INHERITANCE AND POLYMORPHISM

Objectives

■ To develop a subclass from a superclass through inheritance (§11.2).
■ To invoke the superclass’s constructors and methods using the super keyword (§11.3).
■ To override instance methods in the subclass (§11.4).
■ To distinguish differences between overriding and overloading (§11.5).
■ To explore the toString() method in the Object class (§11.6).
■ To discover polymorphism and dynamic binding (§§11.7–11.8).
■ To describe casting and explain why explicit downcasting is necessary (§11.9).
■ To explore the equals method in the Object class (§11.10).
■ To store, retrieve, and manipulate objects in an ArrayList (§11.11).
■ To implement a Stack class using ArrayList (§11.12).
■ To enable data and methods in a superclass accessible in subclasses using the protected visibility modifier (§11.13).
■ To prevent class extending and method overriding using the final modifier (§11.14).
Chapter 11  Inheritance and Polymorphism

11.1 Introduction

Object-oriented programming allows you to derive new classes from existing classes. This is called inheritance. Inheritance is an important and powerful feature in Java for reusing software. Suppose you are to define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so to avoid redundancy and make the system easy to comprehend and easy to maintain? The answer is to use inheritance.

11.2 Superclasses and Subclasses

You use a class to model objects of the same type. Different classes may have some common properties and behaviors, which can be generalized in a class that can be shared by other classes. Inheritance enables you to define a general class and later extend it to more specialized classes. The specialized classes inherit the properties and methods from the general class.

Consider geometric objects. Suppose you want to design the classes to model geometric objects such as circles and rectangles. Geometric objects have many common properties and behaviors. They can be drawn in a certain color, filled or unfilled. Thus a general class

<table>
<thead>
<tr>
<th>GeometricObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>- color: String</td>
</tr>
<tr>
<td>- filled: boolean</td>
</tr>
<tr>
<td>- dateCreated: java.util.Date</td>
</tr>
</tbody>
</table>

+ GeometricObject() |
+ GeometricObject(color: String, filled: boolean) |
+ getColor(): String |
+ setColor(color: String): void |
+ isFilled(): boolean |
+ setFilled(filled: boolean): void |
+ getDateCreated(): java.util.Date |
+ toString(): String |

The color of the object (default: white). Indicates whether the object is filled with a color (default: false). The date when the object was created. Creates a GeometricObject. Creates a GeometricObject with the specified color and filled values. Returns the color. Sets a new color. Returns the filled property. Sets a new filled property. Returns the dateCreated. Returns a string representation of this object.

<table>
<thead>
<tr>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>- radius: double</td>
</tr>
<tr>
<td>+ Circle()</td>
</tr>
<tr>
<td>+ Circle(radius: double)</td>
</tr>
<tr>
<td>+ Circle(radius: double, color: String, filled: boolean)</td>
</tr>
<tr>
<td>+ getRadius(): double</td>
</tr>
<tr>
<td>+ setRadius(radius: double): void</td>
</tr>
<tr>
<td>+ getArea(): double</td>
</tr>
<tr>
<td>+ getPerimeter(): double</td>
</tr>
<tr>
<td>+ printCircle(): void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>- width: double</td>
</tr>
<tr>
<td>- height: double</td>
</tr>
<tr>
<td>+ Rectangle()</td>
</tr>
<tr>
<td>+ Rectangle(width: double, height: double)</td>
</tr>
<tr>
<td>+ getArea(): double</td>
</tr>
<tr>
<td>+ getWidth(): double</td>
</tr>
<tr>
<td>+ setWidth(width: double): void</td>
</tr>
<tr>
<td>+ getHeight(): double</td>
</tr>
<tr>
<td>+ setHeight(height: double): void</td>
</tr>
</tbody>
</table>

FIGURE 11.1 The GeometricObject class is the superclass for Circle and Rectangle.
11.2 Superclasses and Subclasses

GeometricObject can be used to model all geometric objects. This class contains the properties color and filled and their appropriate get and set methods. Assume that this class also contains the dateCreated property and the getDateCreated() and toString() methods. The toString() method returns a string representation for the object. Since a circle is a special type of geometric object, it shares common properties and methods with other geometric objects. Thus it makes sense to define the Circle class that extends the GeometricObject class. Likewise, Rectangle can also be declared as a subclass of GeometricObject. Figure 11.1 shows the relationship among these classes. An arrow pointing to the superclass is used to denote the inheritance relationship between the two classes involved.

In Java terminology, a class C1 extended from another class C2 is called a subclass, and C2 is called a superclass. A superclass is also referred to as a parent class, or a base class, and a subclass as a child class, an extended class, or a derived class. A subclass inherits accessible data fields and methods from its superclass and may also add new data fields and methods.

The Circle class inherits all accessible data fields and methods from the GeometricObject class. In addition, it has a new data field, radius, and its associated get and set methods. It also contains the getArea(), getPerimeter(), and getDiameter() methods for returning the area, perimeter, and diameter of the circle.

The Rectangle class inherits all accessible data fields and methods from the GeometricObject class. In addition, it has the data fields width and height and the associated get and set methods. It also contains the getArea() and getPerimeter() methods for returning the area and perimeter of the rectangle.

The GeometricObject, Circle, and Rectangle classes are shown in Listings 11.1, 11.2, and 11.3.

**Note**
To avoid naming conflict with the improved GeometricObject, Circle, and Rectangle classes introduced in the next chapter, name these classes GeometricObject1, Circle4, and Rectangle1 in this chapter. For convenience, we will still refer to them in the text as GeometricObject, Circle, and Rectangle classes. The best way to avoid naming conflict would be to place these classes in a different package. However, for simplicity and consistency, all classes in this book are placed in the default package.

**LISTING 11.1 GeometricObject1.java**

```java
public class GeometricObject1 {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    /** Construct a default geometric object */
    public GeometricObject1() {
        dateCreated = new java.util.Date();
    }

    /** Construct a geometric object with the specified color * and filled value */
    public GeometricObject1(String Color, boolean filled) {
        this.color = Color;
        this.filled = filled;
    }

    /** Return color */
    public String getColor() {
        return color;
    }
}
```
/** Set a new color */
public void setColor(String color) {
    this.color = color;
}

/** Return filled. Since filled is boolean, 
its get method is named isFilled */
public boolean isFilled() {
    return filled;
}

/** Set a new filled */
public void setFilled(boolean filled) {
    this.filled = filled;
}

/** Get dateCreated */
public java.util.Date getDateCreated() {
    return dateCreated;
}

/** Return a string representation of this object */
public String toString() {
    return "created on " + dateCreated + 
"color: " + color + 
" and filled: " + filled;
}

LISTING 11.2 Circle4.java

public class Circle4 extends GeometricObject1 {
    private double radius;

    public Circle4() {
    }

    public Circle4(double radius) {
        this.radius = radius;
    }

    public Circle4(double radius, String color, boolean filled) {
        this.radius = radius;
        setColor(color);
        setFilled(filled);
    }

    /** Return radius */
    public double getRadius() {
        return radius;
    }

    /** Set a new radius */
    public void setRadius(double radius) {
        this.radius = radius;
    }

    /** Return area */
    public double getArea() {
        return radius * radius * Math.PI;
    }
11.2 Superclasses and Subclasses

The Circle class extends the GeometricObject class (Listing 11.2) using the following syntax:

```
public class Circle extends GeometricObject
```

The keyword `extends` (line 1) tells the compiler that the Circle class extends the GeometricObject class, thus inheriting the methods `getColor`, `setColor`, `isFilled`, `setFilled`, and `toString`. The overloaded constructor `Circle(double radius, string color, boolean filled)` is implemented by invoking the `setColor` and `setFilled` methods to set the color and filled properties (lines 11–15). These two public methods are defined in the base class `GeometricObject` and are inherited in `Circle`. So, they can be used in the derived class.

You might attempt to use the data fields `color` and `filled` directly in the constructor as follows:

```
public Circle4(double radius, String color, boolean filled) {
    this.radius = radius;
    this.color = color; // Illegal
    this.filled = filled; // Illegal
}
```

This is wrong, because the private data fields `color` and `filled` in the GeometricObject class cannot be accessed in any class other than in the GeometricObject class itself. The only way to read and modify `color` and `filled` is through their `get` and `set` methods.

The Rectangle class (Listing 11.3) extends the GeometricObject class (Listing 11.2) using the following syntax:

```
public class Rectangle extends GeometricObject
```
Chapter 11  Inheritance and Polymorphism

The keyword extends (line 1) tells the compiler that the Rectangle class extends the GeometricObject class, thus inheriting the methods getColor, setColor, isFilled, setFilled, and toString.

**Listing 11.3  Rectangle1.java**

```java
public class Rectangle1 extends GeometricObject1 {
    private double width;
    private double height;

    public Rectangle1() {
    }

    public Rectangle1(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public Rectangle1(double width, double height, String color, boolean filled) {
        this.width = width;
        this.height = height;
        setColor(color);
        setFilled(filled);
    }

    /** Return width */
    public double getWidth() {
        return width;
    }

    /** Set a new width */
    public void setWidth(double width) {
        this.width = width;
    }

    /** Return height */
    public double getHeight() {
        return height;
    }

    /** Set a new height */
    public void setHeight(double height) {
        this.height = height;
    }

    /** Return area */
    public double getArea() {
        return width * height;
    }

    /** Return perimeter */
    public double getPerimeter() {
        return 2 * (width + height);
    }
}
```

The code in Listing 11.4 creates objects of Circle and Rectangle and invokes the methods on these objects. The toString() method is inherited from the GeometricObject class and is invoked from a Circle object (line 4) and a Rectangle object (line 10).
LISTING 11.4 TestCircleRectangle.java

```java
public class TestCircleRectangle {
    public static void main(String[] args) {
        Circle4 circle = new Circle4(1);
        System.out.println("A circle " + circle.toString());
        System.out.println("The radius is " + circle.getRadius());
        System.out.println("The area is " + circle.getArea());
        System.out.println("The diameter is " + circle.getDiameter());

        Rectangle1 rectangle = new Rectangle1(2, 4);
        System.out.println("A rectangle " + rectangle.toString());
        System.out.println("The area is " + rectangle.getArea());
        System.out.println("The perimeter is " + rectangle.getPerimeter());
    }
}
```

A circle created on Thu Sep 24 20:31:02 EDT 2009
 color: white and filled: false
 The radius is 1.0
 The area is 3.141592653589793
 The diameter is 2.0

A rectangle created on Thu Sep 24 20:31:02 EDT 2009
 color: white and filled: false
 The area is 8.0
 The perimeter is 12.0

The following points regarding inheritance are worthwhile to note:

- Contrary to the conventional interpretation, a subclass is not a subset of its superclass. In fact, a subclass usually contains more information and methods than its superclass.

- Private data fields in a superclass are not accessible outside the class. Therefore, they cannot be used directly in a subclass. They can, however, be accessed/mutated through public accessor/mutator if defined in the superclass.

- Not all is-a relationships should be modeled using inheritance. For example, a square is a rectangle, but you should not define a Square class to extend a Rectangle class, because there is nothing to extend (or supplement) from a rectangle to a square. Rather you should define a Square class to extend the GeometricObject class. For class A to extend class B, A should contain more detailed information than B.

- Inheritance is used to model the is-a relationship. Do not blindly extend a class just for the sake of reusing methods. For example, it makes no sense for a Tree class to extend a Person class, even though they share common properties such as height and weight. A subclass and its superclass must have the is-a relationship.

- Some programming languages allow you to derive a subclass from several classes. This capability is known as multiple inheritance. Java, however, does not allow multiple inheritance. A Java class may inherit directly from only one superclass. This restriction is known as single inheritance. If you use the extends keyword to define a subclass, it allows only one parent class. Nevertheless, multiple inheritance can be achieved through interfaces, which will be introduced in §14.4, “Interfaces.”
11.3 Using the super Keyword

A subclass inherits accessible data fields and methods from its superclass. Does it inherit constructors? Can superclass constructors be invoked from subclasses? This section addresses these questions and their ramifications.

§10.4, “The this Reference,” introduced the use of the keyword this to reference the calling object. The keyword super refers to the superclass of the class in which super appears. It can be used in two ways:

- To call a superclass constructor.
- To call a superclass method.

11.3.1 Calling Superclass Constructors

The syntax to call a superclass constructor is:

super(), or super(parameters);

The statement super() invokes the no-arg constructor of its superclass, and the statement super(arguments) invokes the superclass constructor that matches the arguments. The statement super() or super(arguments) must appear in the first line of the subclass constructor; this is the only way to explicitly invoke a superclass constructor. For example, the constructor in lines 11–15 in Listing 11.2 can be replaced by the following code:

```java
public Circle4(double radius, String color, boolean filled) {
    super(color, filled);
    this.radius = radius;
}
```

Caution

You must use the keyword super to call the superclass constructor, and the call must be the first statement in the constructor. Invoking a superclass constructor’s name in a subclass causes a syntax error.

Note

A constructor is used to construct an instance of a class. Unlike properties and methods, the constructors of a superclass are not inherited in the subclass. They can only be invoked from the constructors of the subclasses, using the keyword super.

11.3.2 Constructor Chaining

A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically puts super() as the first statement in the constructor. For example,

```java
public ClassName() {
    // some statements
}

public ClassName(double d) {
    // some statements
}
```

Equivalent

```java
public ClassName() {
    super();
    // some statements
}

public ClassName(double d) {
    super();
    // some statements
}
```
In any case, constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain. When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor before performing its own tasks. If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its own tasks. This process continues until the last constructor along the inheritance hierarchy is called. This is constructor chaining.

Consider the following code:

```java
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }
    public Faculty() {
        System.out.println("(4) Performs Faculty's tasks");
    }
}
class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Performs Employee's tasks");
    }
    public Employee(String s) {
        System.out.println(s);
    }
}
class Person {
    public Person() {
        System.out.println("(1) Performs Person's tasks");
    }
}
```

(1) Performs Person's tasks
(2) Invoke Employee's overloaded constructor
(3) Performs Employee's tasks
(4) Performs Faculty's tasks

The program produces the preceding output. Why? Let us discuss the reason. In line 3, `new Faculty()` invokes `Faculty`'s no-arg constructor. Since `Faculty` is a subclass of `Employee`, `Employee`'s no-arg constructor is invoked before any statements in `Faculty`'s constructor are executed. `Employee`'s no-arg constructor invokes `Employee`'s second constructor (line 12). Since `Employee` is a subclass of `Person`, `Person`'s no-arg constructor is invoked before any statements in `Employee`'s second constructor are executed. This process is pictured in the figure below.
Chapter 11  Inheritance and Polymorphism

Caution
If a class is designed to be extended, it is better to provide a no-arg constructor to avoid programming errors. Consider the following code:

```java
1 public class Apple extends Fruit { 
2 } 
3 
4 class Fruit { 
5   public Fruit(String name) { 
6     System.out.println("Fruit's constructor is invoked"); 
7   } 
8 }
```

Since no constructor is explicitly defined in `Apple`, `Apple`'s default no-arg constructor is defined implicitly. Since `Apple` is a subclass of `Fruit`, `Apple`'s default constructor automatically invokes `Fruit`'s no-arg constructor. However, `Fruit` does not have a no-arg constructor, because `Fruit` has an explicit constructor defined. Therefore, the program cannot be compiled.

Design Guide
It is better to provide a no-arg constructor (if desirable) for every class to make the class easy to extend and to avoid errors.

11.3.3 Calling Superclass Methods
The keyword `super` can also be used to reference a method other than the constructor in the superclass. The syntax is like this:

```
super.method(parameters);
```

You could rewrite the `printCircle()` method in the `Circle` class as follows:

```java
public void printCircle() {
    System.out.println("The circle is created " +
                     super.getDateCreated() + " and the radius is " + radius);
}
```

It is not necessary to put `super` before `getDateCreated()` in this case, however, because `getDateCreated` is a method in the `GeometricObject` class and is inherited by the `Circle` class. Nevertheless, in some cases, as shown in the next section, the keyword `super` is needed.

11.4 Overriding Methods
A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as method overriding.

The `toString` method in the `GeometricObject` class returns the string representation for a geometric object. This method can be overridden to return the string representation for a circle. To override it, add the following new method in Listing 11.2, Circle4.java:

```java
1 public class Circle4 extends GeometricObject1 {
2   // Other methods are omitted
3 
4   /** Override the toString method defined in GeometricObject */
5   public String toString() {
6     return super.toString() + "\nradius is " + radius;
7   }
8 }
```
The `toString()` method is defined in the `GeometricObject` class and modified in the `Circle` class. Both methods can be used in the `Circle` class. To invoke the `toString` method defined in the `GeometricObject` class from the `Circle` class, use `super.toString()` (line 6).

Can a subclass of `Circle` access the `toString` method defined in the `GeometricObject` class using a syntax such as `super.super.toString()`? No. This is a syntax error.

Several points are worth noting:

- An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

- Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden. The hidden static methods can be invoked using the syntax `SuperClassName.staticMethodName`.

### 11.5 Overriding vs. Overloading

You have learned about overloading methods in §5.8. Overloading means to define multiple methods with the same name but different signatures. Overriding means to provide a new implementation for a method in the subclass. The method is already defined in the superclass.

To override a method, the method must be defined in the subclass using the same signature and the same return type.

Let us use an example to show the differences between overriding and overloading. In (a) below, the method `p(double i)` in class `A` overrides the same method defined in class `B`. In (b), however, the class `B` has two overloaded methods `p(double i)` and `p(int i)`. The method `p(double i)` is inherited from `B`.

```java
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    // This method overrides the method in B
    public void p(double i) {
        System.out.println(i);
    }
}
```

(a)

```java
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    // This method overrides the method in B
    public void p(double i) {
        System.out.println(i);
    }
    public void p(int i) {
        System.out.println(i);
    }
}
```

(b)

When you run the `Test` class in (a), both `a.p(10)` and `a.p(10.0)` invoke the `p(double i)` method defined in class `A` to display `10.0`. When you run the `Test` class in (b), `a.p(10)` invokes the `p(int i)` method defined in class `B` to display `20`, and `a.p(10.0)` invokes the `p(double i)` method defined in class `A` to display `10.0`.  
Chapter 11  Inheritance and Polymorphism

11.6 The Object Class and Its toString() Method

Every class in Java is descended from the java.lang.Object class. If no inheritance is specified when a class is defined, the superclass of the class is Object by default. For example, the following two class definitions are the same:

```
public class ClassName {
    ...
}
```

```
public class ClassName extends Object {
    ...
}
```

Classes such as String, StringBuilder, Loan, and GeometricObject are implicitly subclasses of Object (as are all the main classes you have seen in this book so far). It is important to be familiar with the methods provided by the Object class so that you can use them in your classes. We will introduce the toString method in the Object class in this section.

The signature of the toString() method is

```
public String toString()
```

Invoking toString() on an object returns a string that describes the object. By default, it returns a string consisting of a class name of which the object is an instance, an at sign (@), and the object’s memory address in hexadecimal. For example, consider the following code for the Loan class defined in Listing 10.2:

```
Loan loan = new Loan();
System.out.println(loan.toString());
```

The code displays something like Loan@15037e5. This message is not very helpful or informative. Usually you should override the toString method so that it returns a descriptive string representation of the object. For example, the toString method in the Object class was overridden in the GeometricObject class in lines 46-49 in Listing 11.1 as follows:

```
public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color + "
    and filled: " + filled;
}
```

Note

You can also pass an object to invoke System.out.println(object) or System.out.print(object). This is equivalent to invoking System.out.println(object.toString()) or System.out.print(object.toString()). So you could replace System.out.println(loan.toString()) with System.out.println(loan).

11.7 Polymorphism

The three pillars of object-oriented programming are encapsulation, inheritance, and polymorphism. You have already learned the first two. This section introduces polymorphism.

First let us define two useful terms: subtype and supertype. A class defines a type. A type defined by a subclass is called a subtype and a type defined by its superclass is called a supertype. So, you can say that Circle is a subtype of GeometricObject and GeometricObject is a supertype for Circle.
The inheritance relationship enables a subclass to inherit features from its superclass with additional new features. A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa. For example, every circle is a geometric object, but not every geometric object is a circle. Therefore, you can always pass an instance of a subclass to a parameter of its superclass type. Consider the code in Listing 11.5.

LISTING 11.5 PolymorphismDemo.java

```java
public class PolymorphismDemo {
    public static void main(String[] args) {
        // Display circle and rectangle properties
        displayObject(new Circle4(1, "red", false));
        displayObject(new Rectangle1(1, 1, "black", false));
    }

    /** Display geometric object properties */
    public static void displayObject(GeometricObject1 object) {
        System.out.println("Created on " + object.getDateCreated() + ". Color is " + object.getColor());
    }
}
```

Polymorphism means that a variable of a supertype can refer to a subtype object. In simple terms, polymorphism means that a variable of a supertype can refer to a subtype object.

11.8 Dynamic Binding

A method may be defined in a superclass and overridden in its subclass. For example, the `toString()` method is defined in the `Object` class and overridden in `GeometricObject1`. Consider the following code:

```java
Object o = new GeometricObject();
System.out.println(o.toString());
```

Which `toString()` method is invoked by `o`? To answer this question, we first introduce two terms: declared type and actual type. A variable must be declared a type. The type of a variable is called its declared type. Here `o`’s declared type is `Object`. A variable of a reference type can hold a null value or a reference to an instance of the declared type. The instance may be created using the constructor of the declared type or its subtype. The actual type of the variable is the actual class for the object referenced by the variable. Here `o`’s actual type is `GeometricObject`, since `o` references to an object created using `new GeometricObject()`. Which `toString()` method is invoked by `o` is determined by `o`’s actual type. This is known as dynamic binding.

Dynamic binding works as follows: Suppose an object `o` is an instance of classes `C_1`, `C_2`, ..., `C_{n-1}`, and `C_n`, where `C_1` is a subclass of `C_2`, `C_2` is a subclass of `C_3`, ..., and `C_{n-1}` is a subclass of `C_n`, as shown in Figure 11.2. That is, `C_n` is the most general class, and `C_1` is the most specific...
class. In Java, $C_n$ is the `Object` class. If $o$ invokes a method $p$, the JVM searches the implementation for the method $p$ in $C_1$, $C_2$, ..., $C_{n-1}$, and $C_n$, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.

Listing 11.6 gives an example to demonstrate dynamic binding.

```java
public class DynamicBindingDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }
    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

Method $m$ (line 9) takes a parameter of the `Object` type. You can invoke $m$ with any object (e.g., `new GraduateStudent()`, `new Student()`, `new Person()`, and `new Object()`) in lines 3–6).

When the method $m($Object $x)$ is executed, the argument $x$'s $toString$ method is invoked. $x$ may be an instance of `GraduateStudent`, `Student`, `Person`, or `Object`. Classes `GraduateStudent`, `Student`, `Person`, and `Object` have their own implementations of the $toString$ method. Which implementation is used will be determined by $x$'s actual type at runtime. Invoking $m($new `GraduateStudent()$)$ (line 3) causes the $toString$ method defined in the `Student` class to be invoked.
11.9 Casting Objects and the `instanceof` Operator

Invoking `m(new Student())` (line 4) causes the `toString` method defined in the `Student` class to be invoked.

Invoking `m(new Person())` (line 5) causes the `toString` method defined in the `Person` class to be invoked. Invoking `m(new Object())` (line 6) causes the `toString` method defined in the `Object` class to be invoked.

Matching a method signature and binding a method implementation are two separate issues. The `declared type` of the reference variable decides which method to match at compile time. The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compile time. A method may be implemented in several subclasses. The JVM dynamically binds the implementation of the method at runtime, decided by the actual type of the variable.

## 11.9 Casting Objects and the `instanceof` Operator

You have already used the casting operator to convert variables of one primitive type to another. Casting can also be used to convert an object of one class type to another within an inheritance hierarchy. In the preceding section, the statement

```java
m(new Student());
```

assigns the object `new Student()` to a parameter of the `Object` type. This statement is equivalent to

```java
Object o = new Student(); // Implicit casting
m(o);
```

The statement `Object o = new Student()`, known as `implicit casting`, is legal because an instance of `Student` is automatically an instance of `Object`.

Suppose you want to assign the object reference `o` to a variable of the `Student` type using the following statement:

```java
Student b = o;
```

In this case a compile error would occur. Why does the statement `Object o = new Student()` work but `Student b = o` doesn’t? The reason is that a `Student` object is always an instance of `Object`, but an `Object` is not necessarily an instance of `Student`. Even though you can see that `o` is really a `Student` object, the compiler is not clever enough to know it. To tell the compiler that `o` is a `Student` object, use an `explicit casting`. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```java
Student b = (Student)o; // Explicit casting
```

It is always possible to cast an instance of a subclass to a variable of a superclass (known as `upcasting`), because an instance of a subclass is `always` an instance of its superclass. When casting an instance of a superclass to a variable of its subclass (known as `downcasting`), explicit casting must be used to confirm your intention to the compiler with the `(SubclassName)` cast notation. For the casting to be successful, you must make sure that the object to be cast is an instance of the subclass. If the superclass object is not an instance of the subclass, a runtime `ClassCastException` occurs. For example, if an object is not an instance of `Student`, it cannot be cast into a variable of `Student`. It is a good practice, therefore, to ensure that the object is an instance of another object before attempting a casting. This can be accomplished by using the `instanceof` operator. Consider the following code:

```java
Object myObject = new Circle();
...
// Some lines of code
```
/** Perform casting if myObject is an instance of Circle */
if (myObject instanceof Circle) {
    System.out.println("The circle diameter is " + ((Circle)myObject).getDiameter());
    ...
}

You may be wondering why casting is necessary. Variable myObject is declared Object. The declared type decides which method to match at compile time. Using myObject.getDiameter() would cause a compile error, because the Object class does not have the getDiameter method. The compiler cannot find a match for myObject.getDiameter(). It is necessary to cast myObject into the Circle type to tell the compiler that myObject is also an instance of Circle.

Why not define myObject as a Circle type in the first place? To enable generic programming, it is a good practice to define a variable with a supertype, which can accept a value of any subtype.

Note
instanceof is a Java keyword. Every letter in a Java keyword is in lowercase.

Tip
To help understand casting, you may also consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for Apple and Orange. An apple is a fruit, so you can always safely assign an instance of Apple to a variable for Fruit. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.

Listing 11.7 demonstrates polymorphism and casting. The program creates two objects (lines 5–6), a circle and a rectangle, and invokes the displayObject method to display them (lines 9–10). The displayObject method displays the area and diameter if the object is a circle (line 15), and the area if the object is a rectangle (line 21).

Listing 11.7 CastingDemo.java

```java
public class CastingDemo {
    /** Main method */
    public static void main(String[] args) {
        // Create and initialize two objects
        Object object1 = new Circle4(1);
        Object object2 = new Rectangle1(1, 1);

        // Display circle and rectangle
        displayObject(object1);
        displayObject(object2);
    }

    /** A method for displaying an object */
    public static void displayObject(Object object) {
        if (object instanceof Circle4) {
            System.out.println("The circle area is " + ((Circle4)object).getArea());
            System.out.println("The circle diameter is " + ((Circle4)object).getDiameter());
        }
        else if (object instanceof Rectangle1) {
            System.out.println("The rectangle area is " + ((Rectangle1)object).getArea());
        }
    }
}
```
The displayObject(Object object) method is an example of generic programming. It can be invoked by passing any instance of Object.

The program uses implicit casting to assign a Circle object to object1 and a Rectangle object to object2 (lines 5–6), then invokes the displayObject method to display the information on these objects (lines 9–10).

In the displayObject method (lines 14–25), explicit casting is used to cast the object to Circle if the object is an instance of Circle, and the methods getArea and getDiameter are used to display the area and diameter of the circle.

Casting can be done only when the source object is an instance of the target class. The program uses the instanceof operator to ensure that the source object is an instance of the target class before performing a casting (line 15).

Explicit casting to Circle (lines 17, 19) and to Rectangle (line 23) is necessary because the getArea and getDiameter methods are not available in the Object class.

Caution
The object member access operator (.) precedes the casting operator. Use parentheses to ensure that casting is done before the . operator, as in

```
((Circle)object).getArea();
```

### 11.10 The Object’s equals Method

Another method defined in the Object class that is often used is the equals method. Its signature is

```
public boolean equals(Object o)
```

This method tests whether two objects are equal. The syntax for invoking it is:

```
object1.equals(object2);
```

The default implementation of the equals method in the Object class is:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

This implementation checks whether two reference variables point to the same object using the == operator. You should override this method in your custom class to test whether two distinct objects have the same content.

You have already used the equals method to compare two strings in §9.2, “The String Class.” The equals method in the String class is inherited from the Object class and is overridden in the String class to test whether two strings are identical in content. You can override the equals method in the Circle class to compare whether two circles are equal based on their radius as follows:

```
public boolean equals(Object o) {
    if (o instanceof Circle) {
        return radius == ((Circle)o).radius;
    }
```

The circle area is 3.141592653589793
The circle diameter is 2.0
The rectangle area is 1.0


390  Chapter 11  Inheritance and Polymorphism

```java
else
    return false;
}
```

**Note**

The `==` comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The `equals` method is intended to test whether two objects have the same contents, provided that the method is overridden in the defining class of the objects. The `==` operator is stronger than the `equals` method, in that the `==` operator checks whether the two reference variables refer to the same object.

**Caution**

Using the signature `equals(SomeClassName obj)` (e.g., `equals(Circle c)`) to override the `equals` method in a subclass is a common mistake. You should use `equals(Object obj)`. See Review Question 11.15.

11.11 The `ArrayList` Class

Now we are ready to introduce a very useful class for storing objects. You can create an array to store objects. But, once the array is created, its size is fixed. Java provides the `ArrayList` class that can be used to store an unlimited number of objects. Figure 11.3 shows some methods in `ArrayList`.

![Video Note](https://example.com/video-note)

the `ArrayList` class

```java
java.util.ArrayList
+ArrayList()
+add(o: Object): void
+add(index: int, o: Object): void
+clear(): void
+contains(o: Object): boolean
+get(index: int): Object
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+size(): int
+remove(index: int): boolean
+set(index: int, o: Object): Object
```

**Figure 11.3** An `ArrayList` stores an unlimited number of objects.

Listing 11.8 gives an example of using `ArrayList` to store objects.

**LISTING 11.8 TestArrayList.java**

```java
public class TestArrayList {
    public static void main(String[] args) {
        // Create a list to store cities
        java.util.ArrayList cityList = new java.util.ArrayList();
        // Add some cities in the list
        cityList.add("London");
    }
}
```
11.11 The ArrayList Class 391

```java
8     // cityList now contains [London]
9     cityList.add("Denver");
10    // cityList now contains [London, Denver]
11    cityList.add("Paris");
12    // cityList now contains [London, Denver, Paris]
13    cityList.add("Miami");
14    // cityList now contains [London, Denver, Paris, Miami]
15    cityList.add("Seoul");
17    cityList.add("Tokyo");
18
19    System.out.println("List size? " + cityList.size());
20    System.out.println("Is Miami in the list? " +
21           cityList.contains("Miami");)
22    System.out.println("The location of Denver in the list? "
23           + cityList.indexOf("Denver");)
24    System.out.println("Is the list empty? " +
25           cityList.isEmpty()); // Print false
26
27    // Insert a new city at index 2
28    cityList.add(2, "Xian");
30
31    // Remove a city from the list
32    cityList.remove("Miami");
34
35    // Remove a city at index 1
36    cityList.remove(1);
38
39    // Display the contents in the list
40    System.out.println(cityList.toString());
41
42    // Display the contents in the list in reverse order
43    for (int i = cityList.size() - 1; i >= 0; i--)
44       System.out.print(cityList.get(i) + " ");
45    System.out.println();
46
47    // Create a list to store two circles
48    java.util.ArrayList list = new java.util.ArrayList();
49
50    // Add two circles
51    list.add(new Circle4(2));
52    list.add(new Circle4(3));
53
54    // Display the area of the first circle in the list
55    System.out.println("The area of the circle? " +
56           ((Circle4)list.get(0)).getArea());
57  }
58  }
```

List size? 6
Is Miami in the list? true
The location of Denver in the list? 1
Is the list empty? false
[London, Xian, Paris, Seoul, Tokyo]
Tokyo Seoul Paris Xian London
The area of the circle? 12.566370614359172
Chapter 11  Inheritance and Polymorphism

The program creates an `ArrayList` using its no-arg constructor (line 4). The `add` method adds any instance of `Object` into the list. Since `String` is a subclass of `Object`, strings can be added to the list. The `add` method (lines 7–17) adds an object to the end of list. So, after `cityList.add("London")` (line 7), the list contains

```
[London]
```

After `cityList.add("Denver")` (line 9), the list contains

```
[London, Denver]
```

After adding Paris, Miami, Seoul, and Tokyo (lines 11–17), the list would contain

```
```

Invoking `size()` (line 20) returns the size of the list, which is currently 6. Invoking `contains("Miami")` (line 22) checks whether the object is in the list. In this case, it returns `true`, since `Miami` is in the list. Invoking `indexOf("Denver")` (line 24) returns the index of the object in the list, which is 1. If the object is not in the list, it returns -1. The `isEmpty()` method (line 26) checks whether the list is empty. It returns `false`, since the list is not empty.

The statement `cityList.add(2, "Xian")` (line 29) inserts an object to the list at the specified index. After this statement, the list becomes

```
[London, Denver, Xian, Paris, Miami, Seoul, Tokyo]
```

The statement `cityList.remove("Miami")` (line 33) removes the object from the list. After this statement, the list becomes

```
```

The statement in line 41 is same as

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

The `toString()` method returns a string representation for the list in the form of `\[e0.toString(),e1.toString(),...,ek.toString()\]`, where `e0,e1,...,and ek` are the elements in the list.

The `getIndex()` method (line 45) returns the object at the specified index.

Note

You will get the following warning when compiling this program from the command prompt:

```
Note: TestArrayList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

This warning can be eliminated using generic types discussed in Chapter 21, “Generics”. For now, ignore it. Despite the warning, the program will be compiled just fine to produce a .class file.

Array vs. `ArrayList`

`ArrayList` objects can be used like arrays, but there are many differences. Table 11.1 lists their similarities and differences.

Once an array is created, its size is fixed. You can access an array element using the square-bracket notation (e.g., `a[index]`). When an `ArrayList` is created, its size is 0. You cannot use the `get` and `set` methods if the element is not in the list. It is easy to add, insert,
11.12 A Custom Stack Class

**Table 11.1** Differences and Similarities between Arrays and ArrayList

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>ArrayList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating an array/ArrayList</td>
<td><code>Object[] a = new Object[10]</code></td>
<td><code>ArrayList list = new ArrayList();</code></td>
</tr>
<tr>
<td>Accessing an element</td>
<td><code>a[index]</code></td>
<td><code>list.get(index);</code></td>
</tr>
<tr>
<td>Updating an element</td>
<td><code>a[index] = &quot;London&quot;;</code></td>
<td><code>list.set(index, &quot;London&quot;);</code></td>
</tr>
<tr>
<td>Returning size</td>
<td><code>a.length</code></td>
<td><code>list.size();</code></td>
</tr>
<tr>
<td>Adding a new element</td>
<td><code>list.add(&quot;London&quot;);</code></td>
<td></td>
</tr>
<tr>
<td>Inserting a new element</td>
<td><code>list.add(index, &quot;London&quot;);</code></td>
<td></td>
</tr>
<tr>
<td>Removing an element</td>
<td><code>list.remove(index);</code></td>
<td></td>
</tr>
<tr>
<td>Removing an element</td>
<td><code>list.remove(Object);</code></td>
<td></td>
</tr>
<tr>
<td>Removing all elements</td>
<td><code>list.clear();</code></td>
<td></td>
</tr>
</tbody>
</table>

and remove elements in a list, but it is rather complex to add, insert, and remove elements in an array. You have to write code to manipulate the array in order to perform these operations.

**Note**

`java.util.Vector` is also a class for storing objects, which is very similar to the `ArrayList` class. All the methods in `ArrayList` are also available in `Vector`. The `Vector` class was introduced in JDK 1.1. The `ArrayList` class introduced in JDK 1.2 was intended to replace the `Vector` class.

## 11.12 A Custom Stack Class

"Designing a Class for Stacks" in §10.8 presented a stack class for storing `int` values. This section introduces a stack class to store objects. You can use an `ArrayList` to implement `Stack`, as shown in Listing 11.9. The UML diagram for the class is shown in Figure 11.4.

![MyStack class](image_url)

**Figure 11.4** The MyStack class encapsulates the stack storage and provides the operations for manipulating the stack.

**Listing 11.9** MyStack.java

```java
public class MyStack {
    private java.util.ArrayList list = new java.util.ArrayList();

    public boolean isEmpty() {
        return list.isEmpty();
    }
}
```

A list to store elements.

Returns true if this stack is empty.

Returns the number of elements in this stack.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns the position of the first element in the stack from the top that matches the specified element.
An array list is created to store the elements in the stack (line 2). The `isEmpty()` method (lines 4–6) returns `list.isEmpty()`. The `getSize()` method (lines 8–10) returns `list.size()`. The `peek()` method (lines 12–14) retrieves the element at the top of the stack without removing it. The end of the list is the top of the stack. The `pop()` method (lines 16–20) removes the top element from the stack and returns it. The `push(Object element)` method (lines 22–24) adds the specified element to the stack. The `search(Object element)` method checks whether the specified element is in the stack, and it returns the index of first-matching element in the stack from the top by invoking `list.lastIndexOf(o)`. The `toString()` method (lines 31–33) defined in the `Object` class is overridden to display the contents of the stack by invoking `list.toString()`. The `toString()` method implemented in `ArrayList` returns a string representation of all the elements in an array list.

### Design Guide

In Listing 11.9, `MyStack` contains `ArrayList`. The relationship between `MyStack` and `ArrayList` is composition. While inheritance models an is-a relationship, composition models a has-a relationship. You may also implement `MyStack` as a subclass of `ArrayList` (see Exercise 11.4). Using composition is better, however, because it enables you to define a completely new stack class without inheriting the unnecessary and inappropriate methods from `ArrayList`.

### 11.13 The protected Data and Methods

So far you have used the `private` and `public` keywords to specify whether data fields and methods can be accessed from the outside of the class. Private members can be accessed only from the inside of the class, and public members can be accessed from any other classes.

Often it is desirable to allow subclasses to access data fields or methods defined in the superclass, but not allow nonsubclasses to access these data fields and methods. To do so, you can use the `protected` keyword. A protected data field or method in a superclass can be accessed in its subclasses.
11.13 The **protected** Data and Methods 395

The modifiers **private**, **protected**, and **public** are known as **visibility** or **accessibility modifiers** because they specify how class and class members are accessed. The visibility of these modifiers increases in this order:

Visibility increases

private, none (if no modifier is used), protected, public

Table 11.2 summarizes the accessibility of the members in a class. Figure 11.5 illustrates how a public, protected, default, and private datum or method in class **C1** can be accessed from a class **C2** in the same package, from a subclass **C3** in the same package, from a subclass **C4** in a different package, and from a class **C5** in a different package.

<table>
<thead>
<tr>
<th>Modifier on members in a class</th>
<th>Accessed from the same class</th>
<th>Accessed from the same package</th>
<th>Accessed from a subclass</th>
<th>Accessed from a different package</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>(default)</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

**TABLE 11.2** Data and Methods Visibility

**FIGURE 11.5** Visibility modifiers are used to control how data and methods are accessed.

Use the **private** modifier to hide the members of the class completely so that they cannot be accessed directly from outside the class. Use no modifiers in order to allow the members of the class to be accessed directly from any class within the same package but not from other
packages. Use the protected modifier to enable the members of the class to be accessed by
the subclasses in any package or classes in the same package. Use the public modifier to enable the members of the class to be accessed by any class.

Your class can be used in two ways: for creating instances of the class, and for defining
subclasses by extending the class. Make the members private if they are not intended for
use from outside the class. Make the members public if they are intended for the users of the
class. Make the fields or methods protected if they are intended for the extenders of the
class but not the users of the class.

The private and protected modifiers can be used only for members of the class. The
public modifier and the default modifier (i.e., no modifier) can be used on members of the
class as well on the class. A class with no modifier (i.e., not a public class) is not accessible by
classes from other packages.

Note
A subclass may override a protected method in its superclass and change its visibility to public.
However, a subclass cannot weaken the accessibility of a method defined in the superclass. For
example, if a method is defined as public in the superclass, it must be defined as public in the
subclass.

11.14 Preventing Extending and Overriding
You may occasionally want to prevent classes from being extended. In such cases, use the
final modifier to indicate that a class is final and cannot be a parent class. The Math class is
a final class. The String, StringBuilder, and StringBuffer classes are also final
classes. For example, the following class is final and cannot be extended:

```java
public final class C {
    // Data fields, constructors, and methods omitted
}
```

You also can define a method to be final; a final method cannot be overridden by its sub-
classes.

For example, the following method is final and cannot be overridden:

```java
public class Test {
    // Data fields, constructors, and methods omitted

    public final void m() {
        // Do something
    }
}
```

Note
The modifiers are used on classes and class members (data and methods), except that the final
modifier can also be used on local variables in a method. A final local variable is a constant inside
a method.

**Key Terms**

- actual type 385
- array list 393
- casting objects 387
- composition 394
- constructor chaining 381
- declared type 385
- dynamic binding 385
- final 396
Chapter Summary

1. You can derive a new class from an existing class. This is known as class inheritance. The new class is called a subclass, child class or extended class. The existing class is called a superclass, parent class, or base class.

2. A constructor is used to construct an instance of a class. Unlike properties and methods, the constructors of a superclass are not inherited in the subclass. They can be invoked only from the constructors of the subclasses, using the keyword super.

3. A constructor may invoke an overloaded constructor or its superclass’s constructor. The call must be the first statement in the constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor, which invokes the superclass’s no-arg constructor.

4. To override a method, the method must be defined in the subclass using the same signature as in its superclass.

5. An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

6. Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.

7. Every class in Java is descended from the java.lang.Object class. If no inheritance is specified when a class is defined, its superclass is Object.

8. If a method’s parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter’s subclasses (e.g., Circle or String). When an object (e.g., a Circle object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically.

9. It is always possible to cast an instance of a subclass to a variable of a superclass, because an instance of a subclass is always an instance of its superclass. When casting an instance of a superclass to a variable of its subclass, explicit casting must be used to confirm your intention to the compiler with the (SubclassName) cast notation.

10. A class defines a type. A type defined by a subclass is called a subtype and a type defined by its superclass is called a supertype.

11. When invoking an instance method from a reference variable, the actual type of the variable decides which implementation of the method is used at runtime. When
accessing a field or a static method, the declared type of the reference variable decides which method is used at compile time.

12. You can use `obj instanceof AClass` to test whether an object is an instance of a class.

13. You can use the `protected` modifier to prevent the data and methods from being accessed by nonsubclasses from a different package.

14. You can use the `final` modifier to indicate that a class is final and cannot be a parent class and to indicate that a method is final and cannot be overridden.

**Review Questions**

Sections 11.2–11.5

11.1 What is the printout of running the class `C` in (a)? What problem arises in compiling the program in (b)?

```java
class A {
    public A() {
        System.out.println(
            "A's no-arg constructor is invoked");
    }
}
class B extends A {
}
public class C {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

(a)

```java
class A {
    public A(int x) {
    }
}
class B extends A {
    public B() {
    }
}
public class C {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

(b)

11.2 True or false?

1. A subclass is a subset of a superclass.
2. When invoking a constructor from a subclass, its superclass’s no-arg constructor is always invoked.
3. You can override a private method defined in a superclass.
4. You can override a static method defined in a superclass.

11.3 Identify the problems in the following classes:

```java
public class Circle {
    private double radius;
    public Circle(double radius) {
        radius = radius;
    }
    public double getRadius() {
        return radius;
    }
    public double getArea() {
        return radius * radius * Math.PI;
    }
}
```
15  
16 class B extends Circle {
17     private double length;
18 
19    B(double radius, double length) {
20        Circle(radius);
21        length = length;
22    }
23 }
24 
25 /** Override getArea() */
26    public double getArea() {
27        return getArea() * length;
28    }
29 }

11.4 How do you explicitly invoke a superclass’s constructor from a subclass?
11.5 How do you invoke an overridden superclass method from a subclass?
11.6 Explain the difference between method overloading and method overriding.
11.7 If a method in a subclass has the same signature as a method in its superclass with
    the same return type, is the method overridden or overloaded?
11.8 If a method in a subclass has the same signature as a method in its superclass with
    a different return type, will this be a problem?
11.9 If a method in a subclass has the same name as a method in its superclass with dif-
    ferent parameter types, is the method overridden or overloaded?

Sections 11.6–11.9
11.10 Does every class have a toString method and an equals method? Where do
    they come from? How are they used? Is it appropriate to override these methods?
11.11 Show the output of following program:

        1 public class Test {
        2        public static void main(String[] args) {
        3            A a = new A(3);
        4        }
        5    }
        6
        7 class A extends B {
        8        public A(int t) {
        9            System.out.println("A's constructor is invoked");
       10        }
       11    }
       12
       13 class B {
       14        public B() {
       15            System.out.println("B's constructor is invoked");
       16        }
       17    }

       Is the no-arg constructor of Object invoked when new A(3) is invoked?
11.12 For the GeometricObject and Circle classes in Listings 11.1 and 11.2, answer
    the following questions:
    (a) Are the following Boolean expressions true or false?

        Circle circle = new Circle(1);
        GeometricObject object1 = new GeometricObject();
Chapter 11  Inheritance and Polymorphism

(c) Can the following statements be compiled?

```java
Circle circle = new Circle(5);
GeometricObject object = circle;
```

(c) Can the following statements be compiled?

```java
GeometricObject object = new GeometricObject();
Circle circle = (Circle)object;
```

11.13 Suppose that Fruit, Apple, Orange, GoldenDelicious, and Macintosh are declared, as shown in Figure 11.6.

Assume that the following declaration is given:

```java
Fruit fruit = new GoldenDelicious();
Orange orange = new Orange();
```

Answer the following questions:

1. Is fruit instanceof Fruit?
2. Is fruit instanceof Orange?
3. Is fruit instanceof Apple?
4. Is fruit instanceof GoldenDelicious?
5. Is fruit instanceof Macintosh?
6. Is orange instanceof Orange?
7. Is orange instanceof Fruit?
8. Is orange instanceof Apple?
9. Suppose the method `makeAppleCider` is defined in the Apple class. Can fruit invoke this method? Can orange invoke this method?
10. Suppose the method `makeOrangeJuice` is defined in the Orange class. Can orange invoke this method? Can fruit invoke this method?
11. Is the statement Orange p = new Apple() legal?
12. Is the statement Macintosh p = new Apple() legal?
13. Is the statement Apple p = new Macintosh() legal?
11.14 What is wrong in the following code?

```
1 public class Test {
2     public static void main(String[] args) {
3         Object fruit = new Fruit();
4         Object apple = (Apple)fruit;
5     }
6 }
7
8 class Apple extends Fruit {
9 }
10
11 class Fruit {
12 }
```

Section 11.10

11.15 When overriding the `equals` method, a common mistake is mistyping its signature in the subclass. For example, the `equals` method is incorrectly written as `equals(Circle circle)`, as shown in (a) in the code below; instead, it should be `equals(Object circle)`, as shown in (b). Show the output of running class `Test` with the `Circle` class in (a) and in (b), respectively.

```
public class Test {
     public static void main(String[] args) {
         Object circle1 = new Circle();
         Object circle2 = new Circle();
         System.out.println(circle1.equals(circle2));
     }
}
```

```
class Circle {
     double radius;
     public boolean equals(Circle circle) {
         return this.radius == circle.radius;
     }
}
```

(a)

```
class Circle {
     double radius;
     public boolean equals(Object circle) {
         return this.radius == ((Circle)circle).radius;
     }
}
```

(b)

Sections 11.11–11.12

11.16 How do you create an `ArrayList`? How do you append an object to a list? How do you insert an object at the beginning of a list? How do you find the number of objects in a list? How do you remove a given object from a list? How do you remove the last object from the list? How do you check whether a given object is in a list? How do you retrieve an object at a specified index from a list?

11.17 There are three errors in the code below. Identify them.

```
ArrayList list = new ArrayList();
list.add("Denver");
list.add("Austin");
list.add(new java.util.Date());
String city = list.get(0);
list.set(3, "Dallas");
System.out.println(list.get(3));
```
Chapter 11  Inheritance and Polymorphism

Sections 11.13–11.14
11.18 What modifier should you use on a class so that a class in the same package can access it, but a class in a different package cannot access it?

11.19 What modifier should you use so that a class in a different package cannot access the class, but its subclasses in any package can access it?

11.20 In the code below, classes A and B are in the same package. If the question marks are replaced by blanks, can class B be compiled? If the question marks are replaced by private, can class B be compiled? If the question marks are replaced by protected, can class B be compiled?

```java
package p1;
public class A {
  int i;
  void m() {
    ...
  }
}
```

```java
package p1;
public class B extends A {
  public void m1(String[] args) {
    System.out.println(i);
    m();
  }
}
```

(a) (b)

11.21 In the code below, classes A and B are in different packages. If the question marks are replaced by blanks, can class B be compiled? If the question marks are replaced by private, can class B be compiled? If the question marks are replaced by protected, can class B be compiled?

```java
package p1;
public class A {
  int i;
  void m() {
    ...
  }
}
```

```java
package p2;
public class A {
  int i;
  void m() {
    ...
  }
}
```

(a) (b)

11.22 How do you prevent a class from being extended? How do you prevent a method from being overridden?

Comprehensive
11.23 Define the following terms: inheritance, superclass, subclass, the keywords super and this, casting objects, the modifiers protected and final.

11.24 Indicate true or false for the following statements:

- A protected datum or method can be accessed by any class in the same package.
- A protected datum or method can be accessed by any class in different packages.
- A protected datum or method can be accessed by its subclasses in any package.
- A final class can have instances.
- A final class can be extended.
- A final method can be overridden.
- You can always successfully cast an instance of a subclass to a superclass.
- You can always successfully cast an instance of a superclass to a subclass.
11.25 Describe the difference between method matching and method binding.

11.26 What is polymorphism? What is dynamic binding?

**Programming Exercises**

**Sections 11.2–11.4**

11.1 *(The Triangle class)* Design a class named **Triangle** that extends **GeometricObject**. The class contains:

- Three **double** data fields named **side1**, **side2**, and **side3** with default values 1.0 to denote three sides of the triangle.
- A no-arg constructor that creates a default triangle.
- A constructor that creates a triangle with the specified **side1**, **side2**, and **side3**.
- The accessor methods for all three data fields.
- A method named **getArea()** that returns the area of this triangle.
- A method named **getPerimeter()** that returns the perimeter of this triangle.
- A method named **toString()** that returns a string description for the triangle.

For the formula to compute the area of a triangle, see Exercise 2.21. The **toString()** method is implemented as follows:

```
return "Triangle: side1 = " + side1 + " side2 = " + side2 + 
    " side3 = " + side3;
```

Draw the UML diagram for the classes **Triangle** and **GeometricObject**. Implement the class. Write a test program that creates a **Triangle** object with sides 1, 1.5, 1, color **yellow** and **filled true**, and displays the area, perimeter, color, and whether filled or not.

**Sections 11.5–11.11**

11.2 *(The Person, Student, Employee, Faculty, and Staff classes)* Design a class named **Person** and its two subclasses named **Student** and **Employee**. Make **Faculty** and **Staff** subclasses of **Employee**. A person has a name, address, phone number, and email address. A student has a class status (freshman, sophomore, junior, or senior). Define the status as a constant. An employee has an office, salary, and date hired. Define a class named **MyDate** that contains the fields **year**, **month**, and **day**. A faculty member has office hours and a rank. A staff member has a title. Override the **toString** method in each class to display the class name and the person’s name.

Draw the UML diagram for the classes. Implement the classes. Write a test program that creates a **Person**, **Student**, **Employee**, **Faculty**, and **Staff**, and invokes their **toString()** methods.

11.3 *(Subclasses of Account)* In Exercise 8.7, the **Account** class was defined to model a bank account. An account has the properties account number, balance, annual interest rate, and date created, and methods to deposit and withdraw funds. Create two subclasses for checking and saving accounts. A checking account has an overdraft limit, but a savings account cannot be overdrawn.

Draw the UML diagram for the classes. Implement the classes. Write a test program that creates objects of **Account**, **SavingsAccount**, and **CheckingAccount** and invokes their **toString()** methods.
(Implementing MyStack using inheritance) In Listing 11.9, MyStack is implemented using composition. Create a new stack class that extends ArrayList. Draw the UML diagram for the classes. Implement MyStack. Write a test program that prompts the user to enter five strings and displays them in reverse order.

(The Course class) Rewrite the Course class in Listing 10.6. Use an ArrayList to replace an array to store students. You should not change the original contract of the Course class (i.e., the definition of the constructors and methods should not be changed).

(Using ArrayList) Write a program that creates an ArrayList and adds a Loan object, a Date object, a string, a JFrame object, and a Circle object to the list, and use a loop to display all the elements in the list by invoking the object’s toString() method.

(Implementing ArrayList) ArrayList is implemented in the Java API. Implement ArrayList and the methods defined in Figure 11.3. (Hint: Use an array to store the elements in ArrayList. If the size of the ArrayList exceeds the capacity of the current array, create a new array that doubles the size of the current array and copy the contents of the current to the new array.)

(New Account class) An Account class was specified in Exercise 8.7. Design a new Account class as follows:

- Add a new data field name of the String type to store the name of the customer.
- Add a new constructor that constructs an account with the specified name, id, and balance.
- Add a new data field named transactions whose type is ArrayList that stores the transaction for the accounts. Each transaction is an instance of the Transaction class. The Transaction class is defined as shown in Figure 11.7.
- Modify the withdraw and deposit methods to add a transaction to the transactions array list.

All other properties and methods are same as in Exercise 8.7.

Write a test program that creates an Account with annual interest rate 1.5%, balance 1000, id 1122, and name George. Deposit $30, $40, $50 to the account and withdraw $5, $4, $2 from the account. Print an account summary that shows account holder name, interest rate, balance, and all transactions.

The Transaction class describes a transaction for a bank account.