangles, that we will use to draw our vehicles, will be specified relative to point (*x*, *y*).

```
// Method in class Car to draw the shape of a car.
public void drawShape(Graphics2D myGraphics) {
  int w1 = 250, h1 = 90, w2 = 143, h2 = 75, w4 = 50, h4 = 45, w5 = 35;
  float e1 = 62.5f, e2 = 22.5f, e3 = 125, e4 = 45, w3 = 16.67f, h3 = 30;
  // draw the lower body
  RoundRectangle2D.Float lower = new RoundRectangle2D.Float(x, y, w1,
h1, e1, e2);
  myGraphics.setPaint(Color.white);
  myGraphics.fill(lower);
  myGraphics.setPaint(Color.blue);
  myGraphics.draw(lower);

  // draw the upper body
  RoundRectangle2D.Float mid = new RoundRectangle2D.Float(x+50, y-63, w2,
h2, e1, e2);
  myGraphics.setPaint(Color.white);
  myGraphics.fill(mid);
  myGraphics.setPaint(Color.blue);
  myGraphics.draw(mid);
  Rectangle2D.Float top = new Rectangle2D.Float(x+50, y, w2, w4/2);
  myGraphics.setPaint(Color.white);
  myGraphics.fill(top);

  // color a yellow headlight
  Ellipse2D.Float light = new Ellipse2D.Float(x+238, y+18, w3, h3);
  myGraphics.setPaint(Color.yellow);
  myGraphics.fill(light);
  myGraphics.setPaint(Color.black);
  myGraphics.draw(light);

  // color a red taillight
  Ellipse2D.Float taillight= new Ellipse2D.Float(x-10, y+18, w3, h3);
  myGraphics.setPaint(Color.red);
  myGraphics.fill(taillight);

  // color windows
  RoundRectangle2D.Float window1 = new RoundRectangle2D.Float(x+62.5f, y-
45, w4, h4, e1/2, e2/2);
  myGraphics.setPaint(Color.lightGray);
  myGraphics.fill(window1);
```

```
  RoundRectangle2D.Float window2 = new RoundRectangle2D.Float(x+125, y-45,
w4, h4, e1/2, e2/2);
  myGraphics.fill(window2);

  // color the bumpers
  RoundRectangle2D.Float b1 = new RoundRectangle2D.Float(x+225, y+65, e1/2,
e2, e3, e4);
  myGraphics.setPaint(Color.gray);
  myGraphics.fill(b1);

  RoundRectangle2D.Float b2 = new RoundRectangle2D.Float(x-5, y+65, w5,
e2, e3, e4);
  myGraphics.fill(b2);

  // draw the wheels
  Ellipse2D.Float wh1 = new Ellipse2D.Float(x+37.5f, y+63, w4, w4);
  Ellipse2D.Float wh2 = new Ellipse2D.Float(x+167.5f, y+63, w4, w4);
  myGraphics.setPaint(Color.white);
  myGraphics.fill(wh1);
  myGraphics.fill(wh2);
  myGraphics.setPaint(Color.darkGray);
  myGraphics.draw(wh1);
  myGraphics.draw(wh2);
}
```

In addition, we add two fields `x` and `y` to `Vehicle`, to represent the *x*- and *y*-coordinates of the vehicle's position in the window.

```
public class Vehicle {
  // vehicle's position
  protected float x = 30, y = 300;
  // method to draw shape of Vehicle
  protected void drawShape(Graphics2D myGraphics) {
  }
}
```

Add these `import` statements to `Car`:

```
import java.awt.*;
import java.awt.geom.*;
import com.programwithjava.basic.DrawingKit;
```

Finally, add this `main` method to `Car`, and then run the program:

```
public static void main(String[] args) {
 DrawingKit dk = new DrawingKit("Car");
```

```
 Graphics2D myGraphics = dk.getGraphics();
 Vehicle myVehicle = new Car();
 myVehicle.drawShape(myGraphics);
}
```

Place the classes `Vehicle.java`, `Car.java`, and `Airplane.java` in the `inheritance` package inside the `JavaBook\src` directory. To compile and run this program, type the following at the command prompt:

```
C:\JavaBook> javac -d bin src\com\programwithjava\basic\DrawingKit.java
src\inheritance\Vehicle.java src\inheritance\Car.java

C:\JavaBook> java -classpath bin inheritance.Car
```

The overriding method `drawShape` of `Car` will be called and the shape will be drawn as shown in Figure 6–12.

### 6.6.4 The Advantage of Overriding Methods

So, what is the advantage of overriding methods? The advantage is that the decision of which of these methods should be used can be put off until run time. Therefore, we can simply write the following:

```
v.drawShape(myGraphics);
```

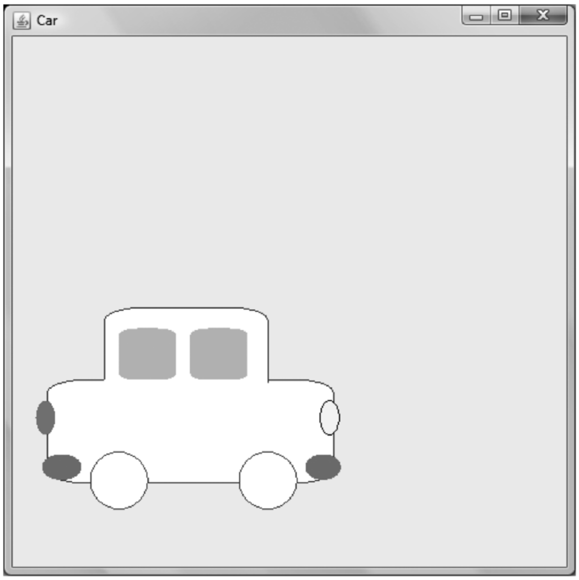



**Figure 6–12**
**Drawing a car using the overriding method `drawShape` of class `Car`.**

At run time, depending on whether `v` is an object of type `Car` or `Airplane`, the corresponding `drawShape` method of that class is invoked. We clarify this with an example. Let us add a third class called `Traffic`, as shown here:

```
package inheritance;
import java.awt.*;
import com.programwithjava.basic.DrawingKit;

public class Traffic {
  Traffic(Vehicle v) {
    DrawingKit dk = new DrawingKit("Traffic");
    Graphics2D myGraphics = dk.getGraphics();
    v.drawShape(myGraphics);
  }
}
```

The constructor for `Traffic` has a parameter of type `Vehicle`. The `Car` and `Airplane` classes are subclasses of `Vehicle`; as a result, objects of these classes can also be passed as an argument to this constructor. Add the following `main` method to the `Traffic` class:

```
public static void main(String[] args) {
  Car c = new Car();
  Traffic t = new Traffic(c); // Call to Traffic constructor
}
```

The `Traffic` constructor is called and the object is passed in as an argument to this constructor is of type `Car`. Therefore, the statement `v.drawShape` calls the `drawShape` method of `Car`.

If method overriding were not allowed, how could the `Traffic` class be written to accomplish the same effect? For one, there could be multiple constructors, each with a different type of parameter:

```
public class Traffic {
  Traffic(Vehicle v) {
    // some code
    v.drawShape(myGraphics);
  }

  Traffic(Car v) {
    // some code
    v.drawShape(myGraphics);
  }

  Traffic(Airplane v) {
```

```
    // some code
    v.drawShape(myGraphics);
  }
}
```

Although this alternative works, it makes the code both lengthy and difficult to maintain because each time a new subclass of `Vehicle` is added, the code for `Traffic` must be updated. Of course, the ability to override methods removes the need to do any of this, which makes the technique very useful.

### 6.6.5 The super Keyword

The subclass can call the overridden method in its superclass by using a special keyword called `super`. The next example shows why this keyword is useful, and how it can be used.

Suppose that we have a class called `PieRecipe` that contains a method called `getDirections`. This method describes how to make the crust of a pie:

```
package inheritance;

public class PieRecipe {
  public void getDirections() {
    System.out.println("To prepare crust, roll out dough and chill in
pie pan.");
  }
}
```

The class `BlueberryPieRecipe` extends the class `PieRecipe`. It contains an overriding method called `getDirections` that describes how to make the filling for the pie:

```
package inheritance;

public class BlueberryPieRecipe extends PieRecipe {
  // overriding method in BlueberryPieRecipe
  public void getDirections() {
    System.out.println("To prepare filling, combine blueberries, flour,
lemon juice and sugar and put in pie pan, then cover with extra dough
and bake.");
  }
}
```

There is a problem with `BlueberryPieRecipe`—its `getDirections` method only describes how to make the filling and not the crust. The `getDirections` method in PieRecipe describes how to make the crust, but this method is overridden and therefore is not inherited by BlueberryPieRecipe. This problem can be easily resolved by using the keyword super. The overridden method in PieRecipe can be accessed in the subclass using the following method:

```
super.getDirections();
```

The getDirections method in BlueberryPieRecipe is modified as shown here:

```
// modified method in BlueberryPieRecipe
public void getDirections() {
      super.getDirections();
      System.out.println("To prepare filling, combine blueberries,
flour, lemon juice and sugar and put in pie pan, then cover with extra
dough and bake.");
}
```

Write a main method to test the getDirections method in BlueberryPieRecipe:

```
public static void main(String[] args) {
  PieRecipe r = new BlueberryPieRecipe();
  r.getDirections();
}
```

The directions for both the crust and the filling are printed out when the program is run:

```
To prepare crust, roll out dough and chill in pie pan.
To prepare filling, combine blueberries, flour, lemon juice and sugar and
put in pie pan, then cover with extra dough and bake.
```

Hidden fields can also be accessed in the subclass by using super. Suppose that BlueberryPieRecipe contains a field called ingredients that hides this field in PieRecipe. Then the hidden field can be accessed in BlueberryPieRecipe using super.ingredients. Remember, though, that in general, fields should not be hidden.

## 6.7 Overloaded Methods

As with constructors, methods can also be overloaded. Overloaded methods are methods with the same name but different parameter lists. (It is important not to confuse *overloaded* methods with the *overriding* methods we discussed earlier.)

An **overloaded method** meets the following requirements:

- It has the same name as another method `M` within the class or in a superclass.
- It has a *different* parameter list from `M`.

The return types and access modifiers of overloaded methods do not have to be the same. However, the data types of the parameters in the method declaration should be different in these methods.

For example, consider a class `Geom` with two methods called `intersects`:

```
class Geom {
  public static boolean intersects(Line2D.Float line1, Line2D.Float line2)
{
    /* code to check if line1 intersects with line2 */
  }

  public static boolean intersects(Rectangle2D.Float r, Line2D.Float line1)
{
    /* code to check if line1 intersects the rectangle r */
  }
}
```

The `intersects` methods are said to be *overloaded* because they have the same name, but a different parameter list. Although the bodies of overloaded methods are different, the goal is to provide the same functionality. For example, both methods check whether the given shapes intersect or not. Nevertheless, which of these methods will be called? This depends on the data type of the arguments passed to `intersects`. If both arguments are of type `Line2D.Float`, the first method will be called. Alternately, if the first argument is of type `Rectangle2D.Float` and the second is of type `Line2D.Float`, the second method is called. In general, the overloaded method to be invoked is the one whose parameter types match those of the arguments in the method call.

You are already familiar with the `println` method. This method is also overloaded, and for this reason, we can use it to print out arguments that are of different types, such as `int`, `float`, and `String`. This is more convenient than having to call a method with a different name for each data type.

Java 2D contains many classes with overloaded methods. One example is the `Rectangle` class. The `Rectangle` class has four overloaded methods called `contains`, each of which checks whether a given point or shape lies inside the `Rectangle` object and returns `true` or `false` accordingly:

> **`public boolean contains(int x, int y)`**—a method that checks whether the point `(x, y)` is inside the `Rectangle` object that calls this method.
>
> **`public boolean contains(int x, int y, int w, int h)`**—a method that checks whether the rectangle formed at coordinates `(x, y)` with width `w` and height `h` is completely inside the `Rectangle` object that calls this method.
>
> **`public boolean contains(Point p)`**—a method that checks whether the object `p` of type `Point` is inside the `Rectangle` object that calls this method. (`Point` is another class in Java 2D.)
>
> **`public boolean contains(Rectangle r)`**—a method that checks whether the `Rectangle` object `r` is entirely within the `Rectangle` object that calls this method.

## Example 2

Which of the following are valid declarations of the overloaded method `compute` in class `Abacus`?

```
class Abacus {
  public int compute(int a, double b, int c) {
    /* some code */
}
  private long compute(double a, long b) {
    /* some code */
  }
  public float compute(double b, int a, int c) {
    /* some code */
  }
  public double compute(double a1, long b1) {
    /* some code */
  }
}
```

***Solution:*** Overloaded methods must not have the same signature. The second and fourth methods have the same signature, shown here, which causes a compilation error:

```
compute(double, long)
```

Note that although the first and third methods both contain two `int`s and one `double` in the method signature, they are valid because the order of these parameters is different. ■

## 6.8 Constructor Chaining

As you already know, an object of a class is created using a constructor of that class. In Chapter 5, we discussed the three different types of constructors: default, constructor without parameters, and constructor with parameters. In this section, we are going to see how the object is created when inheritance is used, because there are multiple constructors involved. It is important to understand these two key points:

1. **A class does not inherit the constructors from its superclass:** Whereas methods and fields can be inherited by a subclass, constructors are *not* inherited.
2. **The first line of any constructor is a call to another constructor:** Either a superclass constructor or another constructor within the same class is called. This happens in one of the following two ways:
   a. Java automatically calls the superclass constructor.
   b. The programmer explicitly calls a superclass constructor or another constructor in the same class.

We will examine these in more detail next.

### 6.8.1 Java Automatically Calls the Superclass Constructor

To create an object of a class `Y`, all of the objects in the inheritance hierarchy of `Y`, starting at the parent class, must be created. Java calls the superclass constructor by inserting the following statement automatically in the first line of the constructor body:

```
super();
```

The `super` keyword was used earlier to call the superclass methods. Here we use it to call the superclass *constructor*. If the superclass does not have any constructors, the *default constructor* of the superclass will be executed. Otherwise, the *constructor without parameters* in the superclass is executed. We clarify this with an example. Figure 6–13(a) shows two classes, `Structure` and `House`, where `Structure` is a superclass of `House`.

`Structure` contains a constructor without parameters:

```
package inheritance;

public class Structure {
  // constructor without parameters
  public Structure() {
    System.out.println("Build foundation.");
  }
}
```

`House` also has a constructor without parameters. In `main`, create an object of `House`:

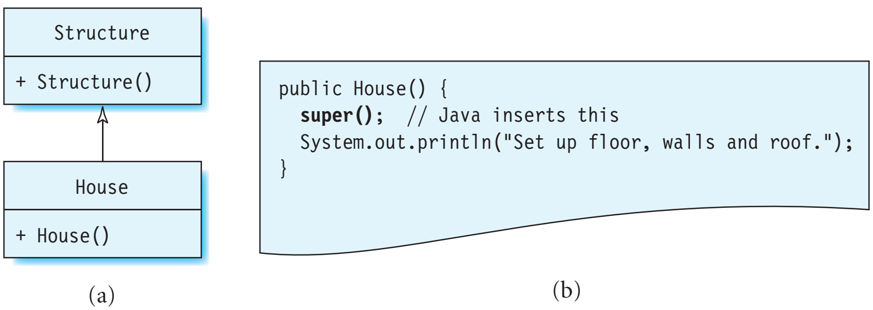
```
package inheritance;

public class House extends Structure {
  // constructor without parameters
  public House() {
    System.out.println("Set up floor, walls and roof.");
  }

  public static void main(String[] args) {
    // create a house
    House myHouse = new House();
  }
}
```

**Figure 6–13**

**(a) The classes `Structure` and `House`. (b) Java inserts the `super` statement automatically to call `Structure`'s constructor.**



When you run this program, it produces the following output:

```
Build foundation.
Set up floor, walls and roof.
```

The first line of the output shows that the constructor in `Structure` has been invoked. But who calls this constructor? This is what happens: *Java inserts a call to the superclass constructor inside* `House`*'s constructor using the* `super` *keyword*, as shown in Figure 6–13(b). This calls `Structure`'s constructor to create an object of this class first, and then `House`'s constructor will execute. The reason for this is that the parent must be created before the child, which is logical.

`House` calls `Structure`'s constructor; thus this constructor must be declared as `protected` or `public`. Otherwise, it is not visible to `House`, which will result in an error because an object of `Structure` cannot be created.

Superclass constructors should not be made private.

The observant reader will note that a `super` statement is inserted automatically in `Structure`'s constructor. However, given that `Structure` does not extend another class, which constructor is called? We explain this next.

### 6.8.2 `Object`: The Granddaddy of All Classes

Java contains a class called `Object` from which *all* classes are derived. `Object` does not have a superclass. Even if a class does not declare that it is derived from another class, it is implicitly derived from `Object`.

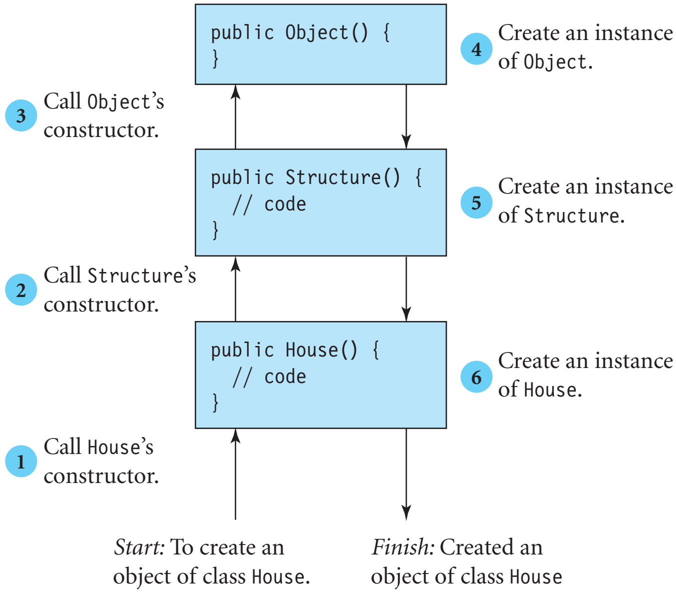
Thus, `Object` is the *parent* of `Structure` and the *grandparent* of `House`. We could have also written:

```
public class Structure extends Object {
  // code for Structure goes here
}
```

There can be any number of intermediate classes between `Object` and another class.

`Object` contains a constructor without parameters. The `super` statement in `Structure`'s constructor calls this constructor of `Object`. (There is no `super` statement in `Object`'s constructor.) The sequence of calls needed to create an object of `House` is shown in Figure 6–14. First, `House`'s constructor is called. The `super` statement is executed here, and the constructor in its superclass `Structure` is called. Similarly, after the `super` statement in `Structure`'s

**Figure 6–14**
**Sequence in which constructors are called to create an instance of House.**




constructor is executed, `Object`'s constructor is called. After this, there are no more constructors to call. The objects are now created in the order `Object`, `Structure`, and finally `House`.

To create an object of a class (say, `House`), all superclass constructors are called, up to the topmost class `Object`. Then the constructors are executed in reverse order, from top to bottom; that is, starting from the constructor `Object` down to the constructor `House`. This process is known as **constructor chaining**. Figure 6–14 shows an example of constructor chaining.

### 6.8.3 A Program with an Error

The `super` statement *cannot* call a constructor with parameters. Additionally, it *can* call the default constructor of the superclass *only* if there is no other constructor in the superclass. To see why this might create an error, let us write a class `AnotherStructure` with a constructor that takes an argument:

```
package inheritance;

public class AnotherStructure {
  // constructor with parameters
  public AnotherStructure(String type) {
    System.out.println("Build foundation of type " +type);
  }
}
```

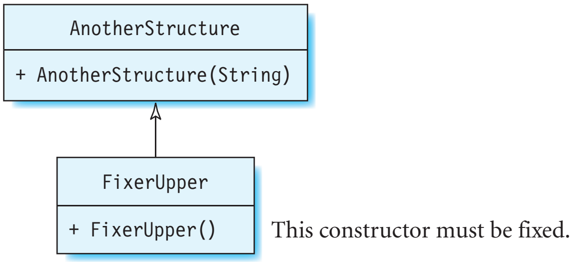



**Figure 6–15**
**The implicit super statement in FixerUpper's constructor cannot call the AnotherStructure constructor.**

Next, we will write a class called FixerUpper (see Figure 6–15) that has an error in its constructor:

```
package inheritance;

public class FixerUpper extends AnotherStructure {
  // this constructor needs to be fixed!
  public FixerUpper() {
    System.out.println("Set up floor, walls and roof.");
  }
}
```

The program will not compile if you add this main method to create an instance of FixerUpper:

```
public static void main(String[] args) {
  FixerUpper fixerupper = new FixerUpper();
}
```

So, what goes wrong when an object of FixerUpper is to be created? The (implicit) super statement in FixerUpper's constructor cannot call the constructor with parameters in the superclass AnotherStructure. It cannot call the default constructor because there is already a constructor present in AnotherStructure. In this case, to create an object of FixerUpper, the programmer must call the constructor of AnotherStructure *explicitly*, as explained next.

### 6.8.4 The Programmer Explicitly Calls a Constructor

The programmer can call a superclass constructor directly, or can call another constructor within the class itself. If the programmer wants to call a superclass constructor with parameters, the programmer must make this call explicitly. This call is made by supplying arguments to the super statement:

```
super(argument 1, argument 2, ..., argument n);
```

The constructor in the superclass whose parameter list matches these arguments is then invoked.

Modify `FixerUpper`'s constructor to call explicitly the superclass constructor `AnotherStructure(String)` in `AnotherStructure`, as shown here:

```
// FixerUpper fixed!
public FixerUpper() {
  super("Slab"); // calls the superclass constructor correctly
  System.out.println("Set up floor, walls and roof.");
}
```

`super` has a `String` argument; as a result, it calls the matching constructor in `AnotherStructure` with a `String` parameter. The program compiles and runs correctly now.

> If the superclass does not have a constructor without parameters, but contains constructors with parameters, the programmer should explicitly call the superclass constructor.

Instead of calling a superclass constructor, the programmer can call another constructor in the same class by using the keyword `this`. We saw examples of using `this` in constructors in Chapter 5. Here, we will briefly review the process again.

The keyword `this` takes arguments and calls a constructor in the class whose parameters match the data type and order of these arguments. Let us add a new constructor to `FixerUpper` that takes a `String` parameter:

```
FixerUpper(String value) {
  super(value); // calls the AnotherStructure(String) constructor
  System.out.println("Set up floor, walls and roof.");
}
```

The constructor without arguments in `FixerUpper` can be modified to call the preceding constructor using `this`:

```
FixerUpper() {
  this("Slab"); // calls the FixerUpper(String) constructor
}
```

Next, we will build the `Vehicle` and `Car` classes further by adding overloaded constructors to these classes.

**Adding Constructors to `Vehicle`** Let us add two overloaded constructors to the `Vehicle` class. The first constructor without parameters sets the fields `x` and `y` (representing the *x*- and *y*-coordinates of the `Vehicle` in the window) to 0. The second constructor updates `x` and `y` to values passed in as arguments. The updated code for `Vehicle` is shown here:

```
public class Vehicle {
  // vehicle's position
  protected float x, y;

  // constructor updates x and y to specific values
  public Vehicle() {
    this(0, 0);
  }

  // constructor updates x and y to values passed in as arguments
  public Vehicle(float xValue, float yValue) {
    x = xValue;
    y = yValue;
  }

  // method to draw shape of Vehicle
  protected void drawShape(Graphics2D myGraphics) {
  }
}
```

A constructor is also added to `Car`. Objects of this class will be positioned at the specified coordinates in the window:

```
// constructor updates x and y to specific values
public Car() {
  super(30, 300);
}
```

Similarly, the constructor for `Airplane` is:

```
// constructor updates x and y to specific values
public Airplane() {
  super(100, 400);
}
```

Note that the programmer explicitly calls `Vehicle`'s constructor here using the `super` keyword with arguments.

In the next section, we discuss abstract classes and their use.

## 6.9 Abstract Classes

Consider the class `Vehicle` described earlier in this chapter. Should we create an object of this class? The answer is no, because a vehicle has no particular shape or size; it is simply a generic term specifying a mode of transport. Let us write a new class called `Subject` (see Figure 6–16) with two subclasses, `Science` and `English`. Does it make sense to create an object of class `Subject`?

A class that should not be instantiated can be made **abstract**. We do this by prefixing the keyword `abstract` before the class declaration. Let us make `Subject` an abstract class:

```
public abstract class Subject {
}
```

This means an object of this class cannot be created:

```
Subject subject = new Subject(); // error! Subject is an abstract class
```

What is the purpose, then, of an abstract class if it cannot be instantiated? *An abstract class is used primarily to create subclasses.* An object of `Science` can be created as follows:

```
public class Science extends Subject {
  public static void main(String[] args) {
    Science sci = new Science(); // okay
  }
}
```

Like any parent class, the abstract class contains methods that can be inherited by these subclasses depending on the access modifiers used for these methods. However, an abstract class *can* also contain special methods called **abstract methods**. Abstract methods do not have a body. They are declared using the keyword `abstract`. The subclass can **implement** an

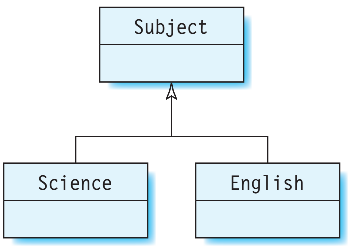



**Figure 6–16**
**It is pointless to make an object of class `Subject`.**

abstract method by providing a body for it. For example, let us add an abstract method called getSyllabus to Subject:

```
public abstract class Subject {
  public abstract void getSyllabus(); // abstract method
}
```

In an abstract method, only the declaration is provided; there is no body. Thus, the following declaration of getSyllabus would be incorrect because a body is specified by including braces:

```
public abstract void getSyllabus() {}; // error!
```

Now, the subclasses Science and English can provide a body for this method getSyllabus:

```
public class Science extends Subject{
  public void getSyllabus() {
    /* some code */
  }
}
```

A subclass must **implement** *all* abstract methods in its parent class; otherwise, it must be declared as abstract. This is another reason for creating an abstract class—*to force its subclasses to implement the abstract methods.* For example, if the class Science did not implement the method getSyllabus, it would have to be declared abstract, or a compilation error occurs.

Earlier in the chapter, we discussed how polymorphism can be achieved through *method overriding*. Polymorphism can be achieved using *abstract methods* as well. Suppose that we have declared a reference variable subject of type Subject. Then, the following statements call the getSyllabus method of the subclass object that subject references at the time the program is run:

```
Subject subject;
subject = new Science();
// gets the Science syllabus
subject.getSyllabus();

subject = new English();
// gets the syllabus of the subject English
subject.getSyllabus();
```

A method declared as abstract should not be made private because subclasses cannot implement the private methods of a superclass. Another restriction is that the implemented method cannot be less visible than the corresponding superclass method (see Table 6–1).

## Example 3

In this example, you will learn how to create a star shape by extending the `java.awt.Polygon` class. Java 2D's `Polygon` class is used to construct polygons. A constructor and method in this class are shown in Figure 6–17.

Recall that a *vertex* is a point where two sides meet. For example, the following code draws a line joining points (10, 20) and (30, 40) of a polygon called `p`:

```
p.addPoint(10, 20);
p.addPoint(30, 40);
```

This example is broken into three parts.

a. Using this class, create a polygon that connects the vertices *A*, *B*, *C*, and *D*, where *A* = (100, 200), *B* = (50, 300), *C* = (75, 400), and *D* = (200, 150). Display the polygon in a window.

### *Solution:*

```
package inheritance;

import java.awt.*;
import com.programwithjava.basic.DrawingKit;

public class PolygonDemo {
  public static void main(String[] args) {
    // store x- and y-coordinates of points A, B, C and D
    int xA = 100, yA = 200, xB = 50, yB = 300, xC = 75, yC = 400,
    xD = 200, yD = 150;

    DrawingKit dk = new DrawingKit("Polygon");
    Polygon p = new Polygon();
    p.addPoint(xA, yA);   // add point A
    p.addPoint(xB, yB);   // add point B
    p.addPoint(xC, yC);   // add point C
    p.addPoint(xD, yD);   // add point D
```

**Figure 6–17**

**A constructor and method in the `Polygon` class.**

| Polygon | |
|---|---|
| `Polygon()` | Constructor to create a polygon with no sides. |
| `addPoint(int x, int y)` | Method adds a vertex at the point (x, y) to the polygon. |

```
    dk.draw(p);
  }
}
```

The polygon is displayed in the window shown in Figure 6–18.

b. Write an abstract class called `Star` that is derived from the `Polygon` class. `Star` contains three integer fields `x`, `y`, and `s` that represent the *x*- and *y*-coordinates and length of a *side*, respectively. It also contains the following constructor and method:

**`public Star(int xstart, int ystart, int length)`**—a constructor that initializes the values of fields `x`, `y`, and `s` to the given arguments.

**`public abstract void drawShape(Graphics2D g)`**—an abstract method to draw a star shape.

### *Solution:*

```
package inheritance;
import java.awt.*;

public abstract class Star extends Polygon {
  protected int x;  // x-coordinate
  protected int y;  // y-coordinate
  protected int s;  // length of a side
```

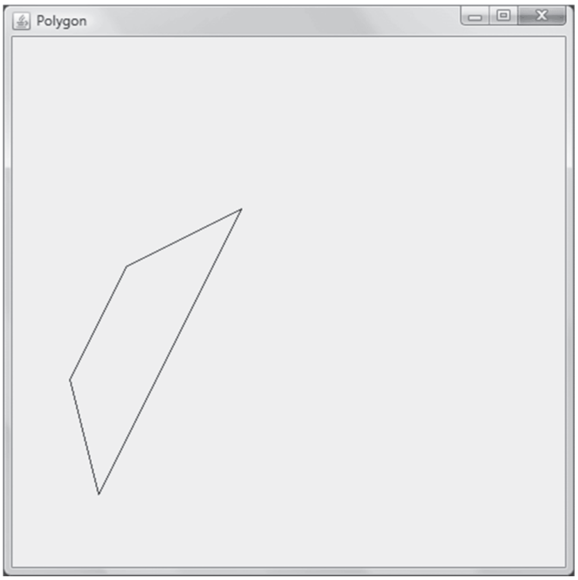


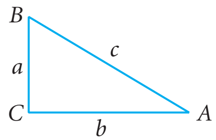

**Figure 6–18**

**Drawing a polygon using the `addPoint` method in class `Polygon`.**

```
  public Star(int xstart, int ystart, int length) {
    x = xstart;
    y = ystart;
    s = length;
  }
  public abstract void drawShape(Graphics2D myGraphics);
}
```

c. Write a class called `FivePointStar` to create a five-pointed star. This class is derived from class `Star`. It implements the method `drawShape` to create the desired shape. The programmer gives the star's position and the length of a side as arguments to its constructor. Add a `main` method to test the class.

***Solution:*** First, we review some results from trigonometry. In a right triangle, the sine of an angle is equal to the opposite side divided by the hypotenuse. The cosine of an angle in a right triangle is equal to the adjacent side divided by the hypotenuse. For example, consider a right triangle *ABC* with a right angle at angle *C*:




$\sin A = a/c$

$\cos A = b/c$

You can obtain the values of sin *B* and cos *B* similarly, so that $\sin B = \cos A$ and $\cos B = \sin A$. Another result is that the sum of the interior angles in a polygon with *n* sides is equal to $180 * (n - 2)$. We can use this result to calculate the angles in the star, as shown in Figure 6–19.

Consider the pentagon *JBDFH* in Figure 6–19. The sum of its interior angles is 540°, which means that each angle is 108°. Using this result, the internal angles of the triangle *AJB* are calculated to be 36°, 72°, and 72°. Suppose that the coordinates of point *A* are $(x, y)$. Drop a perpendicular line *AK* on the side *JB*. Assume that $JK = a$ and $AK = b$. Applying these results gives $a = s * sin\ 18$ and $b = s * cos\ 18$. The coordinates of point *J* are $(x - a, y + b)$. Similarly, calculate the coordinates of the other points.

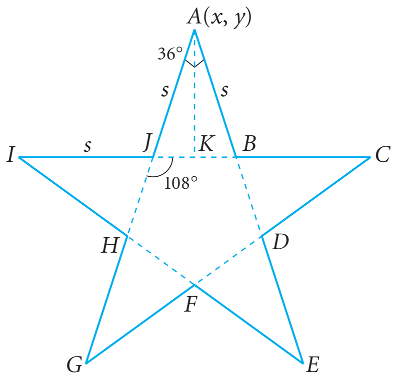



**Figure 6–19**
**Calculating the coordinates of the vertices of a five-pointed star.**

The `Math` class contains the `cos` and `sin` methods to calculate the cosine and sine values of the angles. Note that the arguments to these methods should be specified in radians. There are pi (also denoted as $\pi$) radians in 180°, so we can convert an angle into radians from degrees by multiplying it with pi/180. The `Math` class contains the constant `PI` to define the value of pi (which is approximately 3.14159):

```
static final double PI;
```

To use this value in your program, you write it as `Math.PI`. The following program uses the `addPoint` method in `Polygon` to create the star shape:

```
package inheritance;
import java.awt.*;
import com.programwithjava.basic.DrawingKit;

public class FivePointStar extends Star {

  private static final double RADIANS_PER_DEGREE = Math.PI/180;


  public FivePointStar(int xstart, int ystart, int length) {
    // call the superclass constructor explicitly
    super(xstart, ystart, length);
  }

  // this method implements the abstract drawShape method of Star
  public void drawShape(Graphics2D myGraphics) {
    int angle = 18;       // internal angle of 18 degrees
```

```
    int a = (int) (s * Math.sin(angle * RADIANS_PER_DEGREE));
    int b = (int) (s * Math.cos(angle * RADIANS_PER_DEGREE));
    int c = (int) ((s + 2 * a) * Math.sin(angle * RADIANS_PER_DEGREE));
    int d = (int) ((s + 2 * a) * Math.cos(angle * RADIANS_PER_DEGREE));
    int e = (int) (2 * (s + a) * Math.sin(angle * RADIANS_PER_DEGREE));
    int f = (int) (2 * (s + a) * Math.cos(angle * RADIANS_PER_DEGREE));
    int g = (int) (s * Math.sin(2 * angle * RADIANS_PER_DEGREE));
    addPoint(x, y);                         // Point A
    addPoint(x + a, y + b);                 // Point B
    addPoint(x + s + a, y + b);             // Point C
    addPoint(x + c , y + d);                // Point D
    addPoint(x + e, y + f);                 // Point E
    addPoint(x, y + f - g);                 // Point F
    addPoint(x - e, y + f);                 // Point G
    addPoint(x - c , y + d);                // Point H
    addPoint(x - s - a, y + b);             // Point I
    addPoint(x - a, y + b);                 // Point J
    myGraphics.draw(this);
  }

  public static void main(String[] args) {
    // draw the star at location (230, 180) with side of length 20
    int x = 230, y = 180, length = 20;

    DrawingKit dk = new DrawingKit("Five Point Star");
    Graphics2D myGraphics = dk.getGraphics();
    Star s = new FivePointStar(x, y, length);
    s.drawShape(myGraphics);
  }
}
```

Compile and run this program to see a star shape appear in the window at the specified location, as shown in Figure 6–20.

Note another use of the keyword `this` in the following statement:

```
myGraphics.draw(this);
```

The keyword `this` refers to the object that invokes this method, which is of type `Star`. The `Star` class extends `Polygon`, which is of type `Shape`, and therefore an instance of `Star` is of type `Shape` as well. (You will learn about `Shape` in Chapter 8, *Interfaces and Nested Classes*.) For this reason, an instance of `Star` can be passed as an argument to `myGraphics`'s `draw` method, which takes an argument of type `Shape`.


**Figure 6–20**
**A five-pointed star.**

Experiment with the program by creating stars of different sizes at various positions. ■

### 6.9.1 The `Calendar` Class

The Java API contains the abstract class `Calendar`, which contains several methods for working with dates and times. The class `GregorianCalendar` extends the `Calendar` class. Although many different types of calendars are followed around the world, such as Chinese, Indian, and Japanese, the one most commonly used is the Gregorian calendar, and it is the only calendar implementation provided in the Java API. Note that because `Calendar` is an abstract class, you cannot instantiate it.

A few fields and methods in the `GregorianCalendar` class are shown in Figure 6–21. The `GregorianCalendar` class inherits the constant fields shown in this figure from the `Calendar` class. These fields are used as arguments to the methods in this class. Other constant fields inherited from `Calendar` include those representing the months of a year (`JANUARY`, `FEBRUARY`, . . ., `DECEMBER`), and the days of the week (`MONDAY`, `TUESDAY`, . . ., `SUNDAY`). The methods `add` and `getMaximum`, declared as `abstract` in `Calendar`, are implemented in this class.

**Figure 6–21**
**Some fields and methods in the `GregorianCalendar` class.**

| GregorianCalendar | |
|---|---|
| `static final int YEAR` | Constant field representing a year. |
| `static final int MONTH` | Constant field representing a month. |
| `static final int DATE` (or `DAY_OF_MONTH`) | Both fields represent the day of the month. |
| `static final int DAY_OF_WEEK` | This constant represents a day of the week. |
| `static final int HOUR` | This constant represents the 12-hour time. |
| `static final int MINUTE` | Constant representing minutes in time. |
| `static final int SECOND` | Constant representing seconds. |
| `static final int AM_PM` | Constant representing AM/PM. |
| `GregorianCalendar()` | Constructor. |
| `void add(int field, int amount)` | Adds the specified amount to the given field. |
| `int get(int field)` | Returns the value of the given field. |
| `int set(int field, int value)` | Sets a field to the given value. |
| `int set(int year, int month, int date)` | Sets the values of the `YEAR`, `MONTH`, and `DATE` fields. |
| `int getMaximum(int field)` | Maximum possible value of a field; for example, number of days in a month is 31. |
| `int getActualMaximum(int field)` | Actual maximum value of a field; for example, number of days in February 2009 is 28. |

Examine the complete API for `GregorianCalendar`, and identify the inherited, overriding, and implemented methods in this class. Also, determine which new fields and methods (that is, those not inherited from `Calendar`) have been added to it.

## Example 4

This example shows how to use the `GregorianCalendar` class.

```
package inheritance;
import java.util.*;
```

```
public class CalendarDemo {
  public static void main(String[] args) {
    // create a new GregorianCalendar
    Calendar calendar = new GregorianCalendar();

    // set the date to July 16, 2008;
    // note that January = 0, February = 1,..., December = 11
    calendar.set(2008, 06, 16);

    // add one to the month and print it out
    calendar.add(Calendar.MONTH, 1);
    System.out.println("Month = " +calendar.get(Calendar.MONTH));

    // subtract 10 from the date and print it out
    calendar.add(Calendar.DATE, -10);
    System.out.println("Day = " +calendar.get(Calendar.DATE));

    // print maximum number of days in any month
    System.out.println("Maximum days in a month = "
+calendar.getMaximum(Calendar.DAY_OF_MONTH));

   // change the year to 2009, and month to February
   // then print out maximum number of days in February 2009
    calendar.set(Calendar.YEAR, 2009);
    calendar.set(Calendar.MONTH, Calendar.FEBRUARY);
    System.out.println("Maximum days in February 2009 is "
+calendar.getActualMaximum(Calendar.DAY_OF_MONTH));
   }
}
```

The program output is:

```
Month = 7
Day = 6
Maximum days in a month = 31
Maximum days in February 2009 is 28
```

The `set` method sets the `YEAR`, `MONTH`, and `DATE` fields to 2008, 6, and 16, respectively. The first call to the `add` method increments `MONTH` by 1, and the second call to `add` decrements `DATE` by 10.

Note the the `getMaximum` method takes the constant field `Calendar.DAY_OF_MONTH` to print out the maximum number of days in any month. On the other hand, the `getActualMaximum` method uses the month and year that is currently set on the calendar, so that it returns the number of days in February 2009 in this example. As an exercise, use other fields as arguments to these methods and observe how the outputs change. ■

**`Vehicle` as an `abstract` Class** We have created objects of the `Car` and `Airplane` classes, but not of the `Vehicle` class. The reason is that this class represents a generic vehicle without a specific form or shape. Therefore, this class can be made abstract:

```
public abstract class Vehicle {
  // rest of the code is unchanged
}
```

There is no code in method `drawShape` of `Vehicle`. By making this method abstract, we can force all subclasses of `Vehicle` to implement it. Thus, the updated code for `Vehicle` is:

```
package inheritance;
import java.awt.*;

public abstract class Vehicle {
  protected float x = 30, y = 300; // vehicle's position

  // constructor updates x and y to specific values
  public Vehicle() {
    this(0, 0);
  }

  // constructor updates x and y to values passed in as arguments
  public Vehicle(float xValue, float yValue) {
    x = xValue;
    y = yValue;
  }

  // method to draw shape of Vehicle
  protected abstract void drawShape(Graphics2D myGraphics);
}
```

The class `Car` already implements the `drawShape` method, but `Airplane` does not because it is not necessary for a subclass to override a method in the superclass. However, now that `drawShape` has been made abstract in the superclass, we must either implement this method in `Airplane` or declare it as an abstract class. The following drawShape method is added to the Airplane class:

```
// overriding method in class Airplane to draw the shape of an airplane
public void drawShape(Graphics2D myGraphics) {
    // body of the airplane
    Line2D line1 = new Line2D.Float(x, y, x-4, y-10);
    myGraphics.draw(line1);
    Line2D line2 = new Line2D.Float(x-4, y-10, x+120, y-95);
    myGraphics.draw(line2);
    QuadCurve2D curve1 = new QuadCurve2D.Float();
    curve1.setCurve(x+120, y-95, x+190, y-115, x+130, y-65);
    myGraphics.draw(curve1);
    Line2D line3 = new Line2D.Float(x+130, y-65, x+115, y-55);
    myGraphics.draw(line3);
    Line2D line4 = new Line2D.Float(x+81, y-36, x+14, y-3);
    myGraphics.draw(line4);
    Line2D line5 = new Line2D.Float(x, y, x+4, y);
    myGraphics.draw(line5);

    // left wing
    Line2D wing1 = new Line2D.Float(x+89, y-75, x, y-80);
    myGraphics.draw(wing1);
    Line2D wing2 = new Line2D.Float(x, y-80, x-10, y-70);
    myGraphics.draw(wing2);
    Line2D wing3 = new Line2D.Float(x-10, y-70, x+58, y-52);
    myGraphics.draw(wing3);

    // right wing
    Line2D wing4 = new Line2D.Float(x+110, y-60, x+165, y);
    myGraphics.draw(wing4);
    Line2D wing5 = new Line2D.Float(x+165, y, x+150, y+5);
    myGraphics.draw(wing5);
    Line2D wing6 = new Line2D.Float(x+150, y+5, x+76, y-40);
    myGraphics.draw(wing6);
    Line2D wing7 = new Line2D.Float(x+110, y-60, x+76, y-40);
    myGraphics.draw(wing7);

    // tail
    Line2D tail1 = new Line2D.Float(x+16, y-10, x+10, y+15);
    myGraphics.draw(tail1);
    Line2D tail2 = new Line2D.Float(x+10, y+15, x+5, y+18);
    myGraphics.draw(tail2);
    Line2D tail3 = new Line2D.Float(x+5, y+18, x+5, y-1);
    myGraphics.draw(tail3);
```

```
        Line2D tail4 = new Line2D.Float(x+5, y-1, x+16, y-10);
        myGraphics.draw(tail4);
        Line2D tail5 = new Line2D.Float(x+15, y-25, x-10, y-40);
        myGraphics.draw(tail5);
        Line2D tail6 = new Line2D.Float(x-10, y-40, x-20, y-35);
        myGraphics.draw(tail6);
        Line2D tail7 = new Line2D.Float(x-20, y-35, x, y-14);
        myGraphics.draw(tail7);
        Line2D tail8 = new Line2D.Float(x, y-14, x-15, y-14);
        myGraphics.draw(tail8);
        Line2D tail9 = new Line2D.Float(x-15, y-14, x-18, y-10);
        myGraphics.draw(tail9);
        Line2D tail10 = new Line2D.Float(x-18, y-10, x-2, y-6);
        myGraphics.draw(tail10);

        //cockpit
        QuadCurve2D cockpit= new QuadCurve2D.Float();
        cockpit.setCurve(x+120, y-95, x+125, y-75, x+140, y-100);
        myGraphics.draw(cockpit);

        // logo
        Ellipse2D logo = new Ellipse2D.Float(x-10, y-34, 10, 10);
        myGraphics.setPaint(Color.red);
        myGraphics.fill(logo);
        Line2D line6 = new Line2D.Float(x-1, y-8, x+145, y-80);
        myGraphics.draw(line6);
        Line2D line7 = new Line2D.Float(x+60, y-65, x+7, y-73);
        myGraphics.draw(line7);
        Line2D line8 = new Line2D.Float(x+110, y-35, x+150, y-6);
        myGraphics.draw(line8);
    }
```

Observe that we are using *upcasting* in the `drawShape` method of class `Airplane`. The `Line2D` class in the `java.awt.geom` package is the abstract superclass of the `Line2D.Float` and `Line2D.Double` classes. For this reason, using upcasting, we can write:

```
Line2D line1 = new Line2D.Float(x, y, x-4, y-10);
```

The object of type `Line2D.Float` is assigned to a reference variable of type `Line2D` instead of `Line2D.Float`. The `Ellipse2D` and `QuadCurve2D` classes are the abstract superclasses of `Ellipse2D.Float` and `QuadCurve2D.Float`, respectively, and you can use them similarly.

Now we are ready to test the `drawShape` method. Add a `main` method to `Airplane`. Inside this `main`, we randomly create a `Car` or `Airplane` object and assign it to a myVehicle variable of type Vehicle. Then the following statement calls the corresponding drawShape method of that object:

```
myVehicle.drawShape();
```

The main method is shown here:

```
public static void main(String[] args) {
  DrawingKit dk = new DrawingKit("Vehicle");
  Graphics2D myGraphics = dk.getGraphics();
  Vehicle myVehicle;
  // assign vehicleType a random number equal to 0 or 1
  Random rand = new Random();
  int vehicleType = rand.nextInt(2);
  // if vehicleType is 0, create an object of type Car
  if(vehicleType == 0)
    myVehicle = new Car();
  else
    myVehicle = new Airplane();
  // This will draw the corresponding shape of the
  // object based on its type determined at run time.
  myVehicle.drawShape(myGraphics);
}
```

Add these import statements to Airplane:

```
import java.awt.*;
import java.awt.geom.*;
import java.util.Random;
import com.programwithjava.basic.DrawingKit;
```

Compile and run the program as follows:

```
C:\JavaBook> javac -d bin src\com\programwithjava\basic\DrawingKit.java
src\inheritance\Vehicle.java src\inheritance\Car.java
src\inheritance\Airplane.java

C:\JavaBook> java -classpath bin inheritance.Airplane
```

Each time the program runs, a car or airplane is drawn in the window. The decision of whether a car or airplane will appear depends on the value of the random variable rand. If rand is 0, an instance of Car is created and assigned to the variable myVehicle; however, if rand is 1, an instance of Airplane is created and assigned to myVehicle. The drawShape method of the Car class is called if myVehicle references a car object, and it draws a car shape in the window; otherwise, the drawShape method of the Airplane class is called.


**Figure 6–22**
**Implementing the `drawShape` method in `Airplane` to draw an airplane in the window.**

This is an example of polymorphism. Here, polymorphism is achieved by *implementing* an *abstract* method. For example, if the object is of type `Car`, a car shape will be drawn when the method `drawShape` is called; otherwise, if the object has type `Airplane`, an airplane shape will be drawn.

Run the program several times. Either an airplane or a car is drawn inside the window. A result is shown in Figure 6–22.

## 6.10 The `final` Keyword

Java has a special keyword called `final` that can be used for creating **final methods**, **final classes**, and **final variables**. Each of these is explained in more detail next.

### 6.10.1 Final Methods

Final methods are methods whose implementations cannot be changed. When a method is declared as `final`, it cannot be overridden by a subclass method. For example, consider a class `E` that contains a final method called `computeSquare` that computes the square of its argument:

```
public class E {
  public final int computeSquare(int x) {
    return x*x;
  }
}
```

By declaring `computeSquare` as `final`, we ensure that it will not be overridden in a subclass. Thus, if subclass `F` attempts to override `computeSquare`, it causes a compilation error:

```
public class F extends E {
 // error: cannot override the final method computeSquare in E
  public int computeSquare(int x) {
    // some code goes here
  }
}
```

### 6.10.2 Final Classes

When a class is declared as `final`, it cannot be used to create subclasses. This also ensures that none of the methods in this class can be overridden; that is, all of its methods are final. For example, let us declare a new final class `G`:

```
public final class G {
  // some code
}
```

Then, an attempt to create a subclass of `G` results in an error:

```
// error: G cannot be subclassed
public class H extends G {
}
```

A class can be declared as final for either of the following two reasons:

1. *None of the methods in the class should be overridden.*
2. *The class should not be extended further.*

Examples of final classes in the Java API are `Float`, `Double`, `Integer`, `Boolean`, and `String`. The `Float` class, for example, is a specialized class for manipulating floating-point numbers, and its methods should not be changed.

### 6.10.3 Final Variables

Final variables are declared using the `final` keyword. A final variable can be assigned a value only *once*. This declares a variable called `ANGLE`:

```
final int ANGLE = 10;
```

Reassigning to `ANGLE` again results in an error:

```
ANGLE = 20; // error, ANGLE has been already assigned.
```

Final variables are useful in local and anonymous classes, which are discussed in Chapter 8, *Interfaces and Nested Classes.*

How do final variables differ from constants? Constants are *class* variables, because they are declared with the `static` modifier. This means that a single copy of this variable is shared among all instances of a class. Final variables, on the other hand, are *instance* variables.

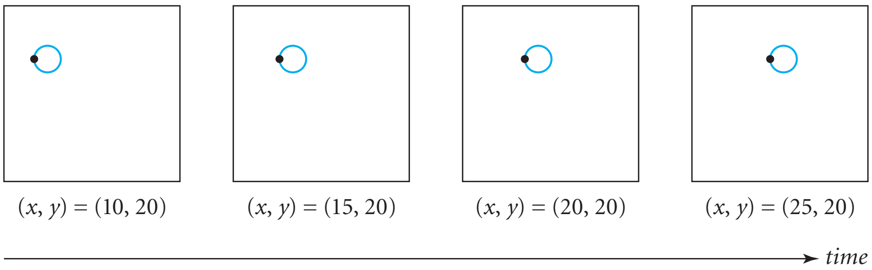
## 6.11 Animation

Animation is the art of making things appear to change with time. Thus, in animation, objects can appear to move or change shape. There are many different techniques and tools to do animation, but all of them use the same basic principle of displaying a sequence of pictures with incremental changes over a short time to produce an impression of continuous movement. For example, to make a ball appear to move, show a sequence of pictures of the ball with each at a location that is slightly different from the previous one, as shown in Figure 6–23.

When these pictures are shown quickly, one after another, the viewer perceives the ball as moving in one smooth, continuous motion. On the ball, the $(x, y)$ coordinates of a point are shown. You can see that the $x$-coordinate is increasing gradually, which gives the impression that the ball is moving to the viewer's right.

The smoothness of motion depends on how quickly the pictures are shown. Thus, if very few pictures (say, 5) are shown in one second, the movement can appear jerky. On the other hand, if too many pictures are shown, the motion can look blurry. The rate at which pictures are shown is known as frame rate and is abbreviated as **fps** (**frames per second**). For example, 24 fps means that 24 pictures are shown in one second. In theaters, movies are displayed at 24 fps.

**Figure 6–23**
**This sequence of pictures, shown one after another quickly, gives the impression that the ball is moving to the right.**



To give an impression of a moving object, you must follow these steps:

1. Clear the window.
2. Draw an image in the window at location (*x*, *y*).
3. Change the position (*x*, *y*).
4. Repeat Steps 1–3.

In the following sections, we will develop the code needed to animate the `Car` and `Airplane` objects. We will add a method called `step` to `Vehicle` and its subclasses. Inside `step`, the vehicle's position, represented by the coordinates (`x`, `y`), is changed. This method is abstract in `Vehicle`, and its implementation is provided in `Car` and `Airplane`.

**Adding Animation to Class `Vehicle`** The `step` method added to `Vehicle` is shown here:

```
// change the (x, y) position by a small amount
protected abstract void step();
```

The `step` method in `Car` is:

```
protected void step() {
  x += 2.5f;
}
```

The `step` method increases the value of `x` slightly to give the impression that the car is moving to the right. The `step` method of `Airplane` can be written similarly. Both the `x` and `y` coordinates are changed here:

```
// change the (x,y) position of the airplane
protected void step() {
  y = y - 1.5f;
  x = x + 2.5f;
}
```

Two classes, called `Controller` and `View`, are provided in the package `com.programwithjava.animation` on the CD-ROM. The `Controller` class controls the animation, and the `View` class displays the animation. These classes are explained in more detail in the next section, *Model View Controller Architecture*.

Create a directory called animation inside the src\com\programwithjava directory on your computer. Copy the Controller.java, View.java, and Vehicle.java files from the CD-ROM into the animation directory.

*Replace* the main method in the Airplane class with this method:

```
public static void main(String[] args) {
  Airplane topGun = new Airplane();
  View v = new View(topGun);
  Controller ct = new Controller(topGun, v);
  v.setVisible(true);
}
```

We will not use DrawingKit here, so you should remove this statement from Airplane:

```
import com.programwithjava.basic.DrawingKit;
```

Also, add this import statement to Airplane:

```
import com.programwithjava.animation.*;
```

Note that the Vehicle class written earlier is also present in the animation subpackage. Therefore, while compiling the program, only use the Vehicle class from the animation subpackage:

```
C:\JavaBook> javac -d bin src\inheritance\Airplane.java
src\com\programwithjava\animation\*

C:\JavaBook> java -classpath bin inheritance.Airplane
```

An instance of Airplane is created, and it is passed to the View and Controller classes. Compile and run the program with the View and Controller classes, which are briefly described in the next section. Now run your program to see the animation. You can press the "start" and "stop" buttons on the window at any time to start and pause the animation.

Run the program to see the plane fly. The output is shown in Figure 6–24.

**Model View Controller (MVC) Architecture** The *Model View Controller* architecture is used to separate an object's design (the **model**) from its display (the **view**). The advantage is that by decoupling the model from the view, we can change the code for the model without affecting the view, and vice versa. Consider the code in Airplane or Car. This code describes the design of these objects, and it represents the model in the MVC architecture. However, this code does not specify *how* the object will be displayed. We could choose to display the information about these objects


**Figure 6–24**
**Animating an instance of class `Airplane`.**

in different ways, such as by using text, or graphics, or both—these details are part of the view. The code in `View` creates the window, buttons, and so on, and represents the view in MVC. There could be many different views for the same model. The **controller** links the model to a view. The code in `Controller` passes the user actions (such as mouse clicks) from the view to the model, and the results are sent back to the view from the model. Both of these classes are in the `com.programwithjava.animation` package.

**`Controller` and `View` Classes** Here, the `Controller` and `View` classes contain code that can be used for animating instances of subclasses of `Vehicle`. In fact, you can write a new `Vehicle` subclass, and use the same `Controller` and `View` classes provided here to do the animation for this new subclass. However, it is not necessary to understand the code in these two classes for now. Therefore, if you want, you can skip the discussion that follows, and proceed directly to using the code given for `Controller` and `View` by adding the package `com.programwithjava.animation` to your code. You should, however, revisit this section after reading Chapter 9, *GUI Programming*.

Next, we briefly discuss two important methods in these classes. The class `Controller` contains one method called `actionPerformed`:

```
public void actionPerformed(ActionEvent e) {
  // move the model by one step
  model.step();
```

```
  // call the paintComponent method in view
  view.repaint();
}
```

Here, `model` is the object that is being animated, and `view` is the window in which this object is displayed. The `model.step` method changes the current position of the object by a small value. The `view.repaint` method calls the `paintComponent` method in the `View` class. This action clears the window and redraws the object. The work of drawing the object is done in the `paintComponent` method of `View` by the following statement:

```
model.drawShape(myGraphics);
```

A *swing timer* (called `timer`) is created in the `Controller` class and it calls the `actionPerformed` method periodically. Each time that it is called, this method clears the window and draws the image at a slightly different position. The rest of the code is used to create the animation for the two buttons (start/stop and pause/restart) when they are clicked.

The code for the `Controller` class is:

```
package com.programwithjava.animation;
import java.awt.event.*;
import javax.swing.*;

public class Controller implements ActionListener {
  // List the models and views that the controller interacts with
  private Vehicle model; // object being animated
  private View view;
  private Timer timer; // create a swing timer to run periodically

  public Controller(Vehicle m, View v) {
    model = m;
    view = v;
    timer = new Timer(30, this);

    // add listeners to view
    view.addStartListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
      // when the start button is pressed timer starts running
        timer.start();
      }
    });

    // add listeners to view
    view.addStopListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
```

```
        // when the stop button is pressed timer stops running
        timer.stop();
      }
    });
  }

  // action performed by timer
  public void actionPerformed(ActionEvent e) {
    // move the model by one step
    model.step();
    // call the paintComponent method in view
    view.repaint();
  }
}
```

The code for the `View` class is:

```
package com.programwithjava.animation;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class View extends JFrame {
  // Components
  private JButton startButton;  // button to start the animation
  private JButton stopButton;   // button to stop the animation

  // Model
  private Vehicle model;

  public View(Vehicle m) {
    model = m;
    // Lay the components
    JPanel panel = new JPanel() {
      public void paintComponent(Graphics g) {
      super.paintComponent(g);
      Graphics2D myGraphics = (Graphics2D) g;
      model.drawShape(myGraphics);
     }
   };
    // create the buttons
    startButton = new JButton("start");
    stopButton = new JButton("stop");

    // add the buttons to the panel
    panel.add(startButton);
    panel.add(stopButton);
```

```
    // add the panel to this window
    setContentPane(panel);
    panel.setOpaque(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(500, 500);
    setTitle("Animation");
  }

  // add a listener to the start button
  public void addStartListener(ActionListener listener) {
    startButton.addActionListener(listener);
  }

  // add a listener to the stop button
  public void addStopListener(ActionListener listener) {
    stopButton.addActionListener(listener);
  }
}
```

## Example 5

Write a `main` method to animate an object of `Car`.

***Solution:*** An instance of `Car` called `roadster` is created, and is passed to `View` and `Controller`. Add this import statement to `Car.java`:

```
import com.programwithjava.animation.*;
```

Again, because `DrawingKit` is not used in this example either, you must comment out this line:

```
import com.programwithjava.basic.DrawingKit;
```

Replace the `main` method in `Car` with the following:

```
public static void main(String[] args) {
  Car roadster = new Car();
  View v = new View(roadster);
  Controller ct = new Controller(roadster, v);
  v.setVisible(true);
}
```

Now run your program to see the animation.

You can use this technique to animate other types of vehicles. To do so, you create a subclass of `Vehicle`, and then define the `drawShape` and `step` methods in this class. Then, use a `main` method that is similar to the one previously described to run the animation. The `Controller` and `View` classes do not have to be changed. ■

## 6.12 Advanced Graphics (Optional)

In this section, we briefly discuss two useful features of `Java 2D`: the `GeneralPath` class and how to perform transformations using `Graphics 2D`. As an example, we show how to create rotating wheels for the `Car` object.

The `GeneralPath` class in the `java.awt.geom` package allows the programmer to build any kind of shape. Using the methods in this class, lines and curves can be joined together to form a regular or irregular shape. There are several overloaded constructors in this class. The constructor without parameters can be used to create a path called `myPath` as follows:

```
GeneralPath myPath = new GeneralPath();
```

The position at which the path should start is given by the `moveTo` method of `GeneralPath`:

> **`public void moveTo(float x, float y)`**—a method to add the point (x, y) to a path.

For example, the following statement starts `myPath` at the point (20, 30):

```
myPath.moveTo(20, 30);
```

Now, lines, curves, and other shapes can be added to this path by using the `append` method:

> **`public void append(Shape s, boolean connect)`**—a method to connect `s` to the path if `connect` is `true`.

`Shape` is an interface, and classes that draw lines and regular shapes (such as `Line2D.Float`, `Rectangle2D.Float`, and `Ellipse2D.Float`) implement this interface. Objects of all classes that implement this interface can be passed as arguments to this method. If the parameter `connect` is `true`, this shape is connected to the current position of the path with a line segment. For example, add `line1` to `myPath`:

```
Line2D.Float line1 = new Line2D.Float(100, 200, 300, 400);
myPath.append(line1, true);
```

This step adds `line1` to `myPath`. In addition, it also draws a line segment connecting the current position (20, 30) with the starting position of `line1` (100, 200).

If you want a curved line, you can draw a quadratic curve using the `quadTo` method to join points (*x1*, *y1*) and (*x2*, *y2*):

> **`public void quadTo(float x1, float y1, float x2, float y2)`**—a method that draws a quadratic curve joining points (`x1`, `y1`) and (`x2`, `y2`).

Another method to draw a curved line is the `curveTo` method, which draws a cubic curve connecting points (*x1*, *y1*) and (*x3*, *y3*) that passes through (*x2*, *y2*):

> **`public void curveTo(float x1, float y1, float x2, float y2, float x3, float y3)`**—a method that draws a cubic curve connecting points (`x1`, `y1`) and (`x3`, `y3`) and passing through (`x2`, `y2`).

**The `Wheel` Class** We next write a class called `Wheel` that contains a method called `createShape`. The constructor has four parameters: *x*- and *y*-coordinates of a reference point on the wheel, *diameter* and *thickness*. The `createShape` method draws a circular shape of the given diameter.

The constructor and `createShape` method for `Wheel` follow. The `createShape` method has a return type of `Shape`. The class `GeneralPath` implements the `Shape` interface. Therefore, the instance `path` in the `createShape` method is also of type `Shape`. We will discuss this interface in Chapter 8, *Interfaces and Nested Classes*.

```
package inheritance;

import java.awt.*;
import java.awt.geom.*;

public class Wheel {
  float x, y, width, height, angle, thickness;
```

```
  public Wheel(float x1, float y1, float diameter, int thick) {
    x = x1;
    y = y1;
    height = diameter;
    width = diameter;
    angle = 0;
    thickness = thick;
  }

// creates circular shape with two spokes representing a wheel
  protected Shape createShape(Graphics2D g2) {
    GeneralPath path = new GeneralPath();
    g2.setPaint(Color.black);
    Stroke s = new BasicStroke(thickness);
    g2.setStroke(s);
    Ellipse2D e1 = new Ellipse2D.Float(x, y, height, width);
    g2.draw(e1);
    path.append(e1, false);
    g2.setPaint(Color.white);
    g2.fill(e1);
    g2.setPaint(Color.black);
    Line2D l1 = new Line2D.Float(x, y+width/2, x+width, y+width/2);
    path.append(l1, false);
    Line2D l2 = new Line2D.Float(x+width/2, y, x+width/2, y+ width);
    path.append(l2,false);
    return path;
    }
  }
```

Next, we will show how the wheel can be rotated by using `Graphics2D` transformations.

### 6.12.1 Graphics2D Transformations

Some simple transformations can be performed on a `Graphics2D` object. These basic transformations are:

- *Rotate:* Rotate the object about a given point
- *Translation:* Move the object to a new point

- *Scale:* Make the object smaller or bigger
- *Shear:* Stretch the object in a nonuniform manner

For example, to rotate a `Graphics2D` object `g2`, the `rotate` method can be used:

```
g2.rotate(angle, x, y);
```

This produces a rotation by the specified angle with (*x*, *y*) as the center of the object, and where `angle` must be specified in radians. (Recall that 360 degrees = 2 * `PI` radians.) When successive transformations are applied to a `Graphics2D` object, their effect is additive. For example, suppose that the `Graphics2D` object `g2` is rotated by `PI` radians with a call to the following method:

```
g2.rotate(PI, x, y);
```

The next call to this method will rotate by 2 * `PI` radians instead of `PI`. To prevent this from occurring, the original `Graphics2D` context should be saved and restored after the transformation. You can do this by using a class called `AffineTransform`, as follows:

```
AffineTransform t = g2.getTransform();
// perform rotate, translate and other transformations
g2.setTransform(t);
```

The original graphics context is saved using `getTransform`, and it is restored using `setTransform`. The next section describes how to use this transformation in the `rotateWheel` method of the `Wheel` class.

**Adding Rotating Wheels to the Car** We will add two more methods to `Wheel`: `drawShape` and `step`. The method `drawShape` draws the `Wheel` instance after rotating it about its center by the specified angle:

```
public void drawShape(Graphics2D g2) {
  AffineTransform t = g2.getTransform();
  Shape shape = createShape(g2);

  // rotate the shape by the specified angle around its center.
  g2.rotate(angle, x + width/2, y + height/2);
  g2.draw(shape);
  g2.setTransform(t);
}
```

The step method takes the displacement in the wheel position as an argument and modifies its position and angle accordingly:

```
public void step(float displacement) {
  x += displacement;
  angle += displacement/width;
}
```

Some methods in Car will need to be changed to add the Wheel objects to Car. Add the following field to Car:

```
private Wheel wheel1, wheel2;
```

In the constructor for Car, create two new wheels using Wheel, in the same position as the previous ones:

```
public Car() {
  super(30, 300);
  wheel1 = new Wheel(x+37.5f, y+63, 50, 5);
  wheel2 = new Wheel(x+167.5f, y+63, 50, 5);
}
```

Modify the drawShape method of Car so that the wheels are drawn as well. Insert the following statements into this method:

```
// draw the wheels
wheel1.drawShape(myGraphics);
wheel2.drawShape(myGraphics);
```

The wheels must be moved forward by the same distance as the car. Modify the step method in Car to call the step method of Wheel:

```
protected void step() {
  float displacement = 2.5f;
  x += displacement;
  wheel1.step(displacement);
  wheel2.step(displacement);
}
```

Check to ensure that you have added the main method and made the other changes described in Example 5. Then, run the program as follows:

```
C:\JavaBook> javac -d bin src\inheritance\Car.java
src\inheritance\Wheel.java src\com\programwithjava\animation\*

C:\JavaBook> java -classpath bin inheritance.Car
```

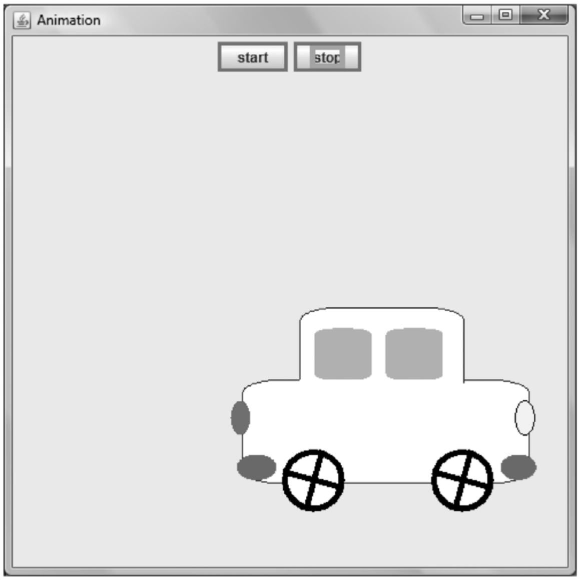



**Figure 6–25**
**Animating an instance of class Car with rotating wheels.**

You can now see the wheels rotate as the car moves. The animation is shown in Figure 6–25.

## 6.13 Computers in Business: Credit Card Finance Charge Calculator

Computers programs are used for processes in accounting, the stock market, and personal finance, among others. Accounting software is used by businesses to keep track of inventory, purchases and sales, and billing. Stock market software programs help investors manage investments. Personal finance software uses include tax calculations and money management. Statistical software can analyze large amounts of data to discern patterns (such as which items are most or least popular among shoppers), and also can predict future outcomes based on both current and historical data. In this section, we will discuss how credit card companies calculate consumer credit. We will write a program to calculate the finance charges for a given card **balance** based on its **APR** (**Annual Percentage Rate**), and the time needed to pay off the balance of the card.

First, we discuss some terminology and the basics needed to understand how finance charges are calculated. Whenever you make a purchase using a credit card, you put a **balance** on the card. Credit card companies charge interest—called a **finance charge**—on this balance. The finance charge is added to the existing balance, which continues to grow unless you pay it off. Finance charges are determined by two factors: the APR of the card and the calculation method used by the company. The APR is a numeric value provided by the company. Many different methods are used to calculate finance charges, but most credit card companies use what is called the **average daily balance method**. In this method, the finance charges are calculated for each day on the **average daily balance**, which is the sum of the charges made on the card during a billing period divided by the number of days in that billing period. The **daily periodic rate** (**DPR**) is used to calculate the daily finance charge, and is given by the (APR/100) divided by the number of days in a year:

$$\text{DPR} = \frac{\text{APR}}{100 \times 365}$$

The finance charges are calculated during a billing period $N$ (typically one month), as follows:

$$\text{Finance Charge} = \text{Average Daily Balance} \times \text{DPR} \times N$$

For example, suppose that you charge \$100.00, \$305.50, and \$50.00 on your credit card on the 1$^{st}$, 7$^{th}$, and 25$^{th}$ of January, respectively. What is the average daily balance? There is a balance of \$100 for the first 6 days, followed by a balance of \$405.50 for the next 18 days, and a balance of \$455.50 for the remaining 7 days. Use these figures to calculate the average daily balance:

$$\text{Average Daily Balance} = \frac{100 \times 6 + 405.50 \times 18 + 455.50 \times 7}{31} = \$357.66$$

The finance charge on this balance is calculated next. Suppose that the card has an APR of 15%. Plugging these values into the equation for finance charge gives the following:

$$\text{Finance Charge} = 357.66 \times \frac{15}{(100 \times 365)} \times 31 = \$4.56$$

The finance charge of \$4.56 is added to the balance of \$455.50, and so the new balance for February is \$460.06. The finance charge for February will be calculated similarly. Suppose that you made a payment of \$100 on February 1st. (For simplicity, assume that this payment is recorded immediately.) Then the finance charge would be calculated on the reduced balance of \$360.06.

### 6.13.1 The `BigDecimal` Class

Before writing the program, we discuss the `BigDecimal` class in the `java.math` package. This class is useful for high-precision operations, especially in financial calculations. You should not use the `double` or `float` types to store currency values in programs, because these numbers might not be stored accurately internally. For example, the value of `num` printed out here is 0.45999999999999996 instead of 0.46:

```
double num = 0.02 + 0.14 + 0.3;
System.out.println(num); // prints out 0.45999999999999996
double num1 = 0.101 + 0.001 + 0.201;
System.out.println(num1==0.303); // prints out false instead of true
```

Using a `float` can result in increasingly pronounced errors:

```
float val = 0.65f * 0.3f;
System.out.println(val); // prints out 0.19500001
```

This type of inaccuracy is unacceptable in financial calculations. The `BigDecimal` class stores values accurately with the **precision** (number of digits after the decimal point) that you specify. A field in this class, as well as some constructors and methods, are shown in Figure 6–26.

Let us recompute the previous result using `BigDecimal`. The following statement creates a `BigDecimal` object storing the number 0.02. Note that the number should be specified as a string:

```
BigDecimal num1 = new BigDecimal("0.02");
```

These two statements create two more `BigDecimals`, storing values 0.14 and 0.3:

```
BigDecimal num2 = new BigDecimal("0.14");
BigDecimal num3 = new BigDecimal("0.3");
```

The `add` method is used to add the numbers in these three objects together:

```
num1 = num1.add(num2).add(num3); // num1 = num1 + num2 + num3
```

**Figure 6–26**
**Some constructors and methods in the `BigDecimal` class.**

| BigDecimal | |
|---|---|
| `static BigDecimal ZERO` | A constant representing a `BigDecimal` object storing the value 0. |
| `BigDecimal(String num)` | Creates a `BigDecimal` object that stores the string `num` as a number. |
| `BigDecimal(int num)` | Creates a `BigDecimal` object that stores the `int num`. |
| `BigDecimal add(BigDecimal obj)` | Returns a `BigDecimal` object that stores the sum of the numbers in this object and `obj`. |
| `BigDecimal subtract(BigDecimal obj)` | Returns a `BigDecimal` object that stores the difference of the numbers in this object and `obj`. |
| `BigDecimal multiply(BigDecimal obj)` | Returns a `BigDecimal` object that stores the product of the numbers in this object and `obj`. |
| `BigDecimal divide(BigDecimal obj, int precision, int roundingMode)` | Returns a `BigDecimal` object that stores the result (with the specified precision and rounding mode) of dividing the number in this object by the number in `obj`. |
| `BigDecimal setScale (int precision, int roundingMode)` | Sets the number of digits after the decimal point (precision) and the rounding mode. |
| `String toString()` | Displays the number stored in this object. |
| `int compareTo(BigDecimal obj)` | Compares the number stored in this object with that in `obj`. Returns 1, 0, or –1, depending on whether the `BigDecimal` value is greater than, equal to, or less than that of `obj`. |

The resulting value in `num1` can be printed out using the `toString` method. This will print out the correct value of 0.46:

```
System.out.println(num1.toString());
```

We can use the methods `subtract` and `multiply` similarly. Like the `add` method, each of these methods also takes an argument of type `BigDecimal`.

The `compareTo` method compares the numbers stored in two objects of type `BigDecimal`. The following code segment shows how to check whether the `BigDecimal num1` is equal to 0:

```
if (num1.compareTo(BigDecimal.ZERO) == 0)
  System.out.println("The two numbers are equal");
else if (num1.compareTo(BigDecimal.ZERO) < 0)
  System.out.println("num1 is less than 0");
```

```
else if (num1.compareTo(BigDecimal.ZERO) > 0)
  System.out.println("num1 is greater than 0");
```

The constant field ZERO in BigDecimal represents a BigDecimal object storing the value 0. This answer will print out:

```
num1 is greater than 0
```

BigDecimal also contains constant fields TEN and ONE to represent the values 10 and 1, respectively.

You can specify the number of digits after the decimal point and the rounding mode using the setScale method. To display a number with two decimal places that are rounded up, use the following:

```
BigDecimal num4 = new BigDecimal("1234.56789");
num4 = num4.setScale(2, RoundingMode.HALF_UP);
System.out.println(num4.toString()); // prints out 1234.57
```

Other rounding modes include ROUND_UP (rounds upward toward 0) and ROUND_DOWN (round downward, away from 0).

Nonterminating numbers (with an infinite number of digits) cannot be represented exactly as BigDecimal numbers. For example, dividing 1 by 3 results in a nonterminating decimal and causes a run-time error:

```
BigDecimal one = new BigDecimal("1");
BigDecimal three = new BigDecimal("3");
BigDecimal nonterm = one.divide(three); // 0.3333...
```

Upon running the program, the following error message is issued:

```
Exception in thread "main" java.lang.ArithmeticException: Non-terminating
decimal expansion; no exact representable decimal result.
      at java.math.BigDecimal.divide(BigDecimal.java:1594)
```

In the divide operation, specify the scale and the rounding mode of the result. For example, the following statement will set the number of decimal places in the result to 20 and the rounding mode to HALF_UP:

```
BigDecimal term = one.divide(three, 20, RoundingMode.HALF_UP);
```

The result shows that the number stored in term has a precision of 20:

```
System.out.println(term.toString()); // displays 0.33333333333333333333
```

### 6.13.2 Program to Calculate Time to Pay Off Balance

In this section, we write a program to calculate the total time needed to pay off the balance on a credit card by making fixed monthly payments. We also calculate the total finance charges incurred over this period. Let us work out an example to explain the steps needed. Suppose that you have a credit card with an APR of 15% and a balance of $1000. You would like to make a payment of $500 on the first day of each month and put no new charges on the card, starting January 1, 2008 (a leap year), until the balance is fully paid off.

**Step 1:** Starting balance = $1000

Payment on January 1 = $500

New balance on January 1 = $500

No new charges are made to the card; thus the average daily balance is $500.

$$\text{Finance Charge} = 500 \times \frac{15}{(100 \times 366)} \times 31 = \$6.35$$

**Step 2:** Add the finance charge to the previous balance. The balance on February 1 = $506.35.

Payment on February 1 = $500

New balance on February 1 = $6.35

$$\text{Finance Charge} = 6.35 \times \frac{15}{(100 \times 366)} \times 29 = \$0.08$$

**Step 3:** The balance on March 1 = $6.43.

Payment on March 1 = $6.43

New balance on March 1 = $0.0

The total payment made in 3 months is $1006.43, of which the net finance charges were $6.43.

The algorithm and program for this problem are discussed next.

```
while balance > 0 {
  Calculate average daily balance after monthly payment is made at start of month
  Calculate DPR
  Calculate finance charge
  Print out the balance and finance charge for this month
  Update running totals of finance charges and monthly payments
  Increment month to next
  Add finance charge to balance to obtain new balance at start of this month
}
```

The class CreditCardInterestCalculator is declared as follows:

```
package inheritance;
import java.util.*;
import java.math.*;

public class CreditCardInterestCalculator extends FinancialCalculator {
  // monthly credit card payment
  private BigDecimal monthlyPayment;

  // starting month from which to calculate interest
  private int startMonth;

  // starting year from which to calculate interest
  private int startYear;

  // annual percentage rate
  private BigDecimal apr;

  // current balance
  private BigDecimal balance;

  // number of months taken to pay off balance
  private int numMonths;

  // monthly finance charge
  private BigDecimal financeCharge;

  // total finance charges until balance is paid
  private BigDecimal totalFinanceCharge;

  // total payments made until balance is paid off
  private BigDecimal totalPayment;

  // precision
  private int precision = 100;

  public CreditCardInterestCalculator() {
    calendar = new GregorianCalendar();
    totalFinanceCharge = new BigDecimal("0");
    totalPayment = new BigDecimal("0");
  }
  // methods for this class will be added here
}
```

This class extends the class FinancialCalculator, which is the superclass representing all types of calculators, such as mortgage and tax calculators. It contains a field of type Calendar and two methods to determine the number of days in a specific month and the number of days in a particular year (365 or 366). Both of these methods, and the Calendar field, will be inherited by the subclasses of this class. The CreditCardInterestCalculator class must implement the abstract methods getUserInput and compute:

```
package inheritance;
import java.util.*;

public abstract class FinancialCalculator {
  // calendar
  protected Calendar calendar;

  // returns the number of days for the current month set on calendar
  protected int getDaysInMonth() {
    return calendar.getActualMaximum(Calendar.DAY_OF_MONTH);
  }

  // returns the number of days in the current year set on calendar
  protected int getDaysInYear() {
    return calendar.getActualMaximum(Calendar.DAY_OF_YEAR);
  }

  protected abstract void getUserInput();
  protected abstract void compute();
}
```

The methods in class CreditCardInterestCalculator are described next. The following method calculates the average daily balance. It assumes that a payment is recorded at the start of the month and that no other purchases are made during that month. In the last month, the balance might fall below the monthly payment, in which case only the remainder is paid.

```
private BigDecimal calculateAverageDailyBalance() {
  // check if balance is less than monthlyPayment
  if (balance.compareTo(monthlyPayment) < 0)
    monthlyPayment = balance;
```

```
  // average daily balance is balance remaining after monthly payment
  // is made
  balance = balance.subtract(monthlyPayment);
  return balance;
}
```

This method calculates the daily periodic rate:

```
// Daily periodic rate (dpr) = APR/(100 * number of days in year)
private BigDecimal calculateDailyPeriodicRate() {
  BigDecimal percent = new BigDecimal("100");
  BigDecimal numDaysInYear = new BigDecimal(getDaysInYear());
  BigDecimal dpr = apr.divide(percent).divide(numDaysInYear, precision,
RoundingMode.HALF_UP);
  return dpr;
 }
```

This method calculates the finance charge on the balance for one month:

```
// finance charge = average daily balance * dpr * num days in month
private BigDecimal calculateMonthlyFinanceCharge(){
  BigDecimal averageDailyBalance = calculateAverageDailyBalance();
  BigDecimal dpr = calculateDailyPeriodicRate();
  BigDecimal numDaysInMonth = new BigDecimal(getDaysInMonth());
  financeCharge =
averageDailyBalance.multiply(dpr).multiply(numDaysInMonth);
  return financeCharge;
}
```

The following code implements the abstract `getUserInput` method in the `FinancialCalculator` class. It prompts the user to enter the card balance, the card APR, and the month and year when payments will begin:

```
public void getUserInput() {
  Scanner scanner = new Scanner(System.in);
  System.out.print("Enter balance on credit card (in dollars):");
  balance = new BigDecimal(scanner.next());
  System.out.print("Enter credit card APR (%):");
  apr = new BigDecimal(scanner.next());
  System.out.print("Enter your monthly payment (in dollars):");
  monthlyPayment = new BigDecimal(scanner.next());
  System.out.print("Enter the starting month and year[Example, 1 2009 for
January 2009]:");
  startMonth = scanner.nextInt();
  startYear = scanner.nextInt();
}
```

The following code implements the abstract compute method in the superclass. It calculates the total finance charges and the time to pay off the balance using the algorithm we discussed previously:

```
public void compute() {
  // initialize calendar
  calendar.set(startYear, startMonth - 1, 1);

  // print out table header
  System.out.println("Month    Year" +"     Balance ($)     "
+"Interest ($)");

  BigDecimal monthlyFinanceCharge;

  while(balance.compareTo(BigDecimal.ZERO) > 0){
    // calculate finance charges for each month
    monthlyFinanceCharge = calculateMonthlyFinanceCharge();

    // round monthlyFinanceCharge and balance up to two decimal places
    monthlyFinanceCharge = monthlyFinanceCharge.setScale(2,
RoundingMode.HALF_UP);
    balance = balance.setScale(2, RoundingMode.HALF_UP);

    // print out monthly finance charge and balance
    System.out.println(String.format("%3d     %5d   %10s    %10s ",
calendar.get(Calendar.MONTH)+1, calendar.get(Calendar.YEAR),
balance.toString(), monthlyFinanceCharge.toString()));

    // running total of credit card finance charges
    totalFinanceCharge = totalFinanceCharge.add(monthlyFinanceCharge);

    // running total of credit card payments
    totalPayment = totalPayment.add(monthlyPayment);

    // increment month by 1
    calendar.add(Calendar.MONTH, 1);
    numMonths++;

    // calculate new balance at the start of next month
    balance = balance.add(monthlyFinanceCharge);
  }
  // round up to two decimal places and print
  totalFinanceCharge = totalFinanceCharge.setScale(2,
RoundingMode.HALF_UP);
  totalPayment = totalPayment.setScale(2, RoundingMode.HALF_UP);
```

```
  System.out.println("Total payment in " +numMonths +" months: $"
+totalPayment.toString() + "   Total Finance Charges paid: $"
+totalFinanceCharge.toString());
}
```

We have specified a precision of 2, and the `HALF_UP` rounding mode in the `setScale` method. A different precision and rounding mode could be used, depending upon the application requirements.

Add the preceding methods to `CreditCardInterestCalculator`. Test the class using this `main` method:

```
public static void main(String[] args) {
  FinancialCalculator calc = new CreditCardInterestCalculator();
  calc.getUserInput(); // polymorphism
  calc.compute();      // polymorphism
}
```

A sample run of the program is shown here:

```
Enter balance on credit card (in dollars):1000
Enter credit card APR (%):15
Enter your monthly payment (in dollars):500
Enter the starting month and year[Example, 1 2009 for January 2009]:1 2008
Month    Year     Balance ($)     Interest ($)
  1      2008       500.00          6.35
  2      2008         6.35          0.08
  3      2008         0.00          0.00
Total payment in 3 months: $1006.43   Total Finance Charges paid: $6.43
```

The output matches the hand calculation. Run the program for other values of input to verify that it works correctly.

## 6.14 Summary

In this chapter, we discussed what inheritance is and how it can be used. Some important points to remember are:

- Inheritance allows a class to reuse code from its superclass and enables method overriding.
- There are many types of inheritance—single-level, multilevel, and hierarchical. Java does not support multiple inheritance.
- Access modifiers determine which fields and methods are inherited by a subclass.

- The keyword `super` can be used to call a superclass method or constructor.
- Polymorphism means that the behavior of an object is based on its type that is determined during run time. Polymorphism is achieved by overriding methods or implementing abstract methods.
- Constructors are not inherited. The first line of a constructor must be a call to another constructor. This can be done automatically by Java, or explicitly by the programmer, using the keywords `super` or `this`.
- Classes declared `abstract` cannot be instantiated.
- Abstract classes can contain abstract methods that do not have a body. A subclass must implement all abstract methods of its superclass; otherwise, it must be made abstract.
- Final methods cannot be overridden.
- Final classes cannot be subclassed.

## Exercises

1. Identify which of the following examples of inheritance are correct by determining whether the *is-a* relationship of each is true or false:
    a. Class `Dog` inherits from class `Animal`
    b. Class `Flower` inherits from class `Seed`
    c. Class `Sun` inherits from class `Star`
    d. Class `Planet` inherits from class `Earth`
    e. Class `Rectangle` inherits from class `GeometricalShapes`
    f. Class `Customer` inherits from class `Bank`
2. Explain each of the following:
    a. Single-level, multilevel, and hierarchical inheritance
    b. `super` keyword
    c. Upcasting
    d. Overridden method
    e. Hidden field
3. Explain briefly:
    a. Why is inheritance useful?

   b. What is polymorphism?
   c. Why are methods overloaded?
   d. When should a class be made abstract?
   e. Why are abstract classes useful?
   f. What is an abstract method?
4. Which of the following statements are true?
   a. Overloaded methods have different signatures.
   b. Overloaded methods can have the same signature if they have different return types.
   c. The `super` keyword can be used only in constructors, and not methods of a class.
   d. A class that contains an abstract method should be declared `abstract`.
   e. A class cannot be made abstract unless it contains an abstract method.
   f. A class cannot be both abstract and final.
   g. The methods in a final class can be overridden.
   h. Private methods in a class cannot be overridden.
   i. The `super` statement must always be the first statement in a constructor.
5. Predict the output of the following program without running it:

```
public class ClassA {
  protected int x = 10;
  public void printA() {
    System.out.println("x = " +x);
  }
}

public class ClassB extends ClassA {
  ClassB() {
    x = 20;
  }
}

public class ClassC extends ClassB {
  ClassC() {
    x = 30;
  }
```

```
public void printC() {
  System.out.println("x = " +x);
}

public static void main(String[] args) {
  ClassC c = new ClassC();
  c.printA();
  c.printC();
  }
}
```

Run the program to check your answer.

6. a. Write a class called Arachnid that contains a constructor without parameters. In the body of this constructor, add a statement to print the words “Executing Arachnid constructor.” Next, create a class called Spider that extends Arachnid. Similarly, add a constructor to this class with a statement that prints out “Executing Spider constructor.” Lastly, create a class called GardenSpider that extends Spider and has a constructor with a print statement. In a main method, create an object of GardenSpider. In which order are the constructors called?

   b. Add a protected field called numberOfLegs to Arachnid, and initialize it to 8 in its constructor. Add a method to GarderSpider called print-NumberOfLegs that displays the value of this field. Call this method in main. What is the output?

   c. Explain what happens if the numberOfLegs field in Arachnid is made:
   –private

   –package-private

7. Create a class called Account with the following fields: number, name, balance, and interestRate. Add acessor methods to display the value of each field in this class. Create another class called SavingsAccount that inherits from Account. Add three fields called day, month, and year. Add overloaded constructors to initialize the fields. Add three methods called deposit, withdraw, and computeInterest to this class, which are declared as follows:

```
public void deposit(BigDecimal amount);

public boolean withdraw(BigDecimal amount);
public BigDecimal computeInterest();
```

The computeInterest() method calculates the monthly interest using the formula:

$$\text{Monthly interest} = \frac{\text{balance} \times \text{interestRate}}{12}$$

Add at least two overloaded constructors to both SavingsAccount and Account. Create two instances of SavingsAccount in the main method and determine the interest on a given balance at the end of a year.

8. a. Create a class called Book with two private fields: name and cost of type String and float, respectively. Write a constructor in this class that takes two arguments and uses them to initialize these two fields. Add two accessor methods, getName and getCost, to Book to return the name and cost of the book.

   b. Create a class called Textbook that inherits from Book. Add a constructor to this class that takes a parameter of type String and another of type float. Use the arguments passed to this constructor to initialize the name and cost fields of Book.

   c. Write a main method to test the two classes. Create an instance of Textbook using the following statement:

   ```
   Textbook myBook = new Textbook("Java Programming", 100);
   ```

   Print out the name and cost of this book in the main method as follows:

   ```
   System.out.println(myBook.getName());
   System.out.println(myBook.getCost());
   ```

9. Create a class called CourtGame. Add a method to this class called playGame that prints out the class name along with a message. Create two subclasses of CourtGame called Tennis and Badminton.

   a. Override the playGame method in the Tennis class, but not in Badminton. Create an instance of Tennis and Badminton in the main method, and invoke the playGame method for each instance. Which playGame method is called for each instance?

   b. Now override the playGame method in the Badminton class. Rerun the program, and check which playGame method is called for the Badminton instance.

10. Create an abstract class called Bird. Write a method called chirp in this class that prints out the word "chirp." Create two classes, Goose and Mallard, which extend Bird, and override the chirp method in both

classes. In the `Goose` class, this method should print out the word "Honk," and in the `Mallard` class, it should print out "Quack." Write a `main` method to demonstrate polymorphism. In `main`, create a reference variable of type `Bird` as follows:

```
Bird bird;
```

Prompt the user to enter a number that is either 1 or 2. If the user enters a 1, this statement should call the `chirp` method of the `Goose` class; otherwise, it should call the method of the `Mallard` class:

```
bird.chirp();
```

a. Create a new class that extends `Bird` called `Crow`, but do not add an overriding method. Modify the `main` program so that if the user enters the number 3, the `chirp` method in `Crow` is called. What is the output of the program?

b. Make the `chirp` method in `Bird` abstract. Which changes must be made to the class `Crow`? Make the necessary changes and rerun the program. How does the output change?

11. Create a final class called `MyFinalClass`. Write a program to show that this class cannot be extended.

12. Write a class called `MortgageCalculator` that extends the `FinancialCalculator` class discussed in this chapter. The `compute` method of this class prints out the total interest paid on a mortgage. The monthly payment is computed using this formula:

$$\text{Monthly payment} = \frac{A \times r/n}{1-[1/(1+r/n)^{nT}]}$$

where $A$ is the mortage amount, $r$ is the interest rate, $n$ is the number of payments in a year, and $T$ is the term of the mortgage in years. The total interest paid is calculated by taking the product of the monthly payment, the number of payments in a year $n$, and the term $T$. Write a program to test this class.

## Graphical Programs

13. The following questions refer to the `Vehicle`, `Car`, and `Airplane` classes that were described in this chapter.

a. Create a new subclass of `Vehicle` called `Ship`. Add a constructor to this class. Also, add a method called `drawShape`, with the following declarations:

```
public void drawShape(Graphics2D myGraphics);
```

The method `drawShape` draws the shape of a ship. Use your imagination to decide what the ship should look like. Write a `main` method to test your class.

b. Add animation to the `Ship` class. To do so, include a new method called `step`:

```
protected void step(Graphics2D g2) {
  // your code to change the (x, y) coordinates of ship
}
```

Implement this `step` method so that the ship will move right to left across the window.

c. Write a `main` method in which you create an instance of `Ship`, `Controller`, and `View classes`. Pass this instance of `Ship` to the `Controller` and `View` classes. Run your program along with the classes in the `src\com\programwithjava\animation` package.

d. Which methods of `Vehicle` should not be made final? Explain your reasoning.

e. Add a new constructor to the `Ship` class. The constructor updates the `x` and `y` position of the ship to specific values that are passed as arguments to the constructor.

14. Write an `abstract` class called `Fish` that contains the following two `abstract` methods:

```
abstract void displayInformation();
abstract void drawShape(Graphics2D g);
```

Select any two fishes (say, shark and clownfish) that you want to use in your program, and create a class for each of them as a subclass of `Fish`. Add any fields and methods that are needed to store and modify information (such as type, size, weight, and interesting facts) about each fish. In each class, override the `displayInformation` method of `Fish` to print out this information on the console. Also, implement the abstract `drawShape` method in the subclasses to draw a fish of a particular type in a window. For example, suppose that you write classes `Shark` and `ClownFish`. Write a `main` method to test your program and verify polymorphic behavior as follows:

```
public static void main(String[] args) {
  DrawingKit dk = new DrawingKit();
  Graphics2D myGraphics = dk.getGraphics();
  Fish f;
  f = new Shark();
  f.drawShape(myGraphics); // shark shape should be displayed
  f = new ClownFish();
  f.displayInformation(); // print information about clown fish
}
```

Move the subclasses to a different package from the parent class. What should the access modifiers of the `getInformation` and `drawShape` methods in `Fish` be?

15. a. Write a class called `FourPointStar` that is derived from the `Star` class described in this chapter. Implement the abstract `drawShape` method of `Star` in this class to draw a four-pointed star shape.

    b. Repeat part (a) to create a six-pointed star instead. Write a class called `SixPointStar` that extends the `Star` class and implements the `drawShape` method of `Star`.

    c. Write a program to demonstrate polymorphism. In the `main` method, prompt the user to enter a number from 1 to 3. A four-, five-, or six-pointed star shape is then drawn on the screen depending on whether a 1, 2, or 3 is entered, respectively. An outline of this method is shown here:

```
public static void main(String[] args) {
// insert code to draw a window and get its graphics context
// prompt user to enter a number from 1 to 3 and store it in a
// variable called input
  Star s;
 if (input == 1)
  s = new FourPointStar();
else if (input == 2)
  s = new FivePointStar();
else if (input == 3)
  s = new SixPointStar();
```

```
else
  // print out an error message and exit
  s.drawShape(myGraphics);
}
```

16. Write a class called `Pentagon` to create regular polygons with 5 sides. (A regular polygon is a polygon whose sides are equal.) This class is derived from the `Polygon` class. The constructor for this class takes one argument of type `int` that represents the length of the side. Write a `main` method in this class to create and display three pentagons having sides of length 25, 50, and 100.

17. Write a class called `Octagon` to create regular polygons with 8 sides. This class is derived from the `Polygon` class. The constructor for this class takes one argument of type `int` that represents the length of the side. Write a `main` method in this class to create and display two octagons having sides of length 15 and 50. Write a program showing polymorphic behavior using the `Pentagon` and `Octagon` classes.

## Further Reading

We used the *average daily balance* method to calculate the finance charges. Creditors also use other methods such as the *unpaid balance method*. You can find more information about these methods in [3] as well as on the websites of various credit card companies.

## References

1. "The Java™ Tutorials." Web. <http://download.oracle.com/javase/tutorial/>.

2. Brinkmann, Ron. *The Art and Science of Digital Compositing: Techniques for Visual Effects, Animation and Motion Graphics, Second Edition*. Burlington, MA: Morgan Kaufmann/Elsevier, 2008. Print.

3. Bluman, Allan G. *Business Math Demystified*. New York: McGraw-Hill, 2006. Print.

4. Eckstein, Robert. "Java SE Application Design With MVC." *Oracle Technology Network*. March 2007. Web. <http://www.oracle.com/technetwork/articles/javase/index-142890.html>.

5. Eckel, Bruce. *Thinking in Java.* Upper Saddle River, NJ: Prentice Hall, 2006. Print.

6. Anderson, Julie, and Herve Franceschi. *Java 6 Illuminated: An Active Learning Approach.* Sudbury, MA: Jones and Bartlett, 2008. Print.

7. Sierra, Kathy, and Bert Bates. *Head First Java.* Sebastopol, CA: O'Reilly, 2005. Print.

8. Knudsen, Jonathan. *Java 2D Graphics.* Beijing: O'Reilly, 1999. Print.

9. *Java ™ Platform, Standard Edition 6, API Specification.* Web. <http://download.oracle.com/javase/6/docs/api/>.