Chapter 9 Inheritance

Topics

9.1 What Is Inheritance?

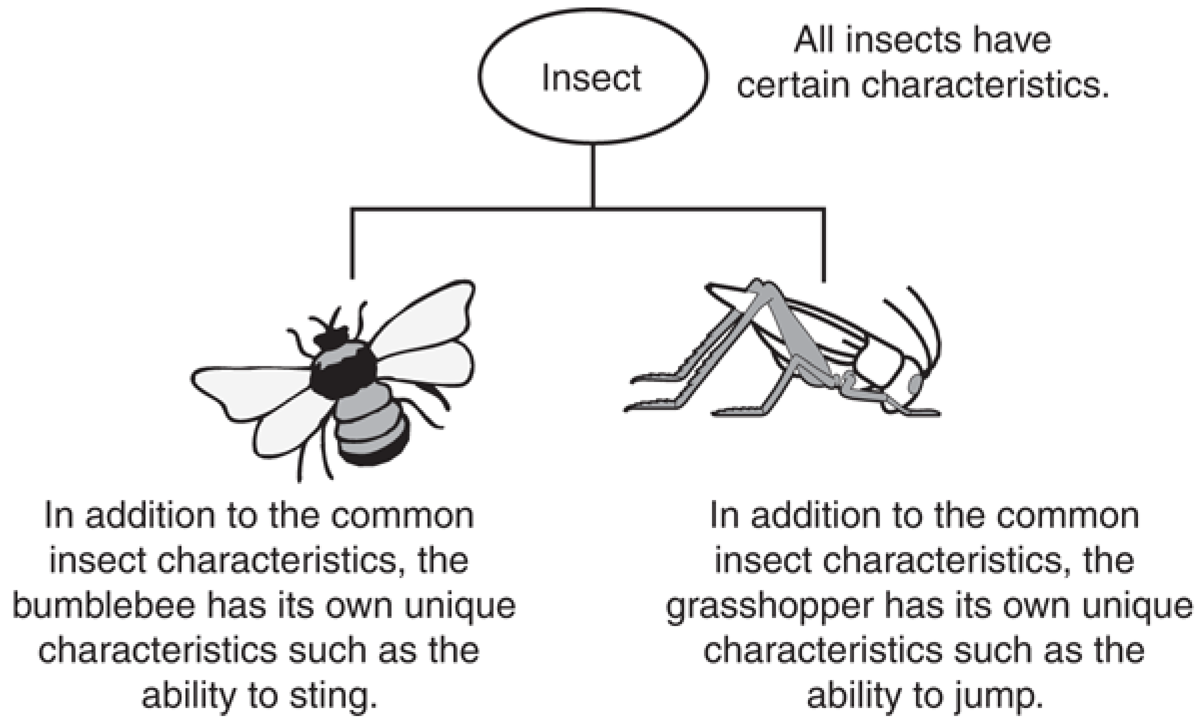
Concept:

Inheritance allows a new class to be based on an existing class. The new class inherits the members of the class it is based on.

Generalization and Specialization

In the real world, you can find many objects that are specialized versions of other more general objects. For example, the term "insect" describes a very general type of creature with numerous characteristics. Because grasshoppers and bumblebees are insects, they have all the general characteristics of an insect. In addition, they have special characteristics of their own. For example, the grasshopper has its jumping ability, and the bumblebee has its stinger. Grasshoppers and bumblebees are specialized versions of an insect. This is illustrated in **Figure 9-1**.

Figure 9-1 Bumblebees and grasshoppers are specialized versions of an insect


Insect
All insects have certain characteristics.
In addition to the common insect characteristics, the bumblebee has its own unique characteristics such as the ability to sting.
In addition to the common insect characteristics, the grasshopper has its own unique characteristics such as the ability to jump.

VideoNote Inheritance

Inheritance and the “Is-a” Relationship

When one object is a specialized version of another object, there is an *“is-a” relationship* between them. For example, a grasshopper *is an* insect. Here are a few other examples of the “is-a” relationship:

- A poodle *is a* dog.
- A car *is a* vehicle.
- A flower *is a* plant.
- A rectangle *is a* shape.
- A football player *is an* athlete.

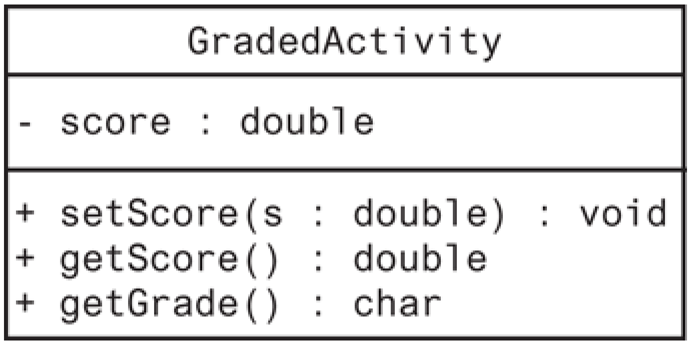
When an “is-a” relationship exists between objects, it means that the specialized object has all of the characteristics of the general object, plus additional characteristics that make it special. In object-oriented programming, *inheritance* is used to create an “is-a” relationship among classes. This allows you to extend the capabilities of a class by creating another class that is a specialized version of it.

Inheritance involves a superclass and a subclass. The *superclass* is the general class, and the *subclass* is the specialized class. (Superclasses are also called *base classes*, and subclasses are also called *derived classes*.) You can think of the subclass as an extended version of the superclass. The subclass inherits fields and methods from the superclass without any of them being rewritten.

Furthermore, new fields and methods can be added to the subclass to make it more specialized than the superclass.

Let’s look at an example of how inheritance can be used. Most teachers assign various graded activities for their students to complete. A graded activity can be given a numeric score such as 70, 85, 90, and so on, and a letter grade such as A, B, C, D or F. **Figure 9-2** shows a UML diagram for the `GradedActivity` class, which is designed to hold the numeric score of a graded activity. The `setScore` method sets a numeric score, and the `getScore` method returns the numeric score. The `getGrade` method returns the letter grade that corresponds to the numeric score. Notice that the class does not have a programmer-defined constructor, so Java will automatically generate a default constructor for it. This will be a point of discussion later. **Code Listing 9-1** shows the code for the class, and the program in **Code Listing 9-2** demonstrates the class.

Figure 9-2 UML diagram for the `GradedActivity` class



Code Listing 9-1 `(GradedActivity.java)`

```java
/**
 * A class that holds a grade for a graded activity.
 */

public class GradedActivity
{
  private double score; // Numeric score

  /**
   * The setScore method stores its argument in
   * the score field.
   */

  public void setScore(double s)
  {
   score = s;
  }

  /**
   * The getScore method returns the score field.
   */

  public double getScore()
  {
   return score;
  }

  /**
  * The getGrade method returns a letter grade
  * determined from the score field.
  */

  public char getGrade()
  {
   char letterGrade; // To hold the grade

   if (score >= 90)
     letterGrade = 'A';
   else if (score >= 80)
     letterGrade = 'B';
   else if (score >= 70)
     letterGrade = 'C';
   else if (score >= 60)
     letterGrade = 'D';
   else
     letterGrade = 'F';

   return letterGrade;
  }
}
```

Code Listing9-2 `(GradeDemo.java)`

```java
import java.util.Scanner;

/**
 * This program demonstrates the GradedActivity class.
 */

public class GradeDemo
{
  public static void main(String[] args)
  {
    double testScore; // To hold a test score

    // Create a Scanner object for keyboard input.
    Scanner keyboard = new Scanner(System.in);

    // Create a GradedActivity object.
    GradedActivity grade = new GradedActivity();

    // Get a test score from the user.
    System.out.print("Enter a numeric test score: ");
    testScore = keyboard.nextDouble();

    // Set the GradedActivity object's score.
    grade.setScore(testScore);

    // Display the letter grade for that score.
```

```
      System.out.println("The grade for that test is " +
                         grade.getGrade());
   }
}
```

Program Output with Example Input Shown in Bold

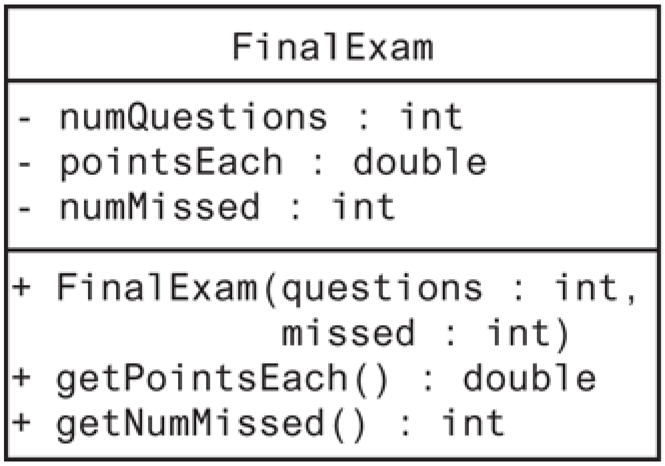
```
Enter a numeric test score: 89 Enter
The grade for that test is B
```

Program Output with Example Input Shown in Bold

```
Enter a numeric test score: 75 Enter
The grade for that test is C
```

The `GradedActivity` class represents the general characteristics of a student’s graded activity. Many different types of graded activities exist, however, such as quizzes, midterm exams, final exams, lab reports, essays, and so on. Because the numeric scores might be determined differently for each of these graded activities, we can create subclasses to handle each one. For example, a `FinalExam` class could inherit from the `GradedActivity` class. **Figure 9-3** shows the UML diagram for such a class, and **Code Listing 9-3** shows its code. It has fields for the number of questions on the exam (`numQuestions`), the number of points each question is worth (`pointsEach`), and the number of questions missed by the student (`numMissed`).

Figure 9-3 UML diagram for the `FinalExam` class



Code Listing 9-3 (FinalExam.java)

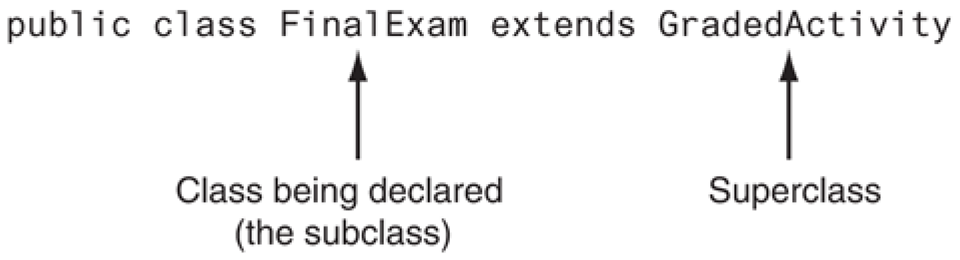
```
/**
 * This class determines the grade for a final exam.
 */

public class FinalExam extends GradedActivity
{
   private int numQuestions; // Number of questions
   private double pointsEach; // Points for each question
   private int numMissed;   // Number of questions missed

   /**
    * The constructor accepts as arguments the number
    * of questions on the exam and the number of
    * questions the student missed.
    */

   public FinalExam(int questions, int missed)
   {
    double numericScore; // To calculate the numeric score

    // Set the numQuestions and numMissed fields.
    numQuestions = questions;
    numMissed = missed;

    // Calculate the points for each question and
    // the numeric score for this exam.
```

```
      pointsEach = 100.0 / questions;
      numericScore = 100.0 - (missed * pointsEach);

      // Call the superclass's setScore method to
      // set the numeric score.
      setScore(numericScore);
   }

   /**
    * The getPointsEach method returns the pointsEach
    * field.
    */

   public double getPointsEach()
   {
      return pointsEach;
   }

   /**
    * The getNumMissed method returns the numMissed
    * field.
    */

   public int getNumMissed()
   {
      return numMissed;
   }
```

```
54 }
```

The only new notation in this class declaration is in the class header (in line 5), which is shown in **Figure 9-4**.

Figure 9-4 First line of the `FinalExam` class declaration


public class FinalExam extends GradedActivity
Class being declared (the subclass)
Superclass

The `extends` key word indicates that this class inherits from another class (a superclass). The name of the superclass is listed after the word `extends`. So, this line of code indicates that `FinalExam` is the name of the class being declared, and `GradedActivity` is the name of the superclass it inherits from.

As you read the line, it communicates the fact that the `FinalExam` class extends the `GradedActivity` class. This makes sense because a subclass is an extension of its superclass. If we want to express the relationship between the two classes, we can say that a `FinalExam` is a `GradedActivity`.

Because the `FinalExam` class inherits from the `GradedActivity` class, it inherits all of the public members of the `GradedActivity` class. Here is a list of the members of the `FinalExam` class:

Fields:

Methods:

Notice the `GradedActivity` class's `score` field is not listed among the members of the `FinalExam` class. That is because the `score` field is private. Private members of the superclass cannot be accessed by the subclass, so technically speaking, they are not inherited. When an object of the subclass is created, the private members of the superclass exist in memory, but only methods in the superclass can access them. They are truly private to the superclass.

You will also notice that the superclass's constructor is not listed among the members of the `FinalExam` class. It makes sense that superclass constructors are not inherited because their purpose is to construct objects of the superclass. In the next section, we will discuss in more detail how superclass constructors operate.

To see how inheritance works in this example, let's take a closer look at the `FinalExam` constructor. The constructor accepts two arguments: the number of test questions on the exam, and the number of questions missed by the student. These values are assigned to the `numQuestions` and `numMissed` fields in lines 22 and 23. Then, the number of points for each question is calculated in line 27, and the numeric test score is calculated in line 28. The last statement in the constructor, in line 32, reads:

```
setScore(numericScore);
```

This is a call to the `setScore` method. Although no `setScore` method appears in the `FinalExam` class, the method is inherited from the `GradedActivity` class. The program in **Code Listing 9-4** demonstrates the `FinalExam` class.

Code Listing 9-4 `(FinalExamDemo.java)`

```java
import java.util.Scanner;

/**
 * This program demonstrates the FinalExam class, which
 * inherits from the GradedActivity class.
 */

public class FinalExamDemo
{
   public static void main(String[] args)
   {
      int questions, // Number of questions
          missed;    // Number of questions missed

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the number of questions on the final exam.
      System.out.print("How many questions are on " +
                       "the final exam? ");
      questions = keyboard.nextInt();

      // Get the number of questions the student missed.
      System.out.print("How many questions did the " +
                       "student miss? ");
      missed = keyboard.nextInt();
27
28    // Create a FinalExam object.
29    FinalExam exam = new FinalExam(questions, missed);
30
31    // Display the test results.
32    System.out.println("Each question counts " +
33                 exam.getPointsEach() +
34                 " points.");
35    System.out.println("The exam score is " +
36                 exam.getScore());
37    System.out.println("The exam grade is " +
38                 exam.getGrade());
39   }
40 }
```

Program Output with Example Input Shown in Bold

```
How many questions are on the final exam? 20 Enter
How many questions did the student miss? 3 Enter
Each question counts 5.0 points.
The exam score is 85.0
The exam grade is B
```

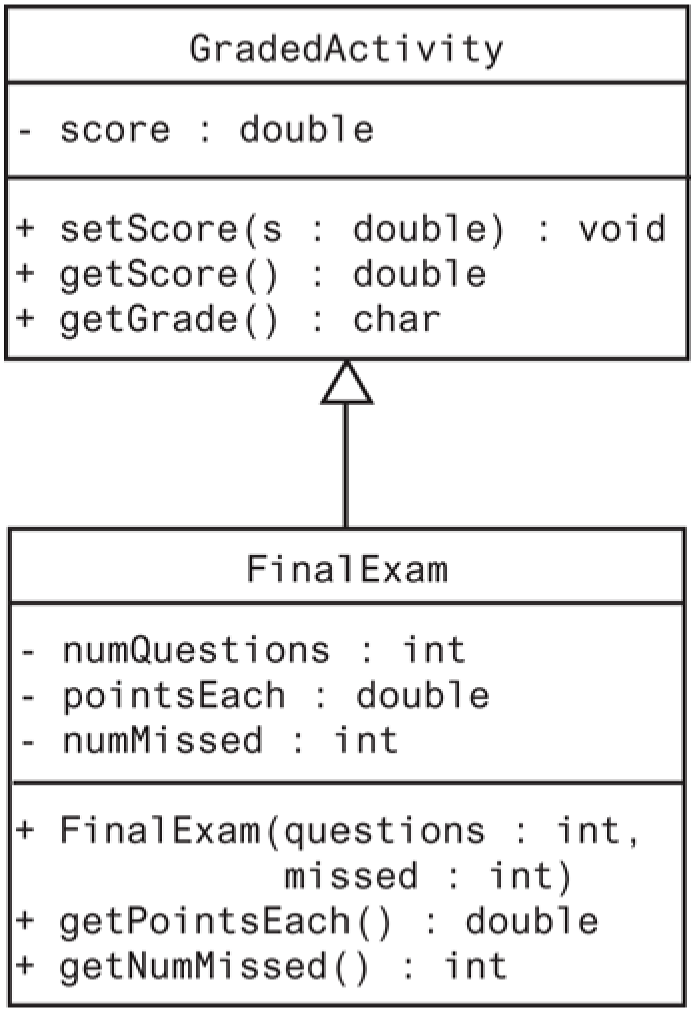
In line 29, this program creates an instance of the `FinalExam` class and assigns its address to the `exam` variable. When a `FinalExam` object is created in memory, it has not only the members declared in the `FinalExam` class, but the members declared in the `GradedActivity` class as well. Notice, in lines 36 and 38, two public methods of the `GradedActivity` class, `getScore` and `getGrade`, are directly called using the object referenced by `exam`. When a class inherits from another class, the public members of the superclass become public members of the subclass. In this program, the `getScore` and `getGrade` methods can be called from the `exam` object because they are public members of the object's superclass.

As mentioned before, the private members of the superclass (in this case, the `score` field) cannot be accessed by the subclass. When the `exam` object is created in memory, a `score` field exists, but only the methods defined in the superclass, `GradedActivity`, can access it. It is truly private to the superclass. Because the `FinalExam` constructor cannot directly access the `score` field, it must call the superclass's `setScore` method (which is public) to store a value in it.

Inheritance in UML Diagrams

You show inheritance in a UML diagram by connecting two classes with a line that has an open arrowhead at one end. The arrowhead points to the superclass. **Figure 9-5** shows a UML diagram depicting the relationship between the `GradedActivity` and `FinalExam` classes.

Figure 9-5 UML diagram showing inheritance



The Superclass’s Constructor

As was mentioned earlier, the `GradedActivity` class has only one constructor, which is the default constructor that Java automatically generated for it. When a `FinalExam` object is created, the `GradedActivity` class’s default constructor is executed just before the `FinalExam` constructor is executed. In an inheritance relationship, the superclass constructor always executes before the subclass constructor.

Code Listing 9-5 shows a class, `SuperClass1`, that has a programmer-defined no-arg constructor. The constructor simply displays the message “This is the superclass constructor.” **Code Listing 9-6** shows `SubClass1`, which inherits from `SuperClass1`. This class also has a programmer-defined no-arg constructor, which displays the message “This is the subclass constructor.”

Code Listing 9-5 *(SuperClass1.java)*

```
public class SuperClass1
{
  // Constructor
  public SuperClass1()
  {
   System.out.println("This is the superclass " +
                 "constructor.");
  }
}
```

Code Listing 9-6 *(SubClass1.java)*

```
public class SubClass1 extends SuperClass1
{
   // Constructor
   public SubClass1()
   {
     System.out.println("This is the subclass " +
                        "constructor.");
   }
}
```

The program in **Code Listing 9-7** creates a `SubClass1` object. As you can see from the program output, the superclass constructor executes first, followed by the subclass constructor.

Code Listing 9-7 `(ConstructorDemo1.java)`

```java
/**
 * This program demonstrates the order in which superclass
 * and subclass constructors are called.
 */

public class ConstructorDemo1
{
   public static void main(String[] args)
   {
      SubClass1 obj = new SubClass1();
   }
}
```

Program Output

```
This is the superclass constructor.
This is the subclass constructor.
```

Inheritance Does Not Work in Reverse

In an inheritance relationship, the subclass inherits members from the superclass, not the other way around. This means it is not possible for a superclass to call a subclass’s method. For example, if we create a `GradedActivity` object, it cannot call the `getPointsEach` or the `getNumMissed` methods, because they are members of the `FinalExam` class.

Checkpoint

9.1 Here is the first line of a class declaration. What is the name of the superclass? What is the name of the subclass?

```
public class Truck extends Vehicle
```

9.2 Look at the following class declarations, and answer the questions that follow them.

```
public class Shape
{
  private double area;

  public void setArea(double a)
   {
    area = a;
   }
  public double getArea()
   {
    return area;
   }
}
public class Circle extends Shape
{
  private double radius;

  public void setRadius(double r)
   {
    radius = r;
    setArea(Math.PI * r * r);
   }

  public double getRadius()
   {
    return radius;
    }
}
```

a. Which class is the superclass? Which class is the subclass?
b. Draw a UML diagram showing the relationship between these two classes.
c. When a `Circle` object is created, what are its public members?
d. What members of the `Shape` class are not accessible to the `Circle` class's methods?
e. Assume a program has the following declarations:

```
Shape s = new Shape();
Circle c = new Circle();
```

Indicate whether the following statements are legal or illegal:

```
c.setRadius(10.0);
s.setRadius(10.0);
System.out.println(c.getArea());
System.out.println(s.getArea());
```

9.3 Class `B` inherits from class `A`. Describe the order in which the class’s constructors execute when a class `B` object is created.

9.2 Calling the Superclass Constructor

Concept:

The super key word refers to an object’s superclass. You can use the super key word to call a superclass constructor.

In the previous section, you learned that a superclass’s default constructor or no-arg constructor is automatically called just before the subclass’s constructor executes. But what if the superclass does not have a default constructor or a no-arg constructor? Or, what if the superclass has multiple overloaded constructors, and you want to make sure a specific one is called? In either of these situations, you use the `super` key word to explicitly call a superclass constructor. The `super` key word refers to an object’s superclass, and can be used to access members of the superclass.

Code Listing 9-8 shows a class, `SuperClass2`, which has a no-arg constructor and a constructor that accepts an `int` argument. **Code Listing 9-9** shows `SubClass2`, which inherits from `SuperClass2`. This class's constructor uses the `super` key word to call the superclass's constructor and pass an argument to it.

Code Listing 9-8 `(SuperClass2.java)`

```java
public class SuperClass2
{
  // No-arg constructor
  public SuperClass2()
  {
   System.out.println("This is the superclass " +
                     "no-arg constructor.");
  }

  // Constructor #2
  public SuperClass2(int arg)
  {
   System.out.println("The following argument was " +
                     "passed to the superclass " +
                     "constructor: " + arg);
  }
}
```

Code Listing 9-9 `(SubClass2.java)`

```
 1 public class SubClass2 extends SuperClass2
 2 {
 3    // Constructor
 4    public SubClass2()
 5    {
 6       // Call the superclass constructor.
 7       super(10);
 8
 9       // Display a message.
10       System.out.println("This is the subclass " +
11                          "constructor.");
12    }
13 }
```

In the `SubClass2` constructor, the statement in line 7 calls the superclass constructor and passes the argument 10 to it. Here are three guidelines you should remember about calling a superclass constructor:

- The `super` statement that calls the superclass constructor can be written only in the subclass's constructor. You cannot call the superclass constructor from any other method.

- The `super` statement that calls the superclass constructor must be the first statement in the subclass's constructor. This is because the superclass's constructor must execute before the code in the subclass's constructor executes.
- If a subclass constructor does not explicitly call a superclass constructor, Java will automatically call the superclass's default constructor, or no-arg constructor, just before the code in the subclass's constructor executes. This is equivalent to placing the following statement at the beginning of a subclass constructor:

  ```
  super();
  ```

The program in **Code Listing 9-10** demonstrates these classes.

Code Listing 9-10 `(ConstructorDemo2.java)`

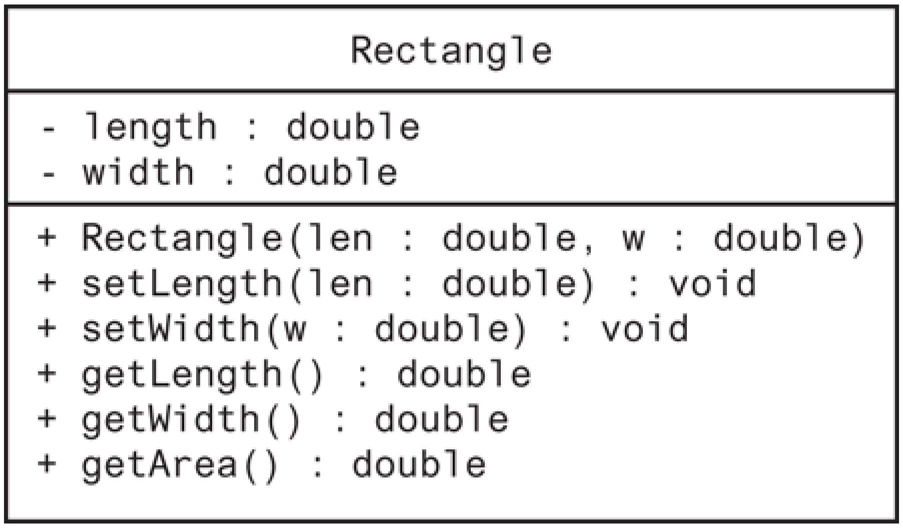
```
/**
 * This program demonstrates how a superclass constructor
 * can be called with the super key word.
 */

public class ConstructorDemo2
{
   public static void main(String[] args)
   {
      SubClass2 obj = new SubClass2();
   }
}
```

Program Output

```
The following argument was passed to the superclass
constructor: 10
This is the subclass constructor.
```

Let’s look at a more meaningful example. Recall the `Rectangle` class that was introduced in **Chapter 3**. **Figure 9-6** shows a UML diagram for the class.

Figure 9-6 UML diagram for the `Rectangle` class



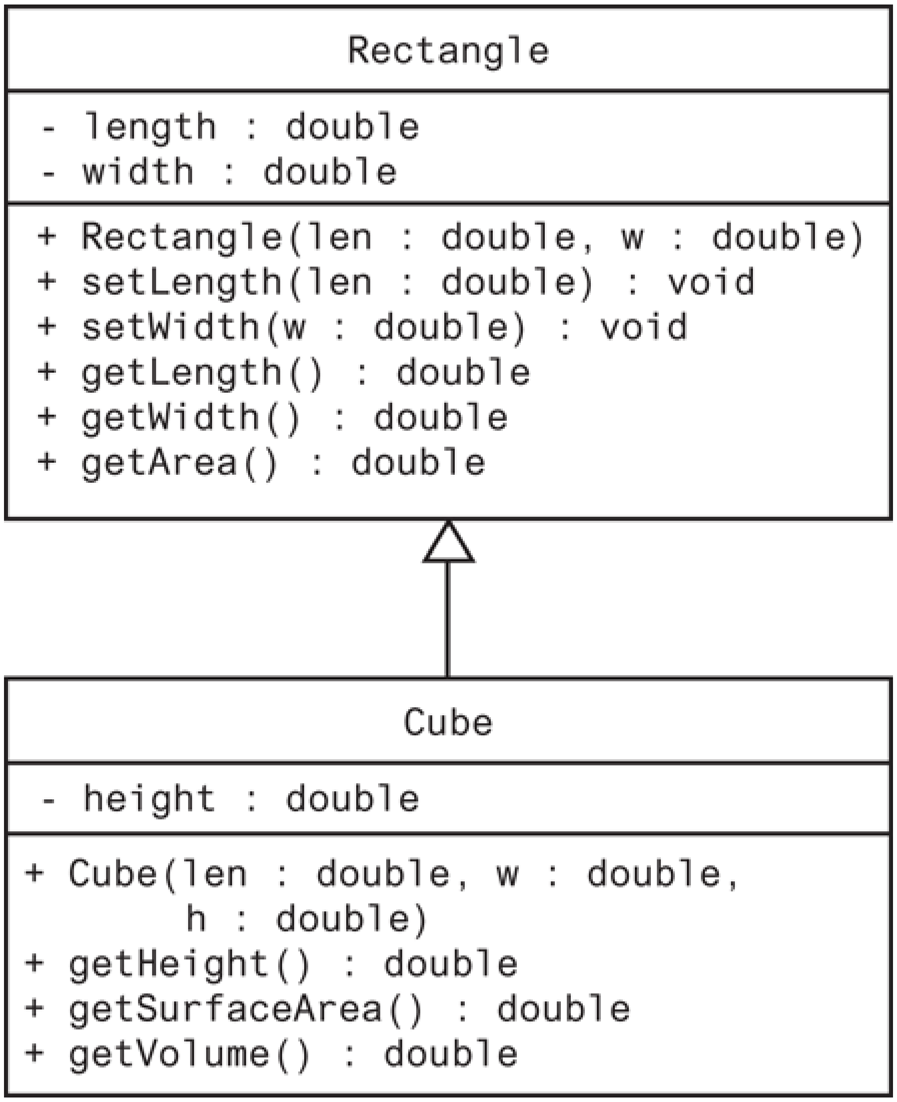
```
Rectangle
- length : double
- width : double
+ Rectangle(len : double, w : double)
+ setLength(len : double) : void
+ setWidth(w : double) : void
+ getLength() : double
+ getWidth() : double
+ getArea() : double
```

Here is part of the class's code:

```
public class Rectangle
{
  private double length;
  private double width;
  /**
  * Constructor
  */
  public Rectangle(double len, double w)
  {
   length = len;
   width = w;
  }
  (Other methods follow . . .)
}
```

Next we will design a `Cube` class, which inherits from the `Rectangle` class. The `Cube` class is designed to hold data about cubes, which not only have a length, width, and area (the area of the base), but a height, surface area, and volume as well. A UML diagram showing the inheritance relationship between the `Cube` and `Rectangle` classes is shown in **Figure 9-7**, and the code for the `Cube` class is shown in **Code Listing 9-11**.

Figure 9-7 UML diagram for the `Rectangle` and `Cube` classes



Code Listing 9-11 `(Cube.java)`

```java
/**
 * This class holds data about a cube.
 */

public class Cube extends Rectangle
{
   private double height; // The height of the cube

   /**
    * The constructor accepts the cube's length,
    * width, and height as arguments.
    */

   public Cube(double len, double w, double h)
   {
     // Call the superclass constructor to
     // initialize length and width.
     super(len, w);

     // Initialize height.
     height = h;
   }

   /**
    * The getHeight method returns the height
    * field.
    */

   public double getHeight()
   {
    return height;
   }

   /**
   * The getSurfaceArea method returns the
   * cube's surface area.
   */

   public double getSurfaceArea()
   {
    return getArea() * 6;
   }

   /**
   * The getVolume method returns the volume of
   * the cube.
   */

   public double getVolume()
   {
    return getArea() * height;
   }
 }
```

The `Cube` constructor accepts arguments for the parameters `w`, `len`, and `h`. The values that are passed to `w` and `len` are subsequently passed as arguments to the `Rectangle` constructor in line 18. When the `Rectangle` constructor finishes, the remaining code in the `Cube` constructor is executed. The program in **Code Listing 9-12** demonstrates the class.

Code Listing 9-12 `(CubeDemo.java)`

```java
import java.util.Scanner;

/**
 * This program demonstrates passing arguments to a
 * superclass constructor.
 */

public class CubeDemo
{
   public static void main(String[] args)
   {
      double length,   // To hold a length
             width,   // To hold a width
             height;  // To hold a height

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the dimensions of a cube from the user.
      System.out.println("Enter the following dimensions " +
                        "of a cube: ");
      System.out.print("Length: ");
      length = keyboard.nextDouble();
      System.out.print("Width: ");
      width = keyboard.nextDouble();
      System.out.print("Height: ");
      height = keyboard.nextDouble();

      // Create a cube object and pass the dimensions
      // to the constructor.
      Cube myCube = new Cube(length, width, height);

      // Display the properties of the cube.
      System.out.println();
      System.out.println("Here are the properties of " +
                "the cube.");
      System.out.println("Length: " + myCube.getLength());
      System.out.println("Width: " + myCube.getWidth());
      System.out.println("Height: " + myCube.getHeight());
      System.out.println("Base Area: " + myCube.getArea());
      System.out.println("Surface Area: " +
                myCube.getSurfaceArea());
      System.out.println("Volume: " + myCube.getVolume());
   }
}
```

Program Output with Example Input Shown in Bold

```
Enter the following dimensions of a cube:
Length: 10 Enter
Width: 15 Enter
Height: 12 Enter
Here are the properties of the cube.
Length: 10.0
Width: 15.0
Height: 12.0
Base Area: 150.0
Surface Area: 900.0
Volume: 1800.0
```

When the Superclass Has No Default or No-Arg Constructor

Recall from **Chapter 3** that Java provides a default constructor for a class only when you provide no constructors for the class. This makes it possible to have a class with no default constructor. The `Rectangle` class we just looked at is an example. It has a constructor that accepts two arguments. Because we have provided this constructor, the `Rectangle` class does not have a default constructor. In addition, we have not written a no-arg constructor for the class.

If a superclass does not have a default constructor and does not have a no-arg constructor, then a class that inherits from it *must* call one of the constructors that the superclass does have. If it does not, an error will result when the subclass is compiled.

Summary of Constructor Issues in Inheritance

We have covered a number of important issues that you should remember about constructors in an inheritance relationship. The following list summarizes them:

- The superclass constructor always executes before the subclass constructor.
- You can write a `super` statement that calls a superclass constructor, but only in the subclass's constructor. You cannot call the superclass constructor from any other method.
- If a `super` statement that calls a superclass constructor appears in a subclass constructor, it must be the first statement.
- If a subclass constructor does not explicitly call a superclass constructor, Java will automatically call `super()` just before the code in the subclass's constructor executes.
- If a superclass does not have a default constructor and does not have a no-arg constructor, then a class that inherits from it *must* call one of the constructors that the superclass does have.

Checkpoint

9.4 Look at the following classes:

```
public class Ground
{
  public Ground()
  {
    System.out.println("You are on the ground.");
  }
}
public class Sky extends Ground
{
  public Sky()
  {
    System.out.println("You are in the sky.");
  }
}
```

What will the following program display?

```
public class Checkpoint
{
  public static void main(String[] args)
  {
    Sky object = new Sky();
  }
}
```

9.5 Look at the following classes:

```
public class Ground
{
  public Ground()
  {
   System.out.println("You are on the ground.");
  }
  public Ground(String groundColor)
  {
   System.out.println("The ground is " + groundColor);
  }
}
public class Sky extends Ground
{
  public Sky()
  {
   System.out.println("You are in the sky.");
  }
  public Sky(String skyColor)
  {
   super("green");
   System.out.println("The sky is " + skyColor);
  }
}
```

What will the following program display?

```
public class Checkpoint
{
  public static void main(String[] args)
  {
   Sky object = new Sky("blue");
  }
}
```

9.3 Overriding Superclass Methods

Concept:

A subclass may have a method with the same signature as a superclass method. In such a case, the subclass method overrides the superclass method.

Sometimes a subclass inherits a method from its superclass, but the method is inadequate for the subclass's purpose. Because the subclass is more specialized than the superclass, it is sometimes necessary for the subclass to replace inadequate superclass methods with more suitable ones. This is known as *method overriding*.

For example, recall the `GradedActivity` class presented earlier in this chapter. This class has a `setScore` method that sets a numeric score, and a `getGrade` method that returns a letter grade based on that score. But, suppose a teacher wants to curve a numeric score before the letter grade is determined. For example, Dr. Harrison determines that in order to curve the grades in her class, she must multiply each student’s score by a certain percentage. This gives an adjusted score that is used to determine the letter grade. To satisfy this need, we can design a new class, `CurvedActivity`, which inherits from the `GradedActivity` class and has its own specialized version of the `setScore` method. The `setScore` method in the subclass *overrides* the `setScore` method in the superclass. **Figure 9-8** shows a UML diagram depicting the relationship between the `GradedActivity` class and the `CurvedActivity` class.

Figure 9-8 The `GradedActivity` and `CurvedActivity` classes

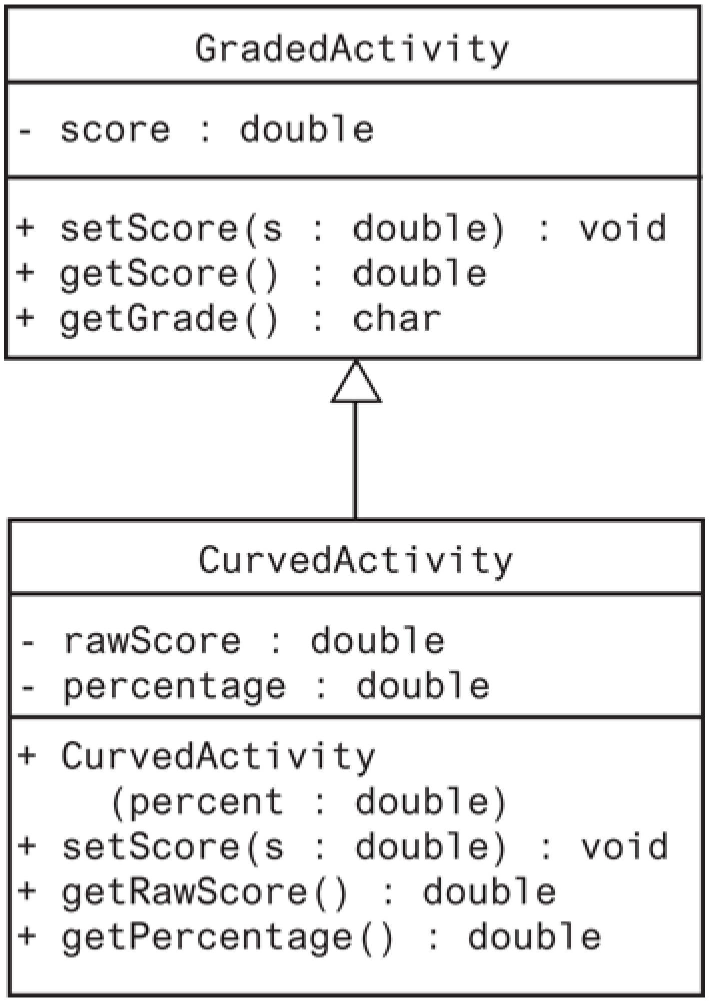


Table 9-1 summarizes the `CurvedActivity` class's fields, and **Table 9-2** summarizes the class's methods.

Table 9-1 `CurvedActivity` class fields

Table 9-2 `CurvedActivity` class methods

Code Listing 9-13 shows the `CurvedActivity` class. The `setScore` method appears in lines 31 through 35. It is important to note that the `setScore` method in the `CurvedActivity` class has the same signature as the `setScore` method in the superclass `GradedActivity`. In order for overriding to occur, the subclass method must have the same signature as the superclass method. When an object of the subclass invokes the `setScore` method, it invokes the subclass's version of the method, not the superclass's.

Note:

Recall from **Chapter 6**, a method's signature consists of the method's name and the data types of the method's parameters, in the order that they appear.

Code Listing 9-13 `(CurvedActivity.java)`

```
/**
 * This class computes a curved grade. It extends
 * the GradedActivity class.
 */

public class CurvedActivity extends GradedActivity
{
   double rawScore;    // Unadjusted score
   double percentage;  // Curve percentage

   /**
    * The constructor sets the curve percentage.
    */

   public CurvedActivity(double percent)
   {
     percentage = percent;
     rawScore = 0.0;
   }

   /**
    * The setScore method overrides the superclass setScore
method.
    * This version accepts the unadjusted score as an
argument. That
    * score is multiplied by the curve percentage and the
```

```
result is
    * sent as an argument to the superclass's setScore
method.
    */

   @Override
   public void setScore(double s)
   {
      rawScore = s;
      super.setScore(rawScore * percentage);
   }

   /**
    * The getRawScore method returns the raw score.
    */

   public double getRawScore()
   {
      return rawScore;
   }

   /**
    * The getPercentage method returns the curve
    * percentage.
    */

   public double getPercentage()
   {
```

```
51     return percentage;
52   }
53 }
```

Notice in line 28, the `@Override` annotation appears just before the `setScore` method definition. This annotation tells the Java compiler that the `setScore` method is meant to override a method in the superclass.

The `@Override` annotation in line 28 is not required, but it is recommended that you use it. If the method fails to correctly override a method in the superclass, the compiler will display an error message. For example, suppose we had written the method header in line 29 like this:

```
public void setscore(double s)
```

If you look closely at the method name, you will see that all the letters are written in lowercase. This does not match the method's name in the superclass, which is `setScore`. Without the `@Override` annotation, the code would still compile and execute, but we would not get the expected results because the method in the subclass would not override the method in the superclass. However, by using the `@Override` annotation in line 28, the compiler would generate an error letting us know that the subclass method does not override any method in the superclass.

Let’s take a closer look at the `setScore` method in the `CurvedActivity` class. It accepts an argument, which is the student’s unadjusted numeric score. This value is stored in the `rawScore` field. Then the following statement, in line 32, is executed:

```
super.setScore(rawScore * percentage);
```

As you already know, the `super` key word refers to the object’s superclass. This statement calls the superclass’s version of the `setScore` method with the result of the expression `rawScore * percentage` passed as an argument. This is necessary because the superclass’s `score` field is private, and the subclass cannot access it directly. In order to store a value in the superclass’s `score` field, the subclass must call the superclass’s `setScore` method. A subclass may call an overridden superclass method by prefixing its name with the `super` key word and a dot (`.`). The program in **Code Listing 9-14** demonstrates this class.

Code Listing 9-14 `(CurvedActivityDemo.java)`

```java
import java.util.Scanner;

/**
 * This program demonstrates the CurvedActivity class,
 * which inherits from the GradedActivity class.
 */

public class CurvedActivityDemo
{
   public static void main(String[] args)
   {
      double score,       // Raw score
             curvePercent;  // Curve percentage

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the unadjusted exam score.
      System.out.print("Enter the student's raw " +
                       "numeric score: ");
      score = keyboard.nextDouble();

      // Get the curve percentage.
      System.out.print("Enter the curve percentage: ");
      curvePercent = keyboard.nextDouble();

```

```
    // Create a CurvedActivity object.
    CurvedActivity curvedExam =
          new CurvedActivity(curvePercent);

    // Set the exam score.
    curvedExam.setScore(score);

    // Display the test results.
    System.out.println("The raw score is " +
              curvedExam.getRawScore() +
              " points.");
    System.out.println("The curved score is " +
              curvedExam.getScore());
    System.out.println("The exam grade is " +
              curvedExam.getGrade());
  }
}
```

Program Output with Example Input Shown in Bold

```
Enter the student's raw numeric score: 87 Enter
Enter the curve percentage: 1.06 Enter
The raw score is 87.0 points.
The curved score is 92.22
The exam grade is A
```

This program uses the `curvedExam` variable to reference a `CurvedActivity` object. The statement in line 32 calls the `setScore` method. Because `curvedExam` references a `CurvedActivity` object, this statement calls the `CurvedActivity` class's `setScore` method, not the superclass's version.

Even though a subclass may override a method in the superclass, superclass objects still call the superclass version of the method. For example, the following code creates an object of the `GradedActivity` class, and calls the `setScore` method:

```
GradedActivity regularExam = new GradedActivity();
regularExam.setScore(85);
```

Because `regularExam` references a `GradedActivity` object, this code calls the `GradedActivity` class's version of the `setScore` method.

Overloading versus Overriding

There is a distinction between overloading a method and overriding a method. Recall from **Chapter 6**, overloading is when a method has the same name as one or more other methods, but a different parameter list. Although overloaded methods have the same name, they have different signatures. When a method overrides another method, however, they both have the same signature.

Both overloading and overriding can take place in an inheritance relationship. You already know overloaded methods can appear within the same class. In addition, a method in a subclass can overload a method in the superclass. If class `A` is the superclass and class `B` is the subclass, a method in class `B` can overload a method in class `A`, or another method in class `B`. Overriding, on the other hand, can take place only in an inheritance relationship. If class `A` is the superclass and class `B` is the subclass, a method in class `B` can override a method in class `A`. However, a method cannot override another method in the same class. The following list summarizes the distinction between overloading and overriding:

- If two methods have the same name but different signatures, they are overloaded. This is true where the methods are in the same class, or where one method is in the superclass and the other method is in the subclass.
- If a method in a subclass has the same signature as a method in the superclass, the subclass method overrides the superclass

method.

The distinction between overloading and overriding is important because it can affect the accessibility of superclass methods in a subclass. When a subclass overloads a superclass method, both methods can be called with a subclass object. However, when a subclass overrides a superclass method, only the subclass's version of the method can be called with a subclass object. For example, look at the `SuperClass3` class in **Code Listing 9-15**. It has two overloaded methods named `showValue`. One of the methods accepts an `int` argument, and the other accepts a `String` argument.

Code Listing 9-15 *(SuperClass3.java)*

```java
public class SuperClass3
{
   /**
    * The following method displays an int.
    */

   public void showValue(int arg)
   {
      System.out.println("SUPERCLASS: The int argument was " + arg);
   }

   /**
    * The following method displays a String.
    */

   public void showValue(String arg)
   {
      System.out.println("SUPERCLASS: The String argument was " + arg);
   }
}
```

Now look at the `SubClass3` class in **Code Listing 9-16**. It inherits from the `SuperClass3` class.

Code Listing 9-16 (SubClass3.java)

```java
public class SubClass3 extends SuperClass3
{
   /**
    * This method overrides one of the superclass methods.
    */

   @Override
   public void showValue(int arg)
   {
     System.out.println("SUBCLASS: The int argument was " +
arg);
   }

   /**
    * This method overloads the superclass methods.
    */

   public void showValue(double arg)
   {
     System.out.println("SUBCLASS: The double argument was
" + arg);
   }
}
```

Notice `SubClass3` also has two methods named `showValue`. The first one, in lines 8 through 11, accepts an `int` argument. This method overrides one of the superclass methods because they have the same signature. The second `showValue` method, in lines 17 through 20, accepts a `double` argument. This method overloads the other `showValue` methods because none of the others have the same signature. Although there is a total of four `showValue` methods in these classes, only three of them can be called from a `SubClass3` object. This is demonstrated in **Code Listing 9-17**.

Code Listing 9-17 `(ShowValueDemo.java)`

```java
/**
 * This program demonstrates the methods in the
 * SuperClass3 and SubClass3 classes.
 */

public class ShowValueDemo
{
   public static void main(String[] args)
   {
      SubClass3 myObject = new SubClass3();

      myObject.showValue(10);      // Pass an int.
      myObject.showValue(1.2);     // Pass a double.
      myObject.showValue("Hello");  // Pass a String.
   }
}
```

Program Output

```
SUBCLASS: The int argument was 10
SUBCLASS: The double argument was 1.2
SUPERCLASS: The String argument was Hello
```

When an `int` argument is passed to `showValue`, the subclass's method is called because it overrides the superclass method. In order to call the overridden superclass method, we would have to use the `super` key word in the subclass method. Here is an example:

```
public void showValue(int arg)
{
  super.showValue(arg);  // Call the superclass method.
  System.out.println("SUBCLASS: The int argument was " + arg);
}
```

Preventing a Method from Being Overridden

When a method is declared with the `final` modifier, it cannot be overridden in a subclass. The following method header is an example that uses the `final` modifier:

```
public final void message()
```

If a subclass attempts to override a `final` method, the compiler generates an error. This technique can be used to make sure a particular superclass method is used by subclasses, and not a modified version of it.

Checkpoint

9.6 Under what circumstances would a subclass need to override a superclass method?
9.7 How can a subclass method call an overridden superclass method?
9.8 If a method in a subclass has the same signature as a method in the superclass, does the subclass method overload or override the superclass method?
9.9 If a method in a subclass has the same name as a method in the superclass, but uses a different parameter list, does the subclass method overload or override the superclass method?
9.10 How do you prevent a method from being overridden?

9.4 Protected Members

Concept:

Protected members of a class can be accessed by methods in a subclass, and by methods in the same package as the class.

Until now, you have used two access specifications within a class: `private` and `public`. Java provides a third access specification, `protected`. A protected member of a class can be directly accessed by methods of the same class or methods of a subclass. In addition, protected members can be accessed by methods of any class that are in the same package as the protected member's class. A protected member is not quite private, because it can be accessed by some methods outside the class. Protected members are not quite public either because access to them is restricted to methods in the same class, subclasses, and classes in the same package as the member's class. A protected member's access is somewhere between private and public.

Let’s look at a class with a protected member. **Code Listing 9-18** shows the `GradedActivity2` class, which is a modification of the `GradedActivity` class presented earlier. In this class, the `score` field has been made protected instead of private.

Code Listing 9-18 `(GradedActivity2.java)`

```java
/**
 * A class that holds a grade for a graded activity.
 */

public class GradedActivity2
{
  protected double score; // Numeric score

  /**
   * The setScore method stores its argument in
   * the score field.
   */

  public void setScore(double s)
  {
   score = s;
  }

  /**
   * The getScore method returns the score field.
   */

  public double getScore()
  {
   return score;
  }

  /**
  * The getGrade method returns a letter grade
  * determined from the score field.
  */

  public char getGrade()
  {
   char letterGrade;  // To hold the grade

   if (score >= 90)
     letterGrade = 'A';
   else if (score >= 80)
     letterGrade = 'B';
   else if (score >= 70)
     letterGrade = 'C';
   else if (score >= 60)
     letterGrade = 'D';
   else
     letterGrade = 'F';

   return letterGrade;
  }
}
```

Because the `score` field is declared as protected, any class that inherits from this class has direct access to it. The `FinalExam2` class, shown in **Code Listing 9-19**, is an example. This class is a modification of the `FinalExam` class, which was presented earlier. This class has a new method, `adjustScore`, which directly accesses the superclass's `score` field. If the contents of `score` have a fractional part of .5 or greater, the method rounds `score` up to the next whole number. The `adjustScore` method is called from the constructor.

Code Listing 9-19 `(FinalExam2.java)`

```java
/**
 * This class determines the grade for a final exam. The
 * numeric score is rounded up to the next whole number
 * if its fractional part is .5 or greater.
 */

public class FinalExam2 extends GradedActivity2
{
   private int numQuestions; // Number of questions
   private double pointsEach; // Points for each question
   private int numMissed;    // Number of questions missed

   /**
    * The constructor accepts as arguments the number
    * of questions on the exam and the number of
    * questions the student missed.
    */

   public FinalExam2(int questions, int missed)
   {
      double numericScore; // To hold the numeric score

      // Set the numQuestions and numMissed fields.
      numQuestions = questions;
      numMissed = missed;

      // Calculate the points for each question and
    // the numeric score for this exam.
    pointsEach = 100.0 / questions;
    numericScore = 100.0 - (missed * pointsEach);

    // Call the superclass's setScore method to
    // set the numeric score.
    setScore(numericScore);
    adjustScore();
   }

   /**
   * The getPointsEach method returns the pointsEach
   * field.
   */

   public double getPointsEach()
   {
    return pointsEach;
   }

   /**
   * The getNumMissed method returns the numMissed
   * field.
   */

   public int getNumMissed()
   {
    return numMissed;
   }

   /**
    * The adjustScore method adjusts a numeric score.
    * If score is within 0.5 points of the next whole
    * number, it rounds the score up.
    */

   private void adjustScore()
   {
     double fraction; // Fractional part of a score

     // Get the fractional part of the score.
     fraction = score - (int) score;

     // If the fractional part is 0.5 or greater,
     // round the score up to the next whole number.
     if (fraction >= 0.5)
        score = score + (1.0 - fraction);
   }
}
```

The program in **Code Listing 9-20** demonstrates the `FinalExam2` class.

Code Listing 9-20 `(ProtectedDemo.java)`

```java
import java.util.Scanner;

/**
 * This program demonstrates the FinalExam2 class, which
 * inherits from the GradedActivity2 class.
 */

public class ProtectedDemo
{
   public static void main(String[] args)
   {
      int questions,  // Number of questions
          missed;  // Number of questions missed

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the number of questions on the final exam.
      System.out.print("How many questions are on " +
                       "the final exam? ");
      questions = keyboard.nextInt();

      // Get the number of questions the student missed.
      System.out.print("How many questions did the " +
                       "student miss? ");
      missed = keyboard.nextInt();

```

```
    // Create a FinalExam2 object.
    FinalExam2 exam =
           new FinalExam2(questions, missed);

    // Display the test results.
    System.out.println("Each question counts " +
              exam.getPointsEach() +
              " points.");
    System.out.println("The exam score is " +
              exam.getScore());
    System.out.println("The exam grade is " +
              exam.getGrade());
   }
}
```

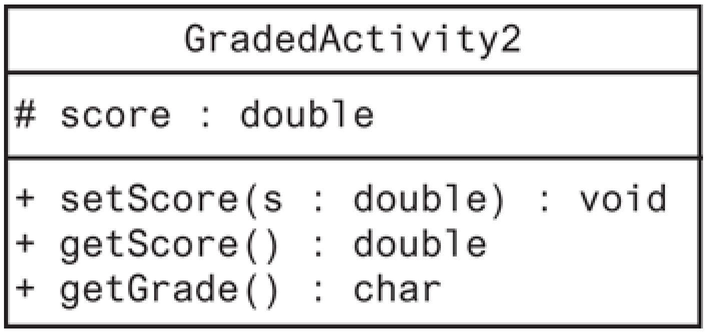
Program Output with Example Input Shown in Bold

```
How many questions are on the final exam? 40 Enter
How many questions did the student miss? 5 Enter
Each question counts 2.5 points.
The exam score is 88.0
The exam grade is B
```

In the example running of the program, the student missed 5 out of 40 questions. The unadjusted numeric score would be 87.5, but the `adjustScore` method rounded the `score` field up to 88.

Protected class members can be denoted in a UML diagram with the # symbol. **Figure 9-9** shows a UML diagram for the `GradedActivity2` class, with the score field denoted as protected.

Figure 9-9 UML diagram for the `GradedActivity2` class



Although making a class member protected instead of private might make some tasks easier, you should avoid this practice when possible. This is because any class that inherits from the class, or is in the same package, has unrestricted access to the protected member. It is always better to make all fields private, then provide public methods for accessing those fields.

Package Access

If you do not provide an access specifier for a class member, the class member is given *package access* by default. This means that any method in the same package can access the member. Here is an example:

```
public class Circle
{
   double radius;
   int centerX, centerY;

   (Method definitions follow . . .)
}
```

In this class, the `radius`, `centerX`, and `centerY` fields were not given an access specifier, so the compiler grants them package access. Any method in the same package as the `Circle` class can directly access these members.

There is a subtle difference between protected access and package access. Protected members can be accessed by methods in the same package or in a subclass. This is true even if the subclass is in a different package. Members with package access, however, cannot be accessed by subclasses that are in a different package.

It is more likely that you will give package access to class members by accident than by design, because it is easy to forget the access specifier. Although there are circumstances under which package access can be helpful, you should normally avoid it. Be careful to always specify an access specifier for class members.

Tables 9-3 and 9-4 summarize how each of the access specifiers affect a class member's accessibility within and outside of the class's package.

Table 9-3 Accessibility from within the class's package

Table 9-4 Accessibility from outside the class's package

Checkpoint

9.11 When a class member is declared as protected, what code can access it?
9.12 What is the difference between private members and protected members?
9.13 Why should you avoid making class members protected when possible?
9.14 What is the difference between private access and package access?
9.15 Why is it easy to give package access to a class member by accident?

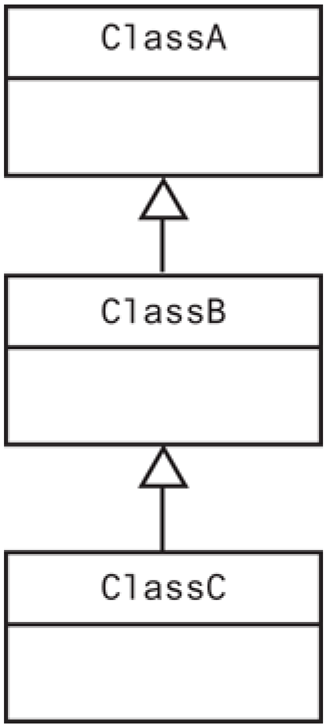
9.5 Classes That Inherit from Subclasses

Concept:

A superclass can also inherit from another class.

Sometimes it is desirable to establish a chain of inheritance in which one class inherits from a second class, which in turn inherits from a third class, as illustrated by **Figure 9-10**. In some cases, this chaining of classes goes on for many layers.

Figure 9-10 A chain of inheritance



In **Figure 9-10**, `ClassC` inherits `ClassB`'s members, including the ones that `ClassB` inherited from `ClassA`. Let's look at an example of such a chain of inheritance. Consider the `PassFailActivity` class, shown in **Code Listing 9-21**, which inherits from the `GradedActivity` class. The class is intended to determine a letter grade of `"P"` for passing, or `"F"` for failing.

Code Listing 9-21 `(PassFailActivity.java)`

```
/**
 * This class holds a numeric score and determines
 * whether the score is passing or failing.
 */

public class PassFailActivity extends GradedActivity
{
   private double minPassingScore; // Minimum passing score

   /**
    * The constructor accepts the minimum passing
    * score as its argument.
    */

   public PassFailActivity(double mps)
   {
    minPassingScore = mps;
   }

   /**
    * The getGrade method returns a letter grade determined
    * from the score field. This method overrides the
getGrade
    * method in the superclass.
    */

   @Override
27   public char getGrade()
28   {
29    char letterGrade; // To hold the letter grade
30
31    if (super.getScore() >= minPassingScore)
32      letterGrade = 'P';
33    else
34      letterGrade = 'F';
35
36    return letterGrade;
37   }
38 }
```

The `PassFailActivity` constructor, in lines 15 through 18, accepts a `double` argument that is the minimum passing grade for the activity. This value is stored in the `minPassingScore` field. The `getGrade` method in lines 27 through 37, which overrides the superclass method, returns a grade of `"P"` if the numeric score is greater than or equal to `minPassingScore`. Otherwise, the method returns a grade of `"F"`.

Suppose we wish to extend this class with another class that is even more specialized. For example, the `PassFailExam` class, shown in **Code Listing 9-22**, determines a passing or failing grade for an exam. It has fields for the number of questions on the exam (`numQuestions`), the number of points each question is worth (`pointsEach`), and the number of questions missed by the student (`numMissed`).

Code Listing 9-22 `(PassFailExam.java)`

```java
/**
 * This class determines a passing or failing grade for
 * an exam.
 */

public class PassFailExam extends PassFailActivity
{
   private int numQuestions;   // Number of questions
   private double pointsEach;   // Points for each question
   private int numMissed;     // Number of questions missed

   /**
   * The constructor accepts as arguments the number
   * of questions on the exam, the number of
   * questions the student missed, and the minimum
   * passing score.
   */

   public PassFailExam(int questions, int missed,
                double minPassing)
   {
    // Call the superclass constructor.
    super(minPassing);

    // Declare a local variable for the numeric score.
    double numericScore;

    // Set the numQuestions and numMissed fields.
```

```
      numQuestions = questions;
      numMissed = missed;

      // Calculate the points for each question and
      // the numeric score for this exam.
      pointsEach = 100.0 / questions;
      numericScore = 100.0 - (missed * pointsEach);

      // Call the superclass's setScore method to
      // set the numeric score.
      setScore(numericScore);
    }

    /**
    * The getPointsEach method returns the pointsEach
    * field.
    */

    public double getPointsEach()
    {
      return pointsEach;
    }

    /**
    * The getNumMissed method returns the numMissed
    * field.
    */

```

```
   public int getNumMissed()
   {
      return numMissed;
   }
}
```

The `PassFailExam` class inherits the `PassFailActivity` class’s members, including the ones that `PassFailActivity` inherited from `GradedActivity`. The program in **Code Listing 9-23** demonstrates the class.

Code Listing 9-23 `(PassFailExamDemo.java)`

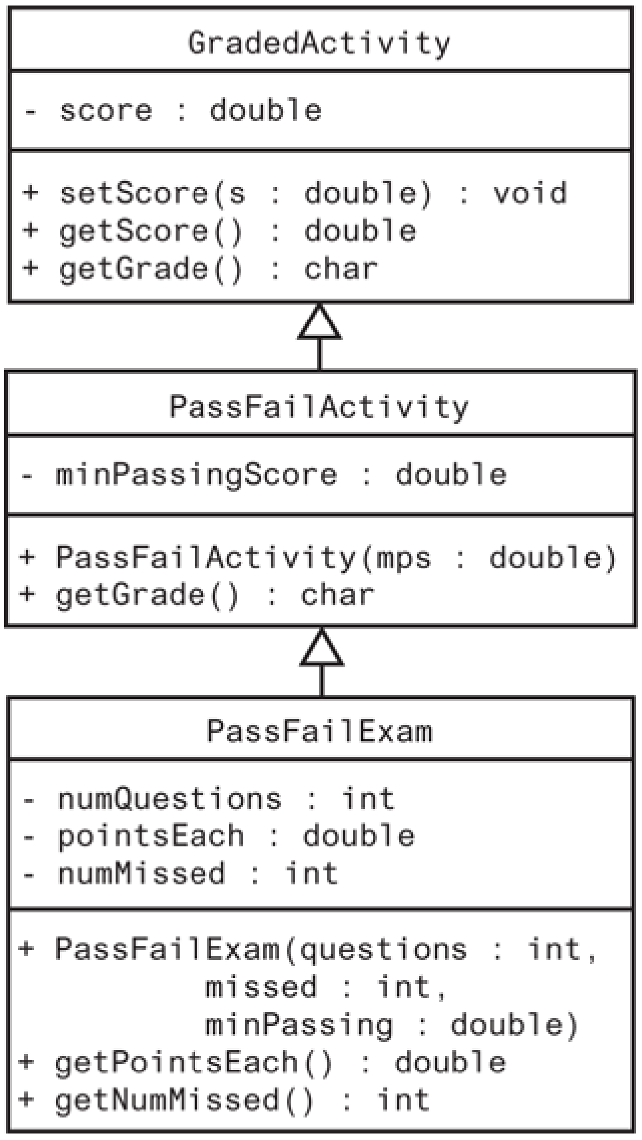
```
import java.util.Scanner;

/**
 * This program demonstrates the PassFailExam class.
 */

public class PassFailExamDemo
{
   public static void main(String[] args)
   {
      int questions,    // Number of questions
          missed;       // Number of questions missed
      double minPassing; // Minimum passing score

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Get the number of questions on the exam.
      System.out.print("How many questions are " +
                       "on the exam? ");
      questions = keyboard.nextInt();

      // Get the number of questions the student missed.
      System.out.print("How many questions did the " +
                       "student miss? ");
      missed = keyboard.nextInt();

      // Get the minimum passing score.
      System.out.print("What is the minimum " +
                    "passing score? ");
      minPassing = keyboard.nextInt();

      // Create a PassFailExam object.
      PassFailExam exam =
         new PassFailExam(questions, missed, minPassing);

      // Display the test results.
      System.out.println("Each question counts " +
                     exam.getPointsEach() +
                     " points.");
      System.out.println("The exam score is " +
                     exam.getScore());
      System.out.println("The exam grade is " +
                     exam.getGrade());
   }
}
```

Program Output with Example Input Shown in Bold

```
How many questions are on the exam? 100 Enter
How many questions did the student miss? 25 Enter
What is the minimum passing score? 60 Enter
Each question counts 1.0 points.
The exam score is 75.0
The exam grade is P
```

Figure 9-11 shows a UML diagram depicting the inheritance relationship among the `GradedActivity`, `PassFailActivity`, and `PassFailExam` classes.

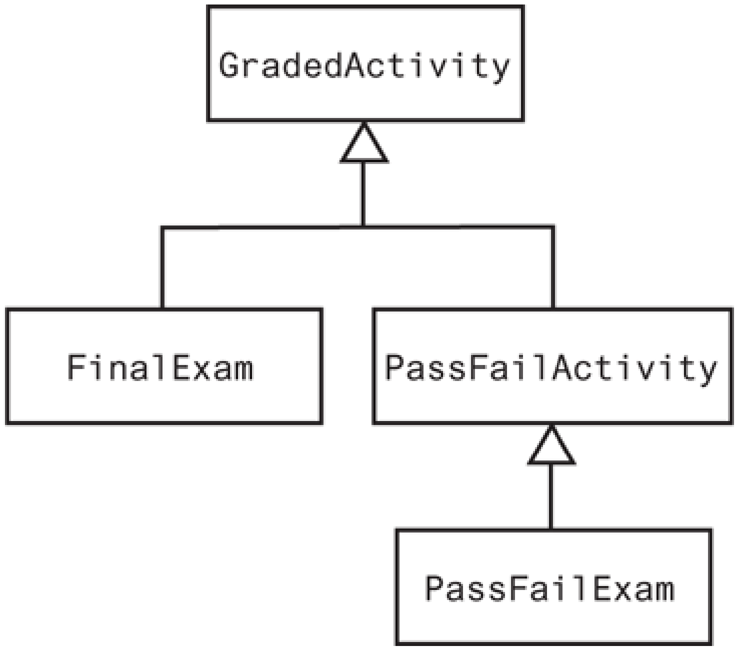
Figure 9-11 The `GradedActivity`, `PassFailActivity`, and `PassFailExam` classes



Class Hierarchies

Classes often are depicted graphically in a *class hierarchy*. Like a family tree, a class hierarchy shows the inheritance relationships among classes. **Figure 9-12** shows a class hierarchy for the `GradedActivity`, `FinalExam`, `PassFailActivity`, and `PassFailExam` classes. The more general classes are toward the top of the tree, and the more specialized classes are toward the bottom.

Figure 9-12 Class hierarchy



9.6 The Object Class

Concept:

The Java API has a class named Object, which all other classes directly or indirectly inherit from.

Every class in Java, including the ones in the API and the classes that you create, directly or indirectly inherit from a class named `Object`, which is part of the `java.lang` package. Here's how it happens: When a class does not use the `extends` key word to inherit from another class, Java automatically extends it from the `Object` class. For example, look at the following class declaration:

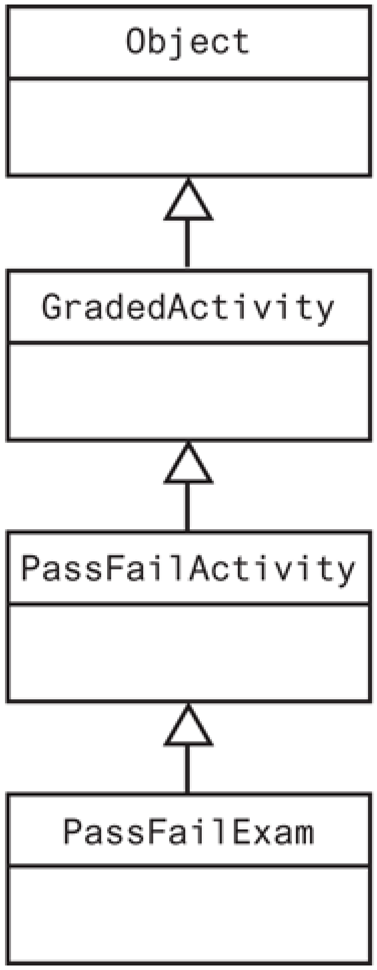
```
public class MyClass
{
   (Member Declarations . . .)
}
```

This class does not explicitly extend any other class, so Java treats it as though it was written as:

```
public class MyClass extends Object
{
   (Member Declarations . . .)
}
```

Ultimately, every class inherits from the `Object` class. **Figure 9-13** shows how the `PassFailExam` class ultimately inherits from `Object`.

Figure 9-13 The line of inheritance from `Object` to `PassFailExam`



Because every class directly or indirectly inherits from the `Object` class, every class inherits the `Object` class's members. Two of the most useful are the `toString` and `equals` methods. In **Chapter 6**, you learned that every object has a `toString` and an `equals` method, and now you know why! It is because those methods are inherited from the `Object` class.

In the `Object` class, the `toString` method returns a string containing the object's class name, followed by the `@` character, followed by the object's hexadecimal hashcode. (An object's hashcode is an integer that is unique to the object.) The `equals` method accepts the address of an object as its argument, and returns `true` if it is the same as the calling object's address. This is demonstrated in **Code Listing 9-24**.

Code Listing 9-24 `(ObjectMethods.java)`

```java
/**
 * This program demonstrates the toString and equals
 * methods that are inherited from the Object class.
 */

public class ObjectMethods
{
   public static void main(String[] args)
   {
     // Create two objects.
     PassFailExam exam1 = new PassFailExam(0, 0, 0);
     PassFailExam exam2 = new PassFailExam(0, 0, 0);

     // Send the objects to println, which will
     // call the toString method.
     System.out.println(exam1);
     System.out.println(exam2);

     // Test the equals method.
     if (exam1.equals(exam2))
       System.out.println("The two are the same.");
     else
       System.out.println("The two are not the same.");
   }
}
```

Program Output

```
 PassFailExam@45a877
PassFailExam@1372a1a
The two are not the same.
```

If you wish to change the behavior of either of these methods for a given class, you must override them in the class.

Checkpoint

9.16 Look at the following class definition:

```
public class ClassD extends ClassB
{
   (Member Declarations . . .)
}
```

Because `ClassD` inherits from `ClassB`, is it true that `ClassD` does not inherit from the `Object` class? Why or why not?

9.17 When you create a class, it automatically has a `toString` method and an equals method. Why?

9.7 Polymorphism

Concept:

A reference variable can reference objects of classes that inherit from the variable’s class.

VideoNote Polymorphism

Look at the following statement that declares a reference variable named `exam`:

```
GradedActivity exam;
```

This statement tells us that the `exam` variable's data type is `GradedActivity`. Therefore, we can use the `exam` variable to reference a `GradedActivity` object, as shown in the following statement:

```
exam = new GradedActivity();
```

The `GradedActivity` class is also used as the superclass for the `FinalExam` class. Because of the "is-a" relationship between a superclass and a subclass, an object of the `FinalExam` class is not just a `FinalExam` object. It is also a `GradedActivity` object. (A final exam *is a* graded activity.) Because of this relationship, we can use a `GradedActivity` variable to reference a `FinalExam` object. For example, look at the following statement:

```
GradedActivity exam = new FinalExam(50, 7);
```

This statement declares `exam` as a `GradedActivity` variable. It creates a `FinalExam` object and stores the object's address in the `exam` variable. This statement is perfectly legal, and will not cause an error message because a `FinalExam` object is also a `GradedActivity` object.

This is an example of polymorphism. The term *polymorphism* means the ability to take many forms. In Java, a reference variable is *polymorphic* because it can reference objects of types different from its own, as long as those types are related to its type through inheritance. All of the following declarations are legal because the `FinalExam`, `PassFailActivity`, and `PassFailExam` classes inherit from `GradedActivity`:

```
GradedActivity exam1 = new FinalExam(50, 7);
GradedActivity exam2 = new PassFailActivity(70);
GradedActivity exam3 = new PassFailExam(100, 10, 70);
```

Although a `GradedActivity` variable can reference objects of any class that inherits from `GradedActivity`, there is a limit to what the variable can do with those objects. Recall the `GradedActivity` class has three methods: `setScore`, `getScore`, and `getGrade`. So, a `GradedActivity` variable can be used to call only those three methods, regardless of the type of object the variable references. For example, look at the following code:

```
GradedActivity exam = new PassFailExam(100, 10, 70);
System.out.println(exam.getScore());    // This works.
System.out.println(exam.getGrade());    // This works.
System.out.println(exam.getPointsEach()); // ERROR! Won't work.
```

In this code, `exam` is declared as a `GradedActivity` variable and is assigned the address of a `PassFailExam` object. The `GradedActivity` class has only the `setScore`, `getScore`, and `getGrade` methods, so those are the only methods that the `exam` variable knows how to execute. The last statement in this code is a call to the `getPointsEach` method, which is defined in the `PassFailExam` class. Because the `exam` variable knows only about methods in the `GradedActivity` class, it cannot execute this method.

Polymorphism and Dynamic Binding

When a superclass variable references a subclass object, a potential problem exists. What if the subclass has overridden a method in the superclass, and the variable makes a call to that method? Does the variable call the superclass's version of the method, or the subclass's version? For example, look at the following code:

```
GradedActivity exam = new PassFailActivity(60);
exam.setScore(70);
System.out.println(exam.getGrade());
```

Recall that the `PassFailActivity` class inherits from the `GradedActivity` class, and it overrides the `getGrade` method. When the last statement calls the `getGrade` method, does it call the `GradedActivity` class's version (which returns `"A"`, `"B"`, `"C"`, `"D"`, or `"F"`) or does it call the `PassFailActivity` class's version (which returns `"P"` or `"F"`)?

Recall from **Chapter 6** the process of matching a method call with the correct method definition is known as binding. Java performs *dynamic binding* or *late binding* when a variable contains a polymorphic reference. This means the Java Virtual Machine determines at runtime which method to call, depending on the type of object that the variable references. So, it is the object's type that determines which method is called, not the variable's type. In this case, the `exam` variable references a `PassFailActivity` object, so the `PassFailActivity` class's version of the `getGrade` method is called. The last statement in this code will display a grade of `P`.

The program in **Code Listing 9-25** demonstrates polymorphic behavior. It declares an array of `GradedActivity` variables, then assigns the addresses of objects of various types to the elements of the array.

Code Listing 9-25 ***`(Polymorphic.java)`***

```java
/**
 * This program demonstrates polymorphic behavior.
 */

public class Polymorphic
{
   public static void main(String[] args)
   {
      // Create an array of GradedActivity references.
      GradedActivity[] tests = new GradedActivity[3];

      // The first test is a regular exam with a
      // numeric score of 95.
      tests[0] = new GradedActivity();
      tests[0].setScore(95);

      // The second test is a pass/fail test. The
      // student missed 5 out of 20 questions, and the
      // minimum passing grade is 60.
      tests[1] = new PassFailExam(20, 5, 60);

      // The third test is the final exam. There were
      // 50 questions and the student missed 7.
      tests[2] = new FinalExam(50, 7);

      // Display the grades.
      for (int index = 0; index < tests.length; index++)
      {
         System.out.println("Test " + (index+ 1) + ": " +
                   "score " + tests[index].getScore() +
                   ", grade " + tests[index].getGrade());
      }
   }
}
```

Program Output

```
Test 1: score 95.0, grade A
Test 2: score 75.0, grade P
Test 3: score 86.0, grade B
```

You can also use parameters to polymorphically accept arguments to methods. For example, look at the following method:

```
public static void displayGrades(GradedActivity g)
{
  System.out.println("Score " + g.getScore() +
           ", grade " + g.getGrade());
}
```

This method's parameter, `g`, is a `GradedActivity` variable. But, it can be used to accept arguments of any type that inherits from `GradedActivity`. For example, the following code passes objects of the `FinalExam`, `PassFailActivity`, and `PassFailExam` classes to the method:

```
GradedActivity exam1 = new FinalExam(50, 7);
GradedActivity exam2 = new PassFailActivity(70);
GradedActivity exam3 = new PassFailExam(100, 10, 70);
displayGrades(exam1);  // Pass a FinalExam object.
displayGrades(exam2);  // Pass a PassFailActivity object.
displayGrades(exam3);  // Pass a PassFailExam object.
```

The “Is-a” Relationship Does Not Work in Reverse

It is important to note that the “is-a” relationship does not work in reverse. Although the statement “a final exam is a graded activity” is true, the statement “a graded activity is a final exam” is not true. This is because not all graded activities are final exams. Likewise, not all `GradedActivity` objects are `FinalExam` objects. So, the following code will not work:

```
GradedActivity activity = new GradedActivity();
FinalExam exam = activity;  // ERROR!
```

You cannot assign the address of a `GradedActivity` object to a `FinalExam` variable. This makes sense because `FinalExam` objects have capabilities that go beyond those of a `GradedActivity` object. Interestingly, the Java compiler will let you make such an assignment if you use a type cast, as shown here:

```
GradedActivity activity = new GradedActivity();
FinalExam exam = (FinalExam) activity; // Will compile but not
run.
```

But, the program will crash when the assignment statement executes.

The `instanceof` Operator

There is an operator in Java named `instanceof` that you can use to determine whether an object is an instance of a particular class. Here is the general form of an expression that uses the `instanceof` operator:

```
refVar instanceof ClassName
```

In the general form, *refVar* is a reference variable, and *ClassName* is the name of a class. This is the form of a `boolean` expression that will return `true` if the object referenced by *refVar* is an instance of *ClassName*. Otherwise, the expression returns `false`. For example, the `if` statement in the following code determines whether the reference variable `activity` references a `GradedActivity` object:

```
GradedActivity activity = new GradedActivity();
if (activity instanceof GradedActivity)
  System.out.println("Yes, activity is a GradedActivity.");
else
  System.out.println("No, activity is not a GradedActivity.");
```

This code will display `"Yes, activity is a GradedActivity."`

The `instanceof` operator understands the “is-a” relationship that exists when a class inherits from another class. For example, look at the following code:

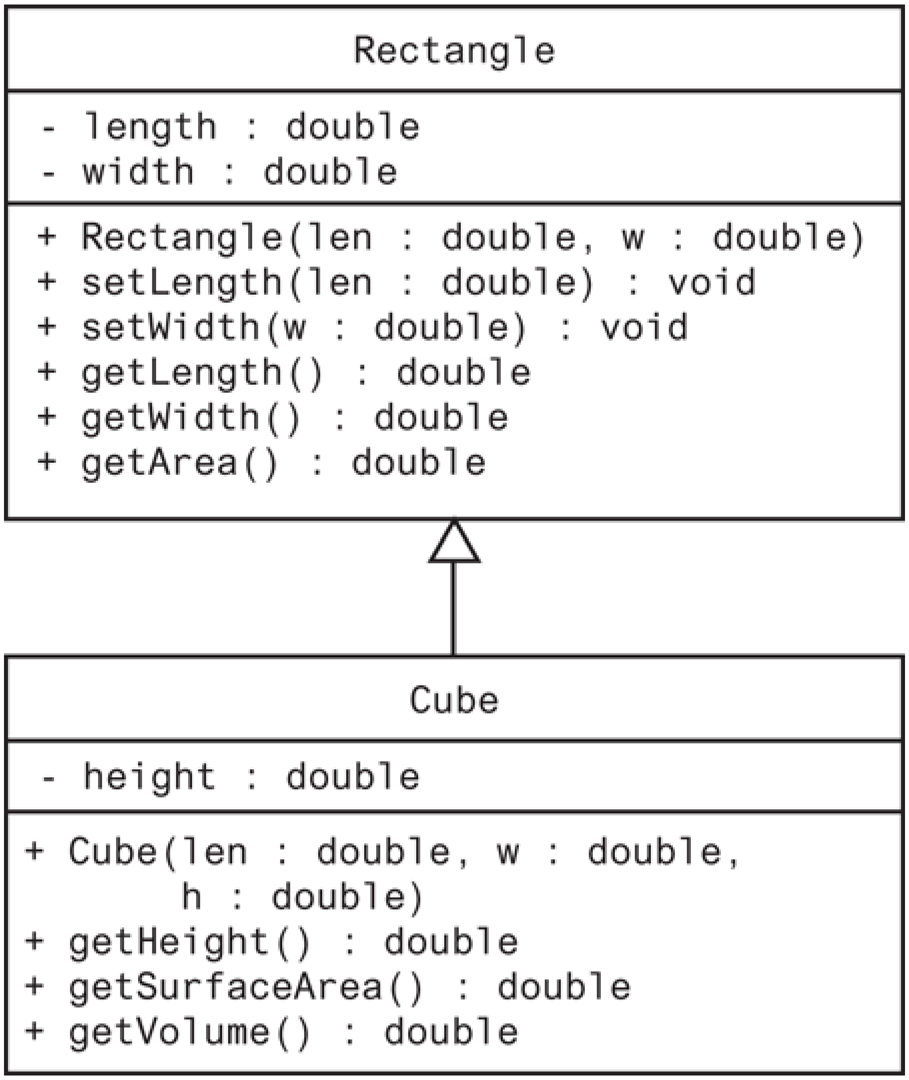
```
FinalExam exam = new FinalExam(20, 2);
if (exam instanceof GradedActivity)
  System.out.println("Yes, exam is a GradedActivity.");
else
  System.out.println("No, exam is not a GradedActivity.");
```

Even though the object referenced by `exam` is a `FinalExam` object, this code will display `"Yes, exam is a GradedActivity."` The `instanceof` operator returns `true` because `FinalExam` is a subclass of `GradedActivity`.

Checkpoint

9.18 Recall the `Rectangle` and `Cube` classes discussed earlier, as shown in **Figure 9-14**.

Figure 9-14 `Rectangle` and `Cube` classes



a. Is the following statement legal or illegal? If it is illegal, why?

```
Rectangle r = new Cube(10, 12, 5);
```

b. If you determined that the statement in Part a is legal, are the following statements legal or illegal? (Indicate legal or illegal for each statement.)

```
System.out.println(r.getLength());
System.out.println(r.getWidth());
System.out.println(r.getHeight());
System.out.println(r.getSurfaceArea());
```

c. Is the following statement legal or illegal? If it is illegal, why?

```
Cube c = new Rectangle(10, 12);
```

9.8 Abstract Classes and Abstract Methods

Concept:

An abstract class is not instantiated, but other classes inherit from it. An abstract method has no body and must be overridden in a subclass.

An *abstract method* is a method that appears in a superclass, but expects to be overridden in a subclass. An abstract method has only a header and no body. Here is the general format of an abstract method header:

```
AccessSpecifier abstract ReturnType MethodName(ParameterList);
```

Notice the key word `abstract` appears in the header, and the header ends with a semicolon. There is no body for the method. Here is an example of an abstract method header:

```
public abstract void setValue(int value);
```

When an abstract method appears in a class, the method must be overridden in a subclass. If a subclass fails to override the method, an error will result. Abstract methods are used to ensure that a subclass implements the method.

When a class contains an abstract method, you cannot create an instance of the class. Abstract methods are commonly used in abstract classes. An *abstract class* is not instantiated itself, but serves as a superclass for other classes. The abstract class represents the generic or abstract form of all the classes that inherit from it.

For example, consider a factory that manufactures airplanes. The factory does not make a generic airplane, but makes three specific types of airplanes: two different models of prop-driven planes, and one commuter jet model. The computer software that catalogs the planes might use an abstract class named `Airplane`. That class has members representing the common characteristics of all airplanes. In addition, the software has classes for each of the three specific airplane models the factory manufactures. These classes all inherit from the `Airplane` class, and they have members representing the unique characteristics of each type of plane. The `Airplane` class is never instantiated, but is used as a superclass for the other classes.

A class becomes abstract when you place the `abstract` key word in the class definition. Here is the general format:

```
public abstract class ClassName
```

For example, look at the following abstract class `Student` shown in **Code Listing 9-26**. It holds data common to all students, but does not hold all the data needed for students of specific majors.

Code Listing 9-26 `(Student.java)`

```java
/**
 * The Student class is an abstract class that holds general
 * data about a student. Classes representing specific types
 * of students should inherit from this class.
 */

public abstract class Student
{
   private String name;    // Student name
   private String idNumber; // Student ID
   private int yearAdmitted; // Year student was admitted

   /**
    * The Constructor accepts as arguments the student's
    * name, ID number, and the year admitted.
    */

   public Student(String n, String id, int year)
   {
    name = n;
    idNumber = id;
    yearAdmitted = year;
   }

   /**
   * toString method
   */

   public String toString()
   {
    String str;

    str = "Name: " + name +
       "\nID Number: " + idNumber +
       "\nYear Admitted: " + yearAdmitted;

    return str;
   }

   /**
   * The getRemainingHours method is abstract.
   * It must be overridden in a subclass.
   */

   public abstract int getRemainingHours();
 }
```

The `Student` class contains fields for storing a student's name, ID number, and year admitted. It also has a constructor, a `toString` method, and an abstract method named `getRemainingHours`.

This abstract method must be overridden in classes that inherit from the `Student` class. The idea behind this method is for it to return the number of hours remaining for a student to take in his or her major. It was made abstract because this class is intended to be the base for other classes that represent students of specific majors. For example, a `CompSciStudent` class might hold the data for a computer science student, and a `BiologyStudent` class might hold the data for a biology student. Computer science students must take courses in different disciplines than those taken by biology students. It stands to reason that the `CompSciStudent` class will calculate the number of hours remaining to be taken in a different manner than the `BiologyStudent` class. Let’s look at an example of the `CompSciStudent` class, which is shown in **Code Listing 9-27**.

Code Listing 9-27 `(CompSciStudent.java)`

```
/**
 * This class holds data for a computer science student.
 */

public class CompSciStudent extends Student
{
   // Constants for the math, computer science and
   // general education hours required for graduation.
   private final int MATH_HOURS = 20,
                     CS_HOURS = 40,
                     GEN_ED_HOURS = 60;

   private int mathHours, // Math hours taken
               csHours,  // Comp. sci. hours taken
               genEdHours; // General ed hours taken

   /**
    * The Constructor accepts as arguments the student's
    * name, ID number, and the year admitted.
    */

   public CompSciStudent(String n, String id, int year)
   {
    super(n, id, year);
   }

```

```
/**
* The setMathHours method accepts a value for
* the number of math hours taken.
*/

public void setMathHours(int math)
{
 mathHours = math;
}

/**
* The setCsHours method accepts a value for
* the number of computer science hours taken.
*/

public void setCsHours(int cs)
{
 csHours = cs;
}

/**
* The setGenEdHours method accepts a value for
* the number of general education hours taken.
*/

public void setGenEdHours(int genEd)
{
 genEdHours = genEd;
}

/**
* toString method
*/

@Override
public String toString()
{
 String str; // To hold a string

 // Create a string representing this computer
 // science student's hours taken.
 str = super.toString() +
    "\nMajor: Computer Science" +
    "\nMath Hours Taken: " + mathHours +
    "\nComputer Science Hours Taken: " + csHours +
    "\nGeneral Ed Hours Taken: " + genEdHours;

 // Return the string.
 return str;
}

/**
* The getRemainingHours method returns the
* the number of hours remaining to be taken.
*/

```

```
83   @Override
84   public int getRemainingHours()
85   {
86    int reqHours,    // Total required hours
87      remainingHours; // Remaining hours
88
89    // Calculate the total required hours.
90    reqHours = MATH_HOURS + CS_HOURS + GEN_ED_HOURS;
91
92    // Calculate the remaining hours.
93    remainingHours = reqHours - (mathHours + csHours +
94             genEdHours);
95
96    // Return the remaining hours.
97    return remainingHours;
98   }
99 }
```

The `CompSciStudent` class, which inherits from the `Student` class, declares the following `final` integer fields in lines 9 through 11: `MATH_HOURS`, `CS_HOURS`, and `GEN_ED_HOURS`. These fields hold the required number of math, computer science, and general education hours for a computer science student. It also declares the following fields in lines 13 through 15: `mathHours`, `csHours`, and `genEdHours`. These fields hold the number of math, computer science, and general education hours taken by the student. Mutator methods are provided to store values in these fields. In addition, the class overrides the `toString` method and the abstract `getRemainingHours` method. The program in **Code Listing 9-28** demonstrates the class.

Code Listing 9-28 `(CompSciStudentDemo.java)`

```java
/**
 * This program demonstrates the CompSciStudent class.
 */

public class CompSciStudentDemo
{
   public static void main(String[] args)
   {
     // Create a CompSciStudent object.
     CompSciStudent csStudent =
       new CompSciStudent("Jennifer Haynes",
                 "167W98337", 2018);

     // Store values for Math, CS, and General Ed hours.
     csStudent.setMathHours(12);
     csStudent.setCsHours(20);
     csStudent.setGenEdHours(40);

     // Display the student's data.
     System.out.println(csStudent);

     // Display the number of remaining hours.
     System.out.println("Hours remaining: " +
                csStudent.getRemainingHours());
   }
}
```

Program Output

```
Name: Jennifer Haynes
ID Number: 167W98337
Year Admitted: 2018
Major: Computer Science
Math Hours Taken: 12
Computer Science Hours Taken: 20
General Ed Hours Taken: 40
Hours remaining: 48
```

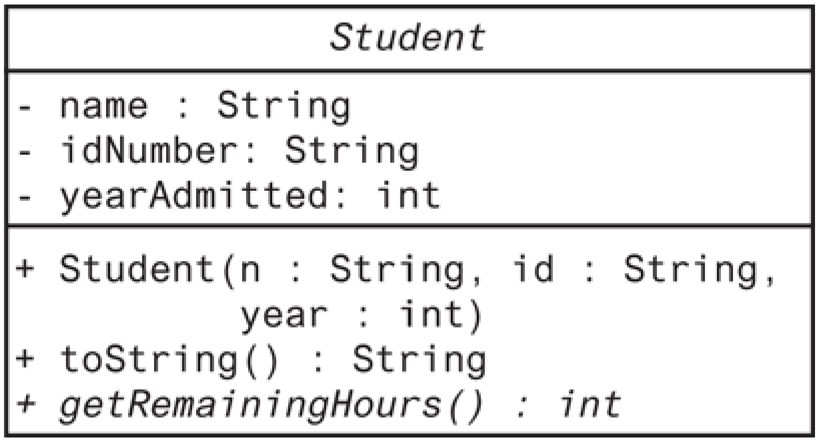
Remember the following points about abstract methods and classes:

- Abstract methods and abstract classes are defined with the `abstract` key word.
- Abstract methods have no body, and their header must end with a semicolon.
- An abstract method must be overridden in a subclass.
- When a class contains an abstract method, it cannot be instantiated. It must serve as a superclass.
- An abstract class cannot be instantiated. It must serve as a superclass.

Abstract Classes in UML

Abstract classes are drawn like regular classes in UML, except the name of the class and the names of abstract methods are shown in italics. For example, **Figure 9-15** shows a UML diagram for the `Student` class.

Figure 9-15 UML diagram for the `Student` class



Checkpoint

9.19 What is the purpose of an abstract method?

9.20 If a subclass extends a superclass with an abstract method, what must you do in the subclass?

9.21 What is the purpose of an abstract class?

9.22 If a class is defined as abstract, what can you not do with the class?

9.9 Interfaces

Concept:

An interface specifies behavior for a class.

In its simplest form, an interface is like a class that contains only abstract methods. An interface cannot be instantiated. Instead, it is *implemented* by other classes. When a class implements an interface, the class must override the methods that are specified by the interface.

An interface looks similar to a class, except the key word `interface` is used instead of the key word `class`, and the methods that are specified in an interface do not usually bodies, only headers that are terminated by semicolons. Here is the general format of an interface definition:

```
public interface InterfaceName
{
   (Method headers . . .)
}
```

Code Listing 9-29 shows an example of an interface named `Displayable`. In line 3, the interface specifies a `void` method named `display()`.

Code Listing 9-29 `(Displayable.java)`

```
1 public interface Displayable
2 {
3    void display();
4 }
```

Notice the `display` method header in line 3 does not have an access specifier. This is because all methods in an interface are implicitly `public`. You can optionally write `public` in the method header, but most programmers leave it out because all interface methods must be `public`.

Any class that implements the `Displayable` interface shown in **Code Listing 9-29** *must* provide an implementation of the `display` method (with the exact signatures specified by the interface, and with the same return type). The `Person` class shown in **Code Listing 9-30** is an example.

Code Listing 9-30 `(Person.java)`

```java
public class Person implements Displayable
{
   private String name;

   // Constructor
   public Person(String n)
   {
     name = n;
   }

   // display method
   public void display()
   {
     System.out.println("My name is " + name);
   }
}
```

When you want a class to implement an interface, you use the `implements` key word in the class header. Notice in line 1 of **Code Listing 9-30**, the `Person` class header ends with the clause `implements` `Displayable`. Because the `Person` class implements the `Displayable` interface, it must provide an implementation of the interface's `display` method. This is done in lines 12 through 15 of the `Person` class. The program in **Code Listing 9-31** demonstrates the `Person` class.

Code Listing 9-31 (InterfaceDemo.java)

```java
/**
 * This program demonstrates a class that implements
 * the Displayable interface.
 */

public class InterfaceDemo
{
   public static void main(String[] args)
   {
      // Create an instance of the Person class.
      Person p = new Person("Antonio");

      // Call the object's display method.
      p.display();
   }
}
```

Program Output

```
My name is Antonio
```

An Interface Is a Contract

When a class implements an interface, it is agreeing to provide all of the methods that are specified by the interface. It is often said that an interface is like a “contract,” and when a class implements an interface, it must adhere to the contract.

For example, **Code Listing 9-32** shows an interface named `Relatable`, which is intended to be used with the `GradedActivity` class presented earlier. This interface has three method headers: `equals`, `isGreater`, and `isLess`. Notice that each method accepts a `GradedActivity` object as its argument.

Code Listing 9-32 `(Relatable.java)`

```java
/**
 * Relatable interface
 */

public interface Relatable
{
   boolean equals(GradedActivity g);
   boolean isGreater(GradedActivity g);
   boolean isLess(GradedActivity g);
}
```

You might have guessed that the `Relatable` interface is named "Relatable" because it specifies methods that, presumably, make relational comparisons with `GradedActivity` objects. The intent is to make any class that implements this interface "relatable" with `GradedActivity` objects by ensuring that it has an `equals`, an `isGreater`, and an `isLess` method that perform relational comparisons. But, the interface specifies only the signatures for these methods, not what the methods should do. Although the programmer of a class that implements the `Relatable` interface can choose what those methods do, he or she should provide methods that comply with this intent.

Code Listing 9-33 shows the code for the `FinalExam3` class, which implements the `Relatable` interface. The `equals`, `isGreater`, and `isLess` methods compare the calling object with the object passed as an argument. The program in **Code Listing 9-34** demonstrates the class.

Code Listing 9-33 `(FinalExam3.java)`

```java
/**
 * This class determines the grade for a final exam.
 */

public class FinalExam3 extends GradedActivity implements
Relatable
{
   private int numQuestions;  // Number of questions
   private double pointsEach;  // Points for each question
   private int numMissed;    // Questions missed

   /**
    * The constructor sets the number of questions on the
    * exam and the number of questions missed.
    */

   public FinalExam3(int questions, int missed)
   {
      double numericScore;   // To hold a numeric score

      // Set the numQuestions and numMissed fields.
      numQuestions = questions;
      numMissed = missed;

      // Calculate the points for each question and
      // the numeric score for this exam.
      pointsEach = 100.0 / questions;
      numericScore = 100.0 - (missed * pointsEach);

      // Call the inherited setScore method to
      // set the numeric score.
      setScore(numericScore);
    }

    /**
    * The getPointsEach method returns the number of
    * points each question is worth.
    */

    public double getPointsEach()
    {
      return pointsEach;
    }

    /**
    * The getNumMissed method returns the number of
    * questions missed.
    */

    public int getNumMissed()
    {
      return numMissed;
    }

```

```
/**
* The equals method compares the calling object
* to the argument object for equality.
*/

public boolean equals(GradedActivity g)
{
 boolean status;

 if (this.getScore() == g.getScore())
   status = true;
 else
   status = false;

 return status;
}

/**
* The isGreater method determines whether the calling
* object is greater than the argument object.
*/

public boolean isGreater(GradedActivity g)
{
 boolean status;

 if (this.getScore() > g.getScore())
   status = true;
```

```java
      else
         status = false;

      return status;
   }

   /**
    * The isLess method determines whether the calling
    * object is less than the argument object.
    */

   public boolean isLess(GradedActivity g)
   {
      boolean status;

      if (this.getScore() < g.getScore())
         status = true;
      else
         status = false;

      return status;
   }
}
```

Code Listing 9-34 (RelatableExams.java)

```java
/**
 * This program demonstrates the FinalExam3 class which
 * implements the Relatable interface.
 */

public class RelatableExams
{
   public static void main(String[] args)
   {
      // Exam #1 had 100 questions and the student
      // missed 20 questions.
      FinalExam3 exam1 = new FinalExam3(100, 20);

      // Exam #2 had 100 questions and the student
      // missed 30 questions.
      FinalExam3 exam2 = new FinalExam3(100, 30);

      // Display the exam scores.
      System.out.println("Exam 1: " + exam1.getScore());
      System.out.println("Exam 2: " + exam2.getScore());

      // Compare the exam scores.
      if (exam1.equals(exam2))
         System.out.println("The exam scores are equal.");

      if (exam1.isGreater(exam2))
      System.out.println("The Exam 1 score is the highest.");

   if (exam1.isLess(exam2))
      System.out.println("The Exam 1 score is the lowest.");
  }
}
```

Program Output

```
Exam 1: 80.0
Exam 2: 70.0
The Exam 1 score is the highest.
```

Fields in Interfaces

An interface can contain field declarations, but all fields in an interface are treated as `final` and `static`. Because they automatically become `final`, you must provide an initialization value. For example, look at the following interface definition:

```
public interface Doable
{
  int FIELD1 = 1,
      FIELD2 = 2;
  (Method headers . . .)
}
```

In this interface, `FIELD1` and `FIELD2` are `final` `static` `int` variables. Any class that implements this interface has access to these variables.

Implementing Multiple Interfaces

You might be wondering why we need both abstract classes and interfaces because they are so similar to each other. The reason is that a class can directly inherit from only one superclass, but Java allows a class to implement multiple interfaces. When a class implements multiple interfaces, it must provide the methods specified by all of them.

To specify multiple interfaces in a class definition, simply list the names of the interfaces, separated by commas, after the `implements` key word. Here is the first line of an example of a class that implements multiple interfaces:

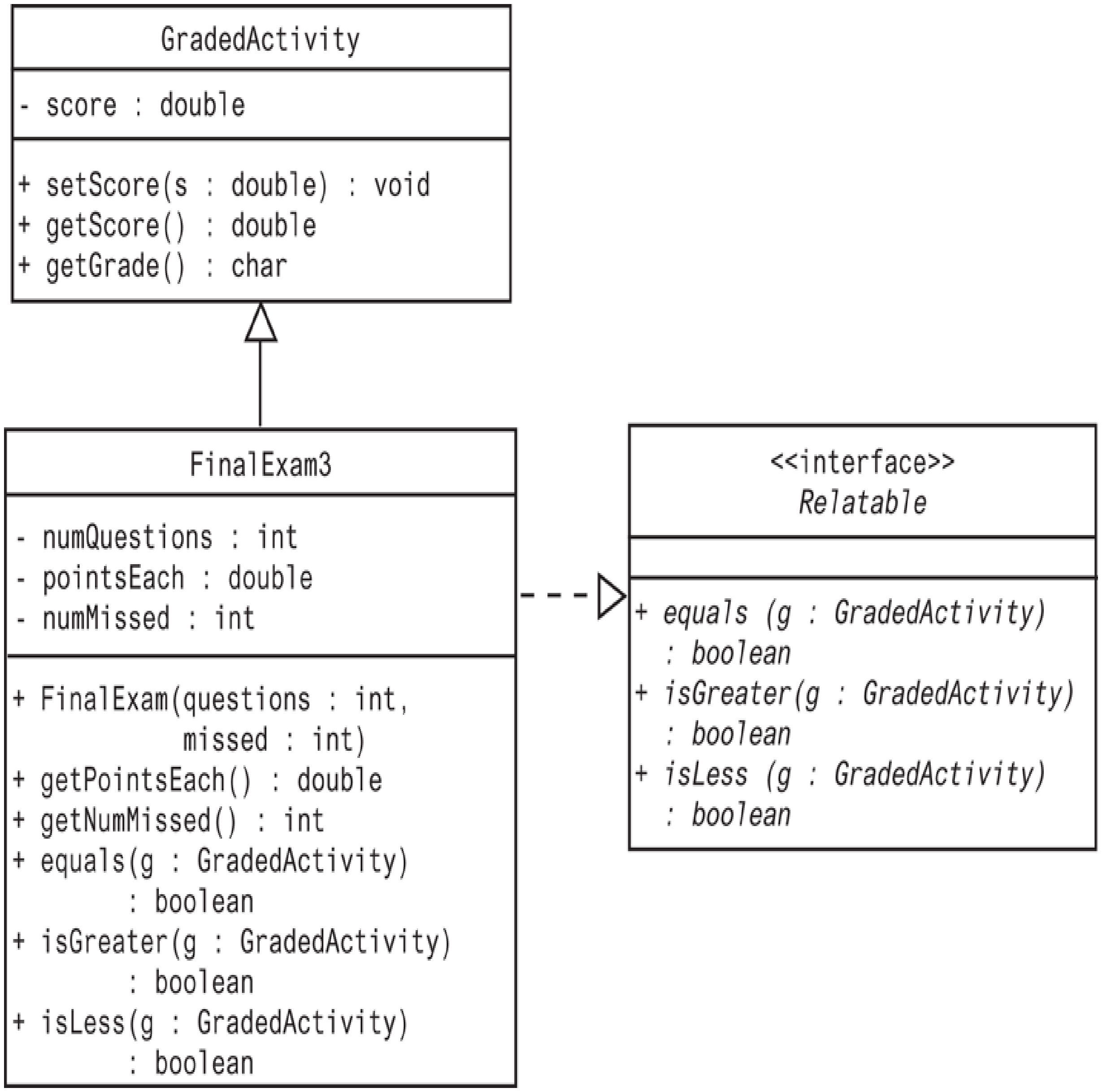
```
public class MyClass implements Interface1,
                                Interface2,
                                Interface3
```

This class implements three interfaces: `Interface1`, `Interface2`, and `Interface3`.

Interfaces in UML

In a UML diagram, an interface is drawn like a class, except the interface name and the method names are italicized, and the <<interface>> tag is shown above the interface name. The relationship between a class and an interface is known as a *realization relationship* (the class realizes the interfaces). You show a realization relationship in a UML diagram by connecting a class and an interface with a dashed line that has an open arrowhead at one end. The arrowhead points to the interface. This depicts the realization relationship. **Figure 9-16** shows a UML diagram depicting the relationships among the `GradedActivity` class, the `FinalExam3` class, and the `Relatable` interface.

Figure 9-16 Realization relationship in a UML diagram


GradedActivity
- score : double
+ setScore(s : double) : void
+ getScore() : double
+ getGrade() : char
FinalExam3
- numQuestions : int
- pointsEach : double
- numMissed : int
+ FinalExam(questions : int, missed : int)
+ getPointsEach() : double
+ getNumMissed() : int
+ equals(g : GradedActivity) : boolean
+ isGreater(g : GradedActivity) : boolean
+ isLess(g : GradedActivity) : boolean
<<interface>>
Relatable
+ equals (g : GradedActivity) : boolean
+ isGreater(g : GradedActivity) : boolean
+ isLess (g : GradedActivity) : boolean

Default Methods

Beginning in Java 8, interfaces may have *default methods*. A default method is an interface method that has a body. **Code Listing 9-35** shows another version of the `Displayable` interface, in which the `display` method is a default method.

Code Listing 9-35 (Displayable.java)

```
1 public interface Displayable
2 {
3   default void display()
4   {
5     System.out.println("This is the default display 
method.");
6   }
7 }
```

Notice in line 3 the method header begins with the key word `default`. This is required for an interface method that has a body. When a class implements an interface with a default method, the class can override the default method, but it is not required to. For example, the `Person` class shown in **Code Listing 9-36** implements the `Displayable` interface, but does not override the `display` method. The program in **Code Listing 9-37** instantiates the `Person` class, and calls the `display` method. As you can see from the program output, the code in the interface's default method is executed.

Code Listing 9-36 `(Person.java)`

```
/**
 * This class implements the Displayable
 * interface, but does not override the
 * default display method.
 */

public class Person implements Displayable
{
  private String name;

  // Constructor
  public Person(String n)
  {
    name = n;
  }
}
```

Code Listing 9-37

(InterfaceDemoDefaultMethod.java)

```java
/**
 * This program demonstrates a class that implements
 * the Displayable interface (with a default method).
 */

public class InterfaceDemoDefaultMethod
{
   public static void main(String[] args)
   {
      // Create an instance of the Person class.
      Person p = new Person("Antonio");

      // Call the object's display method.
      p.display();
   }
}
```

Program Output

```
This is the default display method.
```

Note:

One of the benefits of having default methods is that they allow you to add new methods to an existing interface without causing errors in the classes that already implement the interface. Prior to Java 8, when you added a new method header to an existing interface, all of the classes that already implement that interface had to be rewritten to override the new method. Now you can add default methods to an interface, and if an existing class (that implements the interface) does not need the new method, you do not have to rewrite the class.

Polymorphism and Interfaces

Just as you can create reference variables of a class type, Java allows you to create reference variables of an interface type. An interface reference variable can reference any object that implements that interface, regardless of its class type. This is another example of polymorphism. For example, look at the `RetailItem` interface in **Code Listing 9-38**.

Code Listing 9-38 `(RetailItem.java)`

```
/**
 * RetailItem interface
 */

public interface RetailItem
{
   public double getRetailPrice();
}
```

This interface specifies only one method: `getRetailPrice`. Both the `CompactDisc` and `DvdMovie` classes, shown in **Code Listings 9-39** and **9-40**, implement this interface.

Code Listing 9-39 `(CompactDisc.java)`

```
/**
 * Compact Disc class
 */

public class CompactDisc implements RetailItem
{
   private String title;       // The CD's title
   private String artist;      // The CD's artist
   private double retailPrice;  // The CD's retail price

   /**
    * Constructor
    */

   public CompactDisc(String cdTitle, String cdArtist,
double cdPrice)
   {
    title = cdTitle;
    artist = cdArtist;
    retailPrice = cdPrice;
   }

   /**
    * getTitle method
    */

```

```
  public String getTitle()
  {
   return title;
  }

  /**
  * getArtist method
  */

  public String getArtist()
  {
   return artist;
  }

  /**
  * getRetailPrice method (Required by the RetailItem
  * interface)
  */

  public double getRetailPrice()
  {
   return retailPrice;
  }
}
```

Code Listing 9-40 `(DvdMovie.java)`

```java
/**
 * DvdMovie class
 */

public class DvdMovie implements RetailItem
{
   private String title;      // The DVD's title
   private int runningTime;    // Running time in minutes
   private double retailPrice;  // The DVD's retail price

   /**
   * Constructor
   */

   public DvdMovie(String dvdTitle, int runTime, double
dvdPrice)
   {
    title = dvdTitle;
    runningTime = runTime;
    retailPrice = dvdPrice;
   }

   /**
   * getTitle method
   */

```

```
  public String getTitle()
  {
   return title;
  }

  /**
  * getRunningTime method
  */

  public int getRunningTime()
  {
   return runningTime;
  }

  /**
  * getRetailPrice method (Required by the RetailItem
  * interface)
  */

  public double getRetailPrice()
  {
   return retailPrice;
  }
}
```

Because they implement the `RetailItem` interface, objects of these classes can be referenced by a `RetailItem` reference variable. The following code demonstrates this:

```
RetailItem item1 = new CompactDisc("Songs From the Heart",
                                   "Billy Nelson",
                                   18.95);
RetailItem item2 = new DvdMovie("Planet X",
                                102,
                                22.95);
```

In this code, two `RetailItem` reference variables, `item1` and `item2`, are declared. The `item1` variable references a `CompactDisc` object, and the `item2` variable references a `DvdMovie` object. This is possible because both the `CompactDisc` and `DvdMovie` classes implement the `RetailItem` interface. When a class implements an interface, an inheritance relationship known as *interface inheritance* is established. Because of this inheritance relationship, a `CompactDisc` object *is a* `RetailItem`, and likewise, a `DvdMovie` object *is a* `RetailItem`. Therefore, we can create `RetailItem` reference variables and have them reference `CompactDisc` and `DvdMovie` objects.

The program in **Code Listing 9-41** demonstrates how an interface reference variable can be used as a method parameter.

Code Listing 9-41

(PolymorphicInterfaceDemo.java)

```java
/**
 * This program demonstrates that an interface type may
 * be used to create a polymorphic reference.
 */

public class PolymorphicInterfaceDemo
{
   public static void main(String[] args)
   {
      // Create a CompactDisc object.
      CompactDisc cd =
            new CompactDisc("Greatest Hits",
                            "Joe Looney Band",
                            18.95);
      // Create a DvdMovie object.
      DvdMovie movie =
            new DvdMovie("Wheels of Fury",
                          137, 12.95);

      // Display the CD's title.
      System.out.println("Item #1: " +
                         cd.getTitle());

      // Display the CD's price.
      showPrice(cd);

      // Display the DVD's title.
   System.out.println("Item #2: " +
             movie.getTitle());

   // Display the DVD's price.
   showPrice(movie);
  }

  /**
  * The showPrice method displays the price
  * of the RetailItem object that is passed
  * as an argument.
  */

  private static void showPrice(RetailItem item)
  {
   System.out.printf("Price: $%,.2f\n",
item.getRetailPrice());
  }
}
```

Program Output

```
 Item #1: Greatest Hits
Price: $18.95
Item #2: Wheels of Fury
Price: $12.95
```

There are some limitations to using interface reference variables. As previously mentioned, you cannot create an instance of an interface. The following code will cause a compiler error:

```
RetailItem item = new RetailItem(); // ERROR! Will not compile!
```

In addition, when an interface variable references an object, you can use the interface variable to call only the methods that are specified in the interface. For example, look at the following code:

```
// Reference a CompactDisc object with a RetailItem variable.
RetailItem item = new CompactDisc("Greatest Hits",
                                  "Joe Looney Band",
                                  18.95);
// Call the getRetailPrice method...
System.out.println(item.getRetailPrice()); // OK, this works.
// Attempt to call the getTitle method...
System.out.println(item.getTitle()); // ERROR! Will not
compile!
```

The last line of code will not compile because the `RetailItem` interface specifies only one method: `getRetailPrice`. So, we cannot use a `RetailItem` reference variable to call any other method.[1]

[1] Actually, it is possible to cast an interface reference variable to the type of the object it references, then call methods that are members of that type. The syntax is somewhat awkward, however. The statement that causes the compiler error in the example code could be rewritten as:

```
System.out.println(((CompactDisc)item).getTitle());
```

Checkpoint

9.23 What is the purpose of an interface?

9.24 How is an interface similar to an abstract class?

9.25 How is an interface different from an abstract class, or any class?

9.26 If an interface has fields, how are they treated?

9.27 Write the first line of a class named `Customer`, which implements an interface named `Relatable`.

9.28 Write the first line of a class named `Employee`, which implements `interfaces` named `Payable` and `Listable`.

9.10 Anonymous Inner Classes

Concept:

An inner class is a class that is defined inside another class. An anonymous inner class is an inner class that has no name. An anonymous inner class must implement an interface, or extend another class.

Sometimes you need a class that is simple, and to be instantiated only once in your code. When this is the case, you can use an anonymous inner class. An *anonymous inner class* is a class that has no name. It is called an inner class because it is defined inside another class. You use the `new` operator to simultaneously define an anonymous inner class and create an instance of it. Here is the general syntax for instantiating and defining an anonymous inner class:

```
new ClassOrInterfaceName() {
     (Fields and methods of the anonymous class...)
     }
```

The `new` operator is followed by the name of an existing class or interface, followed by a set of parentheses. Next, you write the body of the class, enclosed in curly braces. The expression creates an object that is an instance of a class that either extends the specified superclass, or implements the specified interface. A reference to the object is returned. (Notice you do *not* use the `extends` or `implements` key words in the expression.)

Before you look at an example, you must understand a few requirements and restrictions:

- An anonymous inner class must either implement an interface, or extend another class.
- If the anonymous inner class extends a superclass, the superclass’s no-arg constructor is called when the object is created.
- An anonymous inner class must override all of the abstract methods specified by the interface it is implementing, or the superclass it is extending.
- Because an anonymous inner class’s definition is written inside a method, it can access that method’s local variables, but only if

they are declared `final`, or if they are effectively `final`. (An effectively `final` variable is a variable whose value is never changed.) A compiler error will result if an anonymous inner class tries to use a variable that is not `final`, or not effectively `final`.

Let’s look at an example of an anonymous inner class that implements an interface. Suppose we have the interface shown in **Code Listing 9-42**.

Code Listing 9-42 `(IntCalculator.java)`

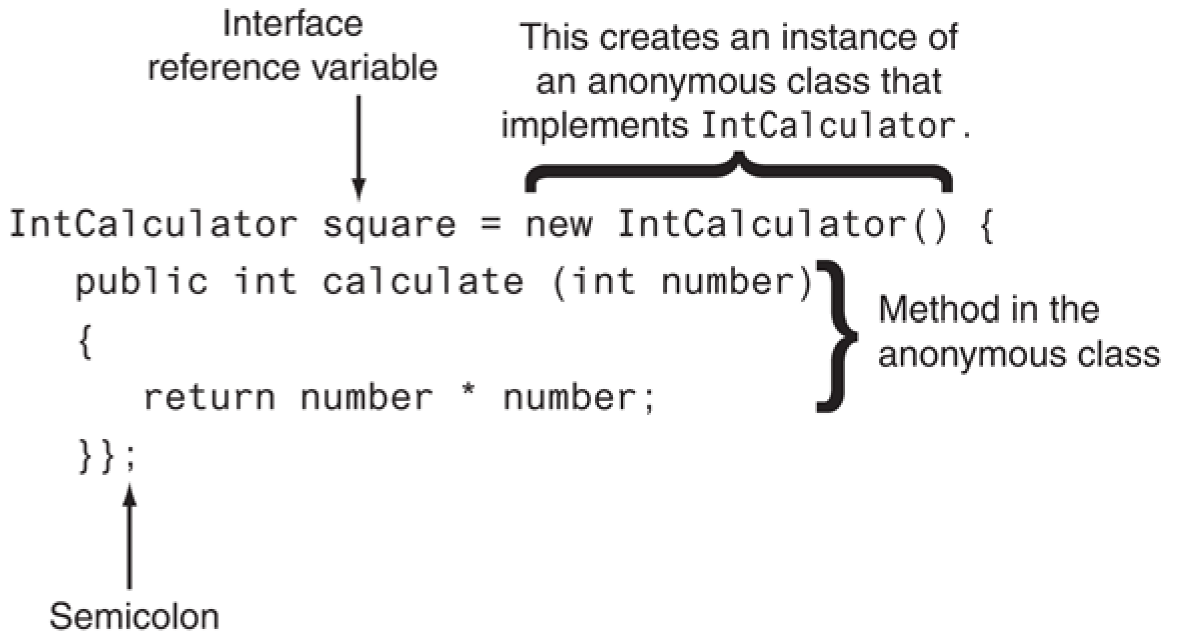
```
interface IntCalculator
{
   int calculate(int number);
}
```

The name of the interface is `IntCalculator`, and it specifies a method named `calculate`. The `calculate` method accepts an `int` argument, and returns an `int` value. Suppose we want to define a class that implements the `IntCalculator` interface, and overrides the `calculate` method so that it returns the square of the argument that is passed to it. The following code snippet shows how:

```
IntCalculator square = new IntCalculator() {
    public int calculate(int number)
    {
      return number * number;
    }};
```

The first line of the code snippet declares a variable named `square`, that is an `IntCalculator` reference variable (meaning that it can refer to any object that implements `IntCalculator`). On the right side of the `=` sign, the expression `new IntCalculator()` creates an instance of an anonymous class that implements the `IntCalculator` interface. The body of the anonymous class appears next, enclosed inside curly braces. In the class body, the `calculate` method is overridden. Because this is a complete statement, it ends with a semicolon. **Figure 9-21** illustrates the different parts of the statement, and **Code Listing 9-43** shows a complete program that uses it.

Figure 9-21 Creating an instance of an anonymous inner class


Interface reference variable
This creates an instance of an anonymous class that implements IntCalculator.
Method in the anonymous class
Semicolon

```
IntCalculator square = new IntCalculator() {
   public int calculate (int number)
   {
      return number * number;
   }};
```

Code Listing 9-43 `(AnonymousClassDemo.java)`

```java
/**
 * This program demonstrates an anonymous inner class
 */

import java.util.Scanner;

public class AnonymousClassDemo
{
  public static void main(String[] args)
  {
   int num;

   // Create a Scanner object for keyboard input.
   Scanner keyboard = new Scanner(System.in);

   // Create an object that implements IntCalculator.
   IntCalculator square = new IntCalculator() {
     public int calculate(int number)
     {
      return number * number;
     }};

   // Get a number from the user.
   System.out.print("Enter an integer number: ");
   num = keyboard.nextInt();

27     // Display the square of the number.
28     System.out.println("The square is " +
square.calculate(num));
29    }
30 }
```

Program Output with Example Input Shown in Bold

```
Enter an integer number: 5 [Enter]
The square is 25
```

Let’s take a closer look at the program:

- Line 11 declares an `int` variable named `num`, that will be used to hold user input.
- Line 14 creates a `Scanner` object for keyboard input.
- Lines 17 through 21 instantiate an anonymous inner class that implements the `IntCalculator` interface. A variable named `square` is used to reference the object. In the class body, lines 18 through 21 override the `calculate` method to return the square of the method’s argument.
- Line 24 prompts the user to enter an integer number, and line 25 reads the number from the keyboard. The number is assigned to the `num` variable.

- Line 28 calls the `square` object’s `calculate` method, and displays the return value in a message.

9.11 Functional Interfaces and Lambda Expressions

Concept:

A functional interface is an interface that has one abstract method. You can use a special type of expression, known as a lambda expression, to create an object that implements a functional interface.

Java 8 introduced two new features that work together to simplify code, particularly in situations where you might use anonymous inner classes. These new features are functional interfaces and lambda expressions. A *functional interface* is simply an interface that has one abstract method. For example, let's take another look at the `IntCalculator` interface that we previously discussed in the section on anonymous classes. Because it has only one abstract method, it is considered a functional interface. For your convenience, **Code Listing 9-44** shows the code for the interface.

Code Listing 9-44 `(IntCalculator.java)`

```java
interface IntCalculator
{
   int calculate(int number);
}
```

The name of the interface is `IntCalculator`, and it specifies one method named `calculate`. The `calculate` method accepts an `int` argument, and returns an `int` value.

Because `IntCalculator` is a functional interface, we do not have to go to the trouble of defining a class that implements the interface. We do not even have to use an anonymous inner class. Instead, we can use a *lambda expression* to create an object that implements the interface, and overrides its abstract method.

You can think of a lambda expression as an anonymous method, or a method with no name. Like regular methods, lambda expressions can accept arguments and return values. Here is the general format of a simple lambda expression that accepts one argument, and returns a value:

```
parameter -> expression
```

In this general format, the lambda expression begins with a parameter variable, followed by the *lambda operator* (`->`), followed by an expression that has a value. Here is an example:

```
x -> x * x
```

The `x` that appears on the left side of the `->` operator is the name of a parameter variable, and the expression `x * x` that appears on the right side of the `->` operator is the value that is returned.

This lambda expression works like a method that has a parameter variable named `x`, and it returns the value of `x * x`.

We can use this lambda expression to create an object that implements the `IntCalculator` interface. Here is an example:

```
IntCalculator square = x -> x * x;
```

On the left side of the `=` operator, we declare an `IntCalculator` reference variable named `square`. On the right side of the `=` operator, we have a lambda expression that creates an object with the following characteristics:

- Because we are assigning the object to an `IntCalculator` reference variable, the object automatically implements the `IntCalculator` interface.
- Because the `IntCalculator` interface has only one abstract method (named `calculate`), the lambda expression will be used to implement that one method.
- The parameter `x` that is used in the lambda expression represents the argument that is passed to the `calculate` method. We do not have to specify the data type of `x` because the compiler will determine it. Because the `calculate` method (in `theIntCalculator` interface) has an `int` parameter, the `x` parameter in the lambda expression will automatically become an `int`.
- The expression `x * x` is the value that is returned from the `calculate` method.

Code Listing 9-45 shows a complete program that uses the previously shown statement to create an object.

Code Listing 9-45 `(LambdaDemo.java)`

```
/**
 * This program demonstrates a simple
 * lambda expression.
 */

import java.util.Scanner;

public class LambdaDemo
{
   public static void main(String[] args)
   {
      int num;

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Create an object that implements IntCalculator.
      IntCalculator square = x -> x * x;

      // Get a number from the user.
      System.out.print("Enter an integer number: ");
      num = keyboard.nextInt();

      // Display the square of the number.
      System.out.println("The square is " +
square.calculate(num));
26    }
27 }
```

Program Output with Example Input Shown in Bold

```
Enter an integer number: 5 Enter
The square is 25
```

Let’s take a closer look at the program:

- Line 12 declares an `int` variable named `num`, which will be used to hold user input.
- Line 15 creates a `Scanner` object for keyboard input.
- Line 18 uses a lambda expression to create an object that implements the `IntCalculator` interface. A variable named `square` is used to reference the object. The object’s `calculate` method will return the square of the method’s argument.
- Line 21 prompts the user to enter an integer number, and line 22 reads the number from the keyboard. The number is assigned to the `num` variable.
- Line 25 calls the `square` object’s `calculate` method, and displays the return value in a message.

Lambda expressions provide a way to easily create and instantiate anonymous inner classes. If you compare **Code Listing 9-45** with the program shown in **Code Listing 9-43**, you can see that the lambda expression is much more concise than the anonymous inner class declaration.

Lambda Expressions That Do Not Return a Value

If a functional interface's abstract method is `void` (does not return a value), any lambda expression that you use with the interface should also be `void`. Here is an example:

```
x -> System.out.println(x);
```

This lambda expression has a parameter, `x`. When the expression is invoked, it displays the value of `x`.

Lambda Expressions with Multiple Parameters

If a functional interface’s abstract method has multiple parameters, any lambda expression that you use with the interface must also have multiple parameters. To use more than one parameter in a lambda expression, simply write a comma-separated list, then enclose the list in parentheses. Here is an example:

```
(a, b) -> a + b;
```

This lambda expression has two parameters, `a` and `b`. The expression returns the value of `a + b`.

Lambda Expressions with No Parameters

If a functional interface’s abstract method has no parameters, any lambda expression that you use with the interface must also have no parameters. Simply write a set of empty parentheses as the parameter list, as shown here:

```
() -> System.out.println();
```

When this lambda expression is invoked, it simply prints a blank line.

Explicitly Declaring a Parameter’s Data Type

You do not have to specify the data type of a lambda expression’s parameter because the compiler will determine it from the interface’s abstract method header. However, you can explicitly declare the data type of a parameter, if you wish. Here is an example:

```
(int x) -> x * x;
```

Note the parameter declaration (on the left side of the `->` operator) must be enclosed in parentheses. Here is another example, involving two parameters:

```
(int a, int b) -> a + b;
```

Using Multiple Statements in the Body of a Lambda Expression

You can write multiple statements in the body of a lambda expression, but if you do, you must enclose the statements in a set of curly braces, and you must write a `return` statement if the expression returns a value. Here is an example:

```
(int x) -> {
   int a = x * 2;
   return a;
};
```

Accessing Variables Within a Lambda Expression

A lambda expression can access variables that are declared in the enclosing scope, as long as those variables are `final`, or effectively `final`. An *effectively* `final` variable is a variable whose value is never changed, but it isn't declared with the `final` key word.

In **Code Listing 9-46**, the `main` method uses a lambda expression that accesses a `final` variable named `factor` that is local to the `main` method.

Code Listing 9-46 `(LabdaDemo2.java)`

```java
/**
 * This program demonstrates a lambda expression
 * that uses a final local variable.
 */

import java.util.Scanner;

public class LambdaDemo2
{
   public static void main(String[] args)
   {
      final int factor = 10;
      int num;

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Create an object that implements IntCalculator.
      IntCalculator multiplier = x -> x * factor;

      // Get a number from the user.
      System.out.print("Enter an integer number: ");
      num = keyboard.nextInt();

      // Display the number multiplied by 10.
      System.out.println("Multiplied by 10, that number is "
+
27                  multiplier.calculate(num));
28    }
29 }
```

Program Output with Example Input Shown in Bold

```
 Enter an integer number: 10 Enter
Multiplied by 10, that number is 100
```

In **Code Listing 9-46**, we could remove the `final` key word from the variable declaration in line 12, and the program would still compile and execute correctly. This is because the `factor` variable is never modified, and therefore is effectively `final`.

9.12 Common Errors to Avoid

The following list describes several errors that are commonly made when learning this chapter's topics:

- **Attempting to directly access a private superclass member from a subclass.** Private superclass members cannot be directly accessed by a method in a subclass. The subclass must call a public or protected superclass method in order to access the superclass's private members.
- **Forgetting to explicitly call a superclass constructor when the superclass has no default constructor or programmer-defined no-arg constructor.** When a superclass does not have a default constructor or a no-arg constructor, the subclass's constructor must explicitly call one of the constructors that the superclass does have.
- **Allowing the superclass's default constructor or no-arg constructor to be implicitly called when you intend to call another superclass constructor.** If a subclass's constructor does not explicitly call a superclass constructor, Java automatically calls `super()`.
- **Forgetting to precede a call to an overridden superclass method with super.** When a subclass method calls an overridden superclass method, it must precede the method call with the key word super and a dot (.). Failing to do so results in the subclass's version of the method being called.

- **Forgetting a class member’s access specifier.** When you do not give a class member an access specifier, it is granted package access by default. This means that any method in the same package can access the member.
- **Writing a body for an abstract method.** An abstract method cannot have a body. It must be overridden in a subclass.
- **Forgetting to terminate an abstract method’s header with a semicolon.** An abstract method header does not have a body, and it must be terminated with a semicolon.
- **Failing to override an abstract method.** An abstract method must be overridden in a subclass.
- **Overloading an abstract method instead of overriding it.** Overloading is not the same as overriding. When a superclass has an abstract method, the subclass must have a method with the same signature as the abstract method.
- **Trying to instantiate an abstract class.** You cannot create an instance of an abstract class.
- **Implementing an interface but forgetting to override all of its methods.** When a class implements an interface, all of the methods specified by the interface must be overridden in the class.
- **Overloading an interface method instead of overriding it.** As previously mentioned, overloading is not the same as overriding. When a class implements an interface, the class must have methods with the same signature as the methods specified in the interface.

Review Questions and Exercises

Multiple Choice and True/False

1. In an inheritance relationship, this is the general class.
    a. subclass
    b. superclass
    c. derived class
    d. child class

2. In an inheritance relationship, this is the specialized class.
    a. superclass
    b. base class
    c. subclass
    d. parent class

3. This key word indicates that a class inherits from another class.
    a. `derived`
    b. `specialized`
    c. `based`
    d. `extends`

4. A subclass does not have access to these superclass members.
    a. public
    b. private
    c. protected
    d. all of these

5. This key word refers to an object’s superclass.
    a. `super`
    b. `base`
    c. `this`
    d. `parent`

6. In a subclass constructor, a call to the superclass constructor must ________________.
    a. appear as the very first statement
    b. appear as the very last statement
    c. appear between the constructor’s header and the opening brace
    d. not appear

7. The following is an explicit call to the superclass’s default constructor.
    a. `default();`
    b. `class();`
    c. `super();`
    d. `base();`

8. A method in a subclass having the same signature as a method in the superclass is an example of ________________.
    a. overloading
    b. overriding
    c. composition
    d. an error

9. A method in a subclass having the same name as a method in the superclass but a different signature is an example of ________________.
   a. overloading
   b. overriding
   c. composition
   d. an error

10. These superclass members are accessible to subclasses and classes in the same package.
    a. private
    b. public
    c. protected
    d. all of these

11. All classes directly or indirectly inherit from this class.
    a. `Object`
    b. `Super`
    c. `Root`
    d. `Java`

12. With this type of binding, the Java Virtual Machine determines at runtime which method to call, depending on the type of the object that a variable references.
    a. static
    b. early
    c. flexible
    d. dynamic

13. When a class implements an interface, it must do which of the following?
    a. Overload all of the methods listed in the interface
    b. Provide all of the nondefault methods that are listed in the interface, with the exact signatures specified
    c. Not have a constructor
    d. Be an abstract class

14. Fields in an interface are _________________.
    a. `final`
    b. `static`
    c. both `final` and `static`
    d. not allowed

15. Abstract methods must be _________________.
    a. overridden
    b. overloaded
    c. deleted and replaced with real methods
    d. declared as `private`

16. Abstract classes cannot _________________.
    a. be used as superclasses
    b. have abstract methods
    c. be instantiated
    d. have fields

17. You use the ___________________ operator to define an anonymous inner class.

    a. `class`
    b. `inner`
    c. `new`
    d. `anonymous`

18. An anonymous inner class must ________________.
    a. be a superclass
    b. implement an interface
    c. extend a superclass
    d. either b or c.

19. A functional interface is an interface with ________________.
    a. only one abstract method
    b. no abstract methods
    c. only private methods
    d. no name

20. You can use a lambda expression to instantiate an object that ________________.
    a. has no constructor
    b. extends any superclass
    c. implements a functional interface
    d. does not implement an interface

21. **True or False:** Constructors are not inherited.
22. **True or False:** In a subclass, a call to the superclass constructor can be written only in the subclass constructor.

23. **True or False:** If a subclass constructor does not explicitly call a superclass constructor, Java will not call any of the superclass's constructors.
24. **True or False:** An object of a superclass can access members declared in a subclass.
25. **True or False:** The superclass constructor always executes before the subclass constructor.
26. **True or False:** When a method is declared with the `final` modifier, it must be overridden in a subclass.
27. **True or False:** A superclass has a member with package access. A class that is outside the superclass's package, but inherits from the superclass, can access this member.
28. **True or False:** A superclass reference variable can reference an object of a class that inherits from the superclass.
29. **True or False:** A subclass reference variable can reference an object of the superclass.
30. **True or False:** When a class contains an abstract method, the class cannot be instantiated.
31. **True or False:** A class can implement only one interface.
32. **True or False:** By default, all members of an interface are public.

Find the Error

Find the error in each of the following code segments.

1.

```
// Superclass
        public class Vehicle
        {
          (Member declarations . . .)
        }
        // Subclass
        public class Car expands Vehicle
        {
          (Member declarations . . .)
        }
```

2.

```
// Superclass
        public class Vehicle
        {
           private double cost;
          (Other methods . . .)
        }
        // Subclass
        public class Car extends Vehicle
        {
          public Car(double c)
          {
            cost = c;
          }
        }
```

3.
```
// Superclass
        public class Vehicle
        {
          private double cost;
          public Vehicle(double c)
          {
            cost = c;
          }
          (Other methods . . .)
        }
        // Subclass
        public class Car extends Vehicle
        {
          private int passengers;
          public Car(int p)
          {
            passengers = c;
          }
          (Other methods . . .)
        }
```

4.
```
// Superclass
        public class Vehicle
        {
         public abstract double getMilesPerGallon();
         (Other methods . . .)
        }
        // Subclass
        public class Car extends Vehicle
        {
         private int mpg;

         public int getMilesPerGallon();
          {
           return mpg;
          }
         (Other methods . . .)
        }
```

Algorithm Workbench

1. Write the first line of the definition for a `Poodle` class. The class should inherit from the `Dog` class.
2. Look at the following code which is the first line of a class definition:

   ```
   public class Tiger extends Felis
   ```

   In what order will the class constructors execute?
3. Write the declaration for class `B`. The class's members should be:
   - `m`, an integer. This variable should not be accessible to code outside the class or to any class that inherits from class `B`.
   - `n`, an integer. This variable should be accessible only to classes that inherit from class `B` or in the same package as class `B`.
   - `setM`, `getM`, `setN`, and `getN`. These are the mutator and accessor methods for the member variables `m` and `n`. These methods should be accessible to code outside the class.
   - `calc`. This is a public abstract method.

Next write the declaration for class `D`, which inherits from class `B`. The class's members should be:

- `q`, a `double`. This variable should not be accessible to code outside the class.
- `r`, a `double`. This variable should be accessible to any class that extends class `D` or in the same package.
- `setQ`, `getQ`, `setR`, and `getR`. These are the mutator and accessor methods for the member variables `q` and `r`. These methods should be accessible to code outside the class.
- `calc`. a public method that overrides the superclass's abstract `calc` method. This method should return the value of `q` times `r`.

4. Write the statement that calls a superclass constructor and passes the arguments `x`, `y`, and `z`.
5. A superclass has the following method:

```
public void setValue(int v)
{
  value = v;
}
```

   Write a statement that can appear in a subclass that calls this method, passing 10 as an argument.
6. A superclass has the following abstract method:

```
public abstract int getValue();
```

Write an example of a `getValue` method that can appear in a subclass.

7. Write the first line of the definition for a `Stereo` class. The class should inherit from the `SoundSystem` class, and it should implement the `CDPlayable`, `TunerPlayable`, and `MP3Playable` interfaces.
8. Write an interface named `Nameable` that specifies the following methods:

```
public void setName(String n)
public String getName()
```

9. Look at the following interface:

```
public interface Computable
{
  void compute(double x);
}
```

Write a statement that uses a lambda expression to create an object that implements the `Computable` interface. The object’s name should be `half`. The `half` object’s `compute` method should return the value of the `x` parameter divided by 2.

Short Answer

1. What is an “is-a” relationship?
2. A program uses two classes: `Animal` and `Dog`. Which class is the superclass and which is the subclass?
3. What is the superclass and what is the subclass in the following line?

   ```
   public class Pet extends Dog
   ```

4. What is the difference between a protected class member and a private class member?
5. Can a subclass ever directly access the private members of its superclass?
6. Which constructor is called first, that of the subclass or the superclass?
7. What is the difference between overriding a superclass method and overloading a superclass method?
8. Reference variables can be polymorphic. What does this mean?
9. When does dynamic binding take place?
10. What is an abstract method?
11. What is an abstract class?
12. What are the differences between an abstract class and an interface?

13. When you instantiate an anonymous inner class, the class must do one of two things. What are they?
14. What is a functional interface?
15. What is a lambda expression?

Programming Challenges

1. **`Employee` and `ProductionWorker` Classes**

   Design a class named `Employee`. The class should keep the following information in fields:

   - Employee name
   - Employee number in the format XXX–L, where each X is a digit within the range 0–9, and the L is a letter within the range A–M.
   - Hire date

   VideoNote The Employee and ProductionWorker Classes Problem

   Write one or more constructors and the appropriate accessor and mutator methods for the class.

   Next, write a class named `ProductionWorker` that inherits from the `Employee` class. The `ProductionWorker` class should have fields to hold the following information:

   - Shift (an integer)

   - Hourly pay rate (a `double`)

   The workday is divided into two shifts: day and night. The shift field will be an integer value representing the shift that the employee works. The day shift is shift 1, and the night shift is shift 2. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the classes by writing a program that uses a `ProductionWorker` object.

2. **`ShiftSupervisor` Class**

   In a particular factory, a shift supervisor is a salaried employee who supervises a shift. In addition to a salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Design a `ShiftSupervisor` class that inherits from the `Employee` class you created in Programming Challenge 1. The `ShiftSupervisor` class should have a field that holds the annual salary, and a field that holds the annual production bonus that a shift supervisor has earned. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

3. **`TeamLeader` Class**

   In a particular factory, a team leader is an hourly paid production worker who leads a small team. In addition to hourly pay, team leaders earn a fixed monthly bonus. Team leaders are required to attend a minimum number of hours of training per year. Design a `TeamLeader` class that inherits from the `ProductionWorker` class you designed in Programming Challenge 1. The `TeamLeader` class should have fields for the monthly bonus amount, the required number of training hours, and the number of training hours that the team leader has attended. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the class by writing a program that uses a `TeamLeader` object.

4. **Essay Class**
   Design an `Essay` class that inherits from the `GradedActivity` class presented in this chapter. The `Essay` class should determine the grade a student receives on an essay. The student's essay score can be up to 100 and is determined in the following manner:

```
 Grammar: 30 points
Spelling: 20 points
Correct length: 20 points
Content: 30 points
Demonstrate the class in a simple program.
```

5. **Course Grades**
   In a course, a teacher gives the following tests and assignments:
   - A **lab activity** that is observed by the teacher and assigned a numeric score.

- A **pass/fail exam** that has 10 questions. The minimum passing score is 70.
- An **essay** that is assigned a numeric score.
- A **final exam** that has 50 questions.

Write a class named `CourseGrades`. The class should have an array of `GradedActivity` objects as a field. The array should be named `grades`. The `grades` array should have four elements, one for each of the assignments previously described. The class should have the following methods:

Demonstrate the class in a program.

6. **Analyzable Interface**

   Modify the `CourseGrades` class you created in Programming Challenge 5 so that it implements the following interface:

   ```
   public interface Analyzable
   {
     double getAverage();
     GradedActivity getHighest();
     GradedActivity getLowest();
   }
   ```

   The `getAverage` method should return the average of the numeric scores stored in the `grades` array. The `getHighest` method should return a reference to the element of the grades array that has the highest numeric score. The `getLowest` method should return a reference to the element of the grades array that has the lowest numeric score. Demonstrate the new methods in a complete program.

7. **`Person` and `Customer` Classes**

   Design a class named `Person` with fields for holding a person’s name, address, and telephone number. Write one or more constructors and the appropriate mutator and accessor methods for the class’s fields.

   Next, design a class named `Customer`, which inherits from the `Person` class. The `Customer` class should have a field for a customer number and a `boolean` field indicating whether the customer wishes to be on a mailing list. Write one or more constructors, and the appropriate mutator and accessor methods, for the class's fields. Demonstrate an object of the `Customer` class in a simple program.

8. **`PreferredCustomer` Class**

   A retail store has a preferred customer plan where customers can earn discounts on all their purchases. The amount of a customer's discount is determined by the amount of the customer's cumulative purchases in the store, as follows:

   - When a preferred customer spends $500, he or she gets a 5 percent discount on all future purchases.
   - When a preferred customer spends $1,000, he or she gets a 6 percent discount on all future purchases.
   - When a preferred customer spends $1,500, he or she gets a 7 percent discount on all future purchases.
   - When a preferred customer spends $2,000 or more, he or she gets a 10 percent discount on all future purchases.

   Design a class named `PreferredCustomer`, which inherits from the `Customer` class you created in Programming Challenge 7. The `PreferredCustomer` class should have fields for the amount of the customer's purchases and the customer's discount level. Write one or more constructors and the appropriate mutator and accessor methods for the class's fields. Demonstrate the class in a simple program.

9. **`BankAccount` and `SavingsAccount` Classes**

Design an abstract class named `BankAccount` to hold the following data for a bank account:

- Balance
- Number of deposits this month
- Number of withdrawals
- Annual interest rate
- Monthly service charges

The class should have the following methods:

Next, design a `SavingsAccount` class that extends the `BankAccount` class. The `SavingsAccount` class should have a status field to represent an active or inactive account. If the balance of a savings account falls below $25, it becomes inactive. (The `status` field could be a `boolean` variable.) No more withdrawals can be made until the balance is raised above $25, at which time the account becomes active again. The savings account class should have the following methods:

10. **Ship, CruiseShip, and CargoShip Classes**

    Design a `Ship` class with the following members:

    - A field for the name of the ship (a string)
    - A field for the year that the ship was built (a string)
    - A constructor and appropriate accessors and mutators
    - A `toString` method that displays the ship's name and the year it was built

    Design a `CruiseShip` class that extends the `Ship` class. The `CruiseShip` class should have the following members:

    - A field for the maximum number of passengers (an `int`)
    - A constructor and appropriate accessors and mutators
    - A `toString` method that overrides the `toString` method in the base class. The `CruiseShip` class's `toString` method should display only the ship's name and the maximum number of passengers.

    Design a `CargoShip` class that extends the `Ship` class. The `CargoShip` class should have the following members:

    - A field for the cargo capacity in tonnage (an `int`)
    - A constructor and appropriate accessors and mutators
    - A `toString` method that overrides the `toString` method in the base class. The `CargoShip` class's `toString` method should display only the ship's name and the ship's cargo capacity.

Demonstrate the classes in a program that has a `Ship` array. Assign various `Ship`, `CruiseShip`, and `CargoShip` objects to the array elements. The program should then step through the array, calling each object’s `toString` method. (See **Code Listing 9-25** as an example.)