11.1 Introduction

Key Point

Object-oriented programming allows you to define new classes from existing classes. This is called inheritance.

inheritance

As discussed earlier in the book, the procedural paradigm focuses on designing methods and the object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

why inheritance?

Inheritance is an important and powerful feature for reusing software. Suppose you need 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 as 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

Key Point

Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., 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. You can define a specialized class that extends the generalized class. The specialized classes inherit the properties and methods from the general class.

VideoNote
Geometric class hierarchy

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 and be filled or unfilled. Thus a general class `GeometricObject` can be used to model all geometric objects. This class contains the properties `color` and `filled` and their appropriate getter and setter methods. Assume that this class also contains the `dateCreated` property and the `getDateCreated()` and `toString()` methods. The `toString()` method returns a string representation of 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 defined as a subclass of `GeometricObject`. Figure 11.1 shows the relationship among these classes. A triangular arrow pointing to the superclass is used to denote the inheritance relationship between the two classes involved.

subclass
superclass

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 getter and setter methods. The `Circle` class 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 their associated getter and setter 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.

avoid naming conflicts

Note

To avoid a naming conflict with the improved `GeometricObject`, `Circle`, and `Rectangle` classes introduced in Chapter 13, we'll name these classes

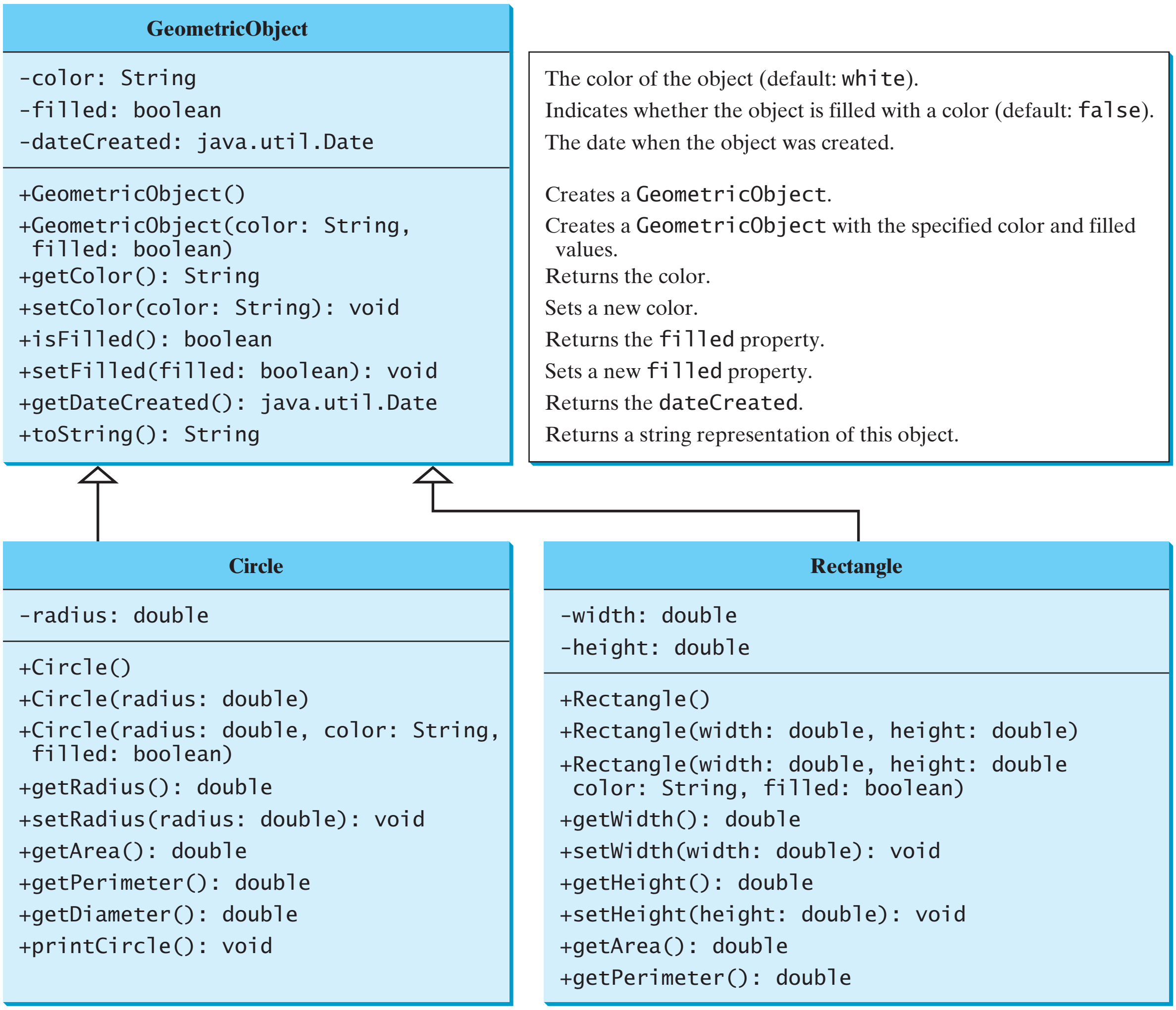


FIGURE 11.1 The `GeometricObject` class is the superclass for `Circle` and `Rectangle`.

`SimpleGeometricObject`, `CircleFromSimpleGeometricObject`, and `RectangleFromSimpleGeometricObject` in this chapter. For simplicity, we will still refer to them in the text as `GeometricObject`, `Circle`, and `Rectangle` classes. The best way to avoid naming conflicts is to place these classes in different packages. However, for simplicity and consistency, all classes in this book are placed in the default package.

LISTING 11.1 SimpleGeometricObject.java

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

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

data fields

constructor

date constructed

```java

  /** Construct a geometric object with the specified color
    *  and filled value */
  public SimpleGeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    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 getter 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 + "\ncolor: " + color +
      " and filled: " + filled;
  }
}
```

LISTING 11.2 CircleFromSimpleGeometricObject.java

extends superclass
data fields
constructor

```java
public class CircleFromSimpleGeometricObject
    extends SimpleGeometricObject {
  private double radius;

  public CircleFromSimpleGeometricObject() {
  }

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

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

```
17    }
18
19    /** Return radius */
20    public double getRadius() {
21      return radius;
22    }
23
24    /** Set a new radius */
25    public void setRadius(double radius) {
26      this.radius = radius;
27    }
28
29    /** Return area */
30    public double getArea() {
31      return radius * radius * Math.PI;
32    }
33
34    /** Return diameter */
35    public double getDiameter() {
36      return 2 * radius;
37    }
38
39    /** Return perimeter */
40    public double getPerimeter() {
41      return 2 * radius * Math.PI;
42    }
43
44    /** Print the circle info */
45    public void printCircle() {
46      System.out.println("The circle is created " + getDateCreated() +
47        " and the radius is " + radius);
48    }
49  }
```

methods

The **Circle** class (Listing 11.2) extends the **GeometricObject** class (Listing 11.1) using the following syntax:

```
          Subclass                   Superclass
                 \                  /
                  v                v
       public class Circle extends GeometricObject
```

The keyword **extends** (lines 1–2) 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 12–17). These two public methods are defined in the superclass **GeometricObject** and are inherited in **Circle**, so they can be used in the **Circle** class.

private member in superclass

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

```
public CircleFromSimpleGeometricObject(
    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 getter and setter methods.

The **Rectangle** class (Listing 11.3) extends the **GeometricObject** class (Listing 11.1) using the following syntax:

```
           Subclass                          Superclass

public class Rectangle extends GeometricObject
```

The keyword **extends** (lines 1–2) tells the compiler that the **Rectangle** class extends the **GeometricObject** class, thus inheriting the methods **getColor**, **setColor**, **isFilled**, **setFilled**, and **toString**.

LISTING 11.3 RectangleFromSimpleGeometricObject.java

extends superclass
data fields
constructor
methods

```
1  public class RectangleFromSimpleGeometricObject
2      extends SimpleGeometricObject {
3    private double width;
4    private double height;
5
6    public RectangleFromSimpleGeometricObject() {
7    }
8
9    public RectangleFromSimpleGeometricObject(
10       double width, double height) {
11     this.width = width;
12     this.height = height;
13   }
14
15   public RectangleFromSimpleGeometricObject(
16       double width, double height, String color, boolean filled) {
17     this.width = width;
18     this.height = height;
19     setColor(color);
20     setFilled(filled);
21   }
22
23   /** Return width */
24   public double getWidth() {
25     return width;
26   }
27
28   /** Set a new width */
29   public void setWidth(double width) {
30     this.width = width;
31   }
32
33   /** Return height */
34   public double getHeight() {
35     return height;
36   }
37
38   /** Set a new height */
39   public void setHeight(double height) {
40     this.height = height;
41   }
```

```
    /** 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 5) and a **Rectangle** object (line 13).

LISTING 11.4 TestCircleRectangle.java

```
 1  public class TestCircleRectangle {
 2    public static void main(String[] args) {
 3       CircleFromSimpleGeometricObject circle =
 4         new CircleFromSimpleGeometricObject(1);
 5      System.out.println("A circle " + circle.toString());
 6      System.out.println("The color is " + circle.getColor());
 7      System.out.println("The radius is " + circle.getRadius());
 8      System.out.println("The area is " + circle.getArea());
 9      System.out.println("The diameter is " + circle.getDiameter());
10
11      RectangleFromSimpleGeometricObject rectangle =
12        new RectangleFromSimpleGeometricObject(2, 4);
13      System.out.println("\nA rectangle " + rectangle.toString());
14      System.out.println("The area is " + rectangle.getArea());
15      System.out.println("The perimeter is " +
16        rectangle.getPerimeter());
17    }
18  }
```

Circle object
invoke toString
invoke getColor

Rectangle object
invoke toString

```
A circle created on Thu Feb 10 19:54:25 EST 2011
color: white and filled: false
The color is white
The radius is 1.0
The area is 3.141592653589793
The diameter is 2.0
A rectangle created on Thu Feb 10 19:54:25 EST 2011
color: white and filled: false
The area is 8.0
The perimeter is 12.0
```

Note the following points regarding inheritance:

- 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. (more in subclass)
- 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 accessors/mutators if defined in the superclass. (private data fields)

nonextensible is-a

- Not all is-a relationships should be modeled using inheritance. For example, a square is a rectangle, but you should not extend a **Square** class from a **Rectangle** class, because the **width** and **height** properties are not appropriate for a square. Instead, you should define a **Square** class to extend the **GeometricObject** class and define the **side** property for the side of a square.

no blind extension

- 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.

multiple inheritance

single inheritance

- 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 Section 13.4.

Check Point

11.1 True or false? A subclass is a subset of a superclass.

11.2 What keyword do you use to define a subclass?

11.3 What is single inheritance? What is multiple inheritance? Does Java support multiple inheritance?

11.3 Using the **super** Keyword

Key Point

*The keyword **super** refers to the superclass and can be used to invoke the superclass's methods and constructors.*

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

Section 9.14, 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

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

The syntax to call a superclass's 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 be the first statement of the subclass's constructor; this is the only way to explicitly invoke a superclass constructor. For example, the constructor in lines 12–17 in Listing 11.2 can be replaced by the following code:

```
public CircleFromSimpleGeometricObject(
    double radius, String color, boolean filled) {
```

```java
    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.

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
}
```

Equivalent

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

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

Equivalent

```java
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 called *constructor chaining*.

constructor chaining

Consider the following code:

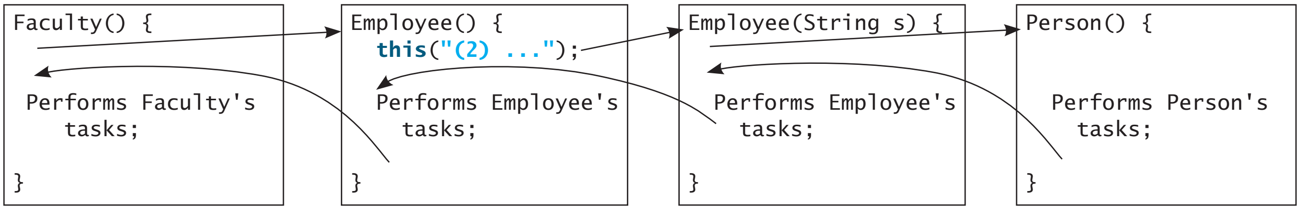
```java
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  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 {
```

invoke overloaded constructor

```
23     public Person() {
24       System.out.println("(1) Performs Person's tasks");
25     }
26   }
```

```
(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 13). 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 illustrated in the following figure.



no-arg constructor

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:

```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

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.

no-arg constructor

Design Guide

If possible, you should provide a no-arg constructor 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:

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

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

```
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.

Check Point

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

```
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)

```
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)

11.5 How does a subclass invoke its superclass's constructor?

11.6 True or false? When invoking a constructor from a subclass, its superclass's no-arg constructor is always invoked.

11.4 Overriding Methods

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

Key Point

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*.

method overriding

The `toString` method in the `GeometricObject` class (lines 46–49 in Listing 11.1) returns the string representation of a geometric object. This method can be overridden to return the string representation of a circle. To override it, add the following new method in the `Circle` class in Listing 11.2.

```
public class CircleFromSimpleGeometricObject
    extends SimpleGeometricObject {
  // Other methods are omitted

  // Override the toString method defined in the superclass
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

`toString` in superclass

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 7).

no super.super.methodName()

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

Several points are worth noting:

override accessible instance method

- 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.

cannot override static method

- 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`.

Check Point

11.7 True or false? You can override a private method defined in a superclass.

11.8 True or false? You can override a static method defined in a superclass.

11.9 How do you explicitly invoke a superclass's constructor from a subclass?

11.10 How do you invoke an overridden superclass method from a subclass?

11.5 Overriding vs. Overloading

Key Point

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.

You learned about overloading methods in Section 6.8. 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 `A` 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 overloads the method in B
  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 `A` to display `10`, and `a.p(10.0)` invokes the `p(double i)` method defined in class `B` to display `20.0`.

Note the following:

- Overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class or different classes related by inheritance.
- Overridden methods have the same signature and return type; overloaded methods have the same name but a different parameter list.

override annotation

To avoid mistakes, you can use a special Java syntax, called *override annotation*, to place `@Override` before the method in the subclass. For example:

```
public class CircleFromSimpleGeometricObject
    extends SimpleGeometricObject {
  // Other methods are omitted

  @Override
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

`toString` in superclass

This annotation denotes that the annotated method is required to override a method in the superclass. If a method with this annotation does not override its superclass's method, the compiler will report an error. For example, if `toString` is mistyped as `tostring`, a compile error is reported. If the override annotation isn't used, the compile won't report an error. Using annotation avoids mistakes.

Check Point

11.11 Identify the problems in the following code:

```
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;
  }
}

class B extends Circle {
  private double length;

  B(double radius, double length) {
    Circle(radius);
    length = length;
  }

  @Override
```

```
    public double getArea() {
      return getArea() * length;
    }
  }
```

11.12 Explain the difference between method overloading and method overriding.

11.13 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?

11.14 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?

11.15 If a method in a subclass has the same name as a method in its superclass with different parameter types, is the method overridden or overloaded?

11.16 What is the benefit of using the `@Override` annotation?

11.6 The `Object` Class and Its `toString()` Method

Key Point

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 {
  ...
}
```

Equivalent

```
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. This section introduces the `toString` method in the `Object` class.

toString()

The signature of the `toString()` method is:

```
public String toString()
```

string representation

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 output for this 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())`. Thus, you could replace `System.out.println(loan.toString())` with `System.out.println(loan)`.

print object

11.7 Polymorphism

Polymorphism means that a variable of a supertype can refer to a subtype object.

Key Point

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*. Therefore, you can say that `Circle` is a subtype of `GeometricObject` and `GeometricObject` is a supertype for `Circle`.

subtype

supertype

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

```
1  public class PolymorphismDemo {
2    /** Main method */
3    public static void main(String[] args) {
4      // Display circle and rectangle properties
5      displayObject(new CircleFromSimpleGeometricObject
6               (1, "red", false));
7      displayObject(new RectangleFromSimpleGeometricObject
8               (1, 1, "black", true));
9    }
10
11   /** Display geometric object properties */
12   public static void displayObject(SimpleGeometricObject object) {
13     System.out.println("Created on " + object.getDateCreated() +
14       ". Color is " + object.getColor());
15   }
16 }
```

polymorphic call

polymorphic call

```
Created on Mon Mar 09 19:25:20 EDT 2011. Color is red
Created on Mon Mar 09 19:25:20 EDT 2011. Color is black
```

The method `displayObject` (line 12) takes a parameter of the `GeometricObject` type. You can invoke `displayObject` by passing any instance of `GeometricObject` (e.g., `new CircleFromSimpleGeometricObject(1, "red", false)` and `new RectangleFromSimpleGeometricObject(1, 1, "black", false)` in lines 5–8). An object of a subclass can be used wherever its superclass object is used. This is commonly known as *polymorphism* (from a Greek word meaning "many forms"). In simple terms, polymorphism means that a variable of a supertype can refer to a subtype object.

what is polymorphism?

11.8 Dynamic Binding

Key Point

A method can be implemented in several classes along the inheritance chain. The JVM decides which method is invoked at runtime.

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

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

declared type

actual type

dynamic binding

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 that declares a variable is called the variable's *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**, because **o** references 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 for the implementation of 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.

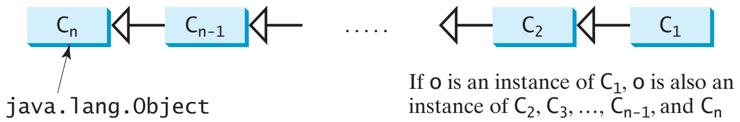


FIGURE 11.2 The method to be invoked is dynamically bound at runtime.

VideoNote
Polymorphism and dynamic binding demo

Listing 11.6 gives an example to demonstrate dynamic binding.

LISTING 11.6 DynamicBindingDemo.java

polymorphic call

dynamic binding

override toString()

```
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 {
  @Override
  public String toString() {
```

```
20         return "Student" ;
21       }
22     }
23
24     class Person extends Object {
25       @Override
26       public String toString() {
27         return "Person" ;
28       }
29     }
```

override toString()

```
Student
Student
Person
java.lang.Object@130c19b
```

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`. The 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.

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; and invoking `m(new Object())` (line 6) causes the `toString` method defined in the `Object` class to be invoked.

matching vs. binding

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 the parameter type, number of parameters, and order of the parameters at compile time. A method may be implemented in several classes along the inheritance chain. The JVM dynamically binds the implementation of the method at runtime, decided by the actual type of the variable.

Check Point

11.17 What is polymorphism? What is dynamic binding?

11.18 Describe the difference between method matching and method binding.

11.19 Can you assign `new int[50]`, `new Integer[50]`, `new String[50]`, or `new Object[50]`, into a variable of `Object[]` type?

11.20 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    Integer[] list1 = {12, 24, 55, 1};
    Double[] list2 = {12.4, 24.0, 55.2, 1.0};
    int[] list3 = {1, 2, 3};
    printArray(list1);
    printArray(list2);
    printArray(list3);
  }

  public static void printArray(Object[] list) {
    for (Object o: list)
```

```
            System.out.print(o + " ");
          System.out.println();
        }
      }
```

11.21 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    new Person().printPerson();
    new Student().printPerson();
  }
}

class Student extends Person {
  @Override
  public String getInfo() {
    return "Student";
  }
}

class Person {
  public String getInfo() {
    return "Person";
  }

  public void printPerson() {
    System.out.println(getInfo());
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    new Person().printPerson();
    new Student().printPerson();
  }
}

class Student extends Person {
  private String getInfo() {
    return "Student";
  }
}

class Person {
  private String getInfo() {
    return "Person";
  }

  public void printPerson() {
    System.out.println(getInfo());
  }
}
```

(b)

11.22 Show the output of following program:

```
public class Test {
  public static void main(String[] args) {
    A a = new A(3);
  }
}

class A extends B {
  public A(int t) {
    System.out.println("A's constructor is invoked");
  }
}

class B {
  public B() {
    System.out.println("B's constructor is invoked");
  }
}
```

Is the no-arg constructor of `Object` invoked when `new A(3)` is invoked?

11.23 Show the output of following program:

```
public class Test {
  public static void main(String[] args) {
    new A();
    new B();
  }
}
```

```
class A {
  int i = 7;

  public A() {
    setI(20);
    System.out.println("i from A is " + i);
  }

  public void setI(int i) {
    this.i = 2 * i;
  }
}

class B extends A {
  public B() {
    System.out.println("i from B is " + i);
  }

  public void setI(int i) {
    this.i = 3 * i;
  }
}
```

11.9 Casting Objects and the **instanceof** Operator

One object reference can be typecast into another object reference. This is called casting object.

Key Point

casting object

In the preceding section, the statement

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

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

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

implicit casting

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

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

```
Student b = o;
```

explicit casting

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 *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:

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

upcasting
downcasting

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:

ClassCastException

instanceof

```
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. The 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()**. Therefore, 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 an object of any subtype.

lowercase keywords

Note

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

casting analogy

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 (lines 21–22).

LISTING 11.7 CastingDemo.java

```
1  public class CastingDemo {
2    /** Main method */
3    public static void main(String[] args) {
4      // Create and initialize two objects
5      Object object1 = new CircleFromSimpleGeometricObject(1);
6      Object object2 = new RectangleFromSimpleGeometricObject(1, 1);
7
8      // Display circle and rectangle
9      displayObject(object1);
10     displayObject(object2);
11   }
12
```

```
13    /** A method for displaying an object */
14    public static void displayObject(Object object) {
15      if (object instanceof CircleFromSimpleGeometricObject) {
16        System.out.println("The circle area is " +
17          ((CircleFromSimpleGeometricObject)object).getArea());
18        System.out.println("The circle diameter is " +
19          ((CircleFromSimpleGeometricObject)object).getDiameter());
20      }
21      else if (object instanceof
22                    RectangleFromSimpleGeometricObject) {
23        System.out.println("The rectangle area is " +
24          ((RectangleFromSimpleGeometricObject)object).getArea());
25      }
26    }
27  }
```

polymorphic call (line 17)

polymorphic call (line 24)

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

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 14–26), 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 24) 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

precedes casting

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

Casting a primitive type value is different from casting an object reference. Casting a primitive type value returns a new value. For example:

```
int age = 45;
byte newAge = (byte)age; // A new value is assigned to newAge
```

However, casting an object reference does not create a new object. For example:

```
Object o = new Circle();
Circle c = (Circle)o; // No new object is created
```

Now reference variables **o** and **c** point to the same object.

Check Point

11.24 Indicate true or false for the following statements:

- 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 For the `GeometricObject` and `Circle` classes in Listings 11.1 and 11.2, answer the following questions:

a. Assume are `circle` and `object` created as follows:

```
Circle circle = new Circle(1);
GeometricObject object = new GeometricObject();
```

Are the following Boolean expressions true or false?

```
(circle instanceof GeometricObject)
(object instanceof GeometricObject)
(circle instanceof Circle)
(object instanceof Circle)
```

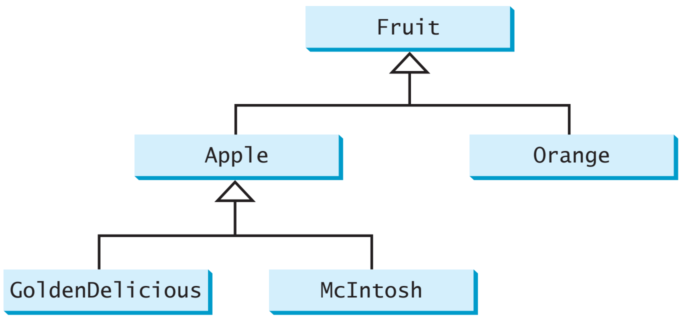
b. Can the following statements be compiled?

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

c. Can the following statements be compiled?

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

11.26 Suppose that `Fruit`, `Apple`, `Orange`, `GoldenDelicious`, and `McIntosh` are defined in the following inheritance hierarchy:



Assume that the following code is given:

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

Answer the following questions:

a. Is `fruit instanceof Fruit`?

b. Is `fruit instanceof Orange`?

c. Is `fruit instanceof Apple`?

d. Is `fruit instanceof GoldenDelicious`?

e. Is `fruit instanceof McIntosh`?

f. Is `orange instanceof Orange`?

g. Is `orange instanceof Fruit`?

h. Is `orange instanceof Apple`?

i. Suppose the method `makeAppleCider` is defined in the `Apple` class. Can `fruit` invoke this method? Can `orange` invoke this method?

j. Suppose the method `makeOrangeJuice` is defined in the `Orange` class. Can `orange` invoke this method? Can `fruit` invoke this method?

k. Is the statement `Orange p = new Apple()` legal?

l. Is the statement `McIntosh p = new Apple()` legal?

m. Is the statement `Apple p = new McIntosh()` legal?

11.27 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    Object fruit = new Fruit();
    Object apple = (Apple)fruit;
  }
}

class Apple extends Fruit {
}

class Fruit {
}
```

11.10 The **Object**'s **equals** Method

Key Point

Like the `toString()` *method, the* `equals(Object)` *method is another useful method defined in the* `Object` *class.*

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.

The `equals` method is overridden in many classes in the Java API, such as `java.lang.String` and `java.util.Date`, to compare whether the contents of two objects are equal. You have already used the `equals` method to compare two strings in Section 4.4.7, 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;
  else
    return this == o;
}
```

== vs. equals

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.

equals(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 CheckPoint Question 11.29.

Check Point

11.28 Does every object have a `toString` method and an `equals` method? Where do they come from? How are they used? Is it appropriate to override these methods?

11.29 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 following the code; 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)

If `Object` is replaced by `Circle` in the `Test` class, what would be the output to run `Test` using the `Circle` class in (a) and (b), respectively?

11.11 The `ArrayList` Class

Key Point

An `ArrayList` object can be used to store a list of objects.

VideoNote
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, which can be used to store an unlimited number of objects. Figure 11.3 shows some methods in **ArrayList**.

java.util.ArrayList<E>	
+ArrayList()	Creates an empty list.
+add(o: E): void	Appends a new element o at the end of this list.
+add(index: int, o: E): void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int): E	Returns the element from this list at the specified index.
+indexOf(o: Object): int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object): int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the first element o from this list. Returns true if an element is removed.
+size(): int	Returns the number of elements in this list.
+remove(index: int): boolean	Removes the element at the specified index. Returns true if an element is removed.
+set(index: int, o: E): E	Sets the element at the specified index.

FIGURE 11.3 An **ArrayList** stores an unlimited number of objects.

ArrayList is known as a generic class with a generic type **E**. You can specify a concrete type to replace **E** when creating an **ArrayList**. For example, the following statement creates an **ArrayList** and assigns its reference to variable **cities**. This **ArrayList** object can be used to store strings.

```
ArrayList<String> cities = new ArrayList<String>();
```

The following statement creates an **ArrayList** and assigns its reference to variable **dates**. This **ArrayList** object can be used to store dates.

```
ArrayList<java.util.Date> dates = new ArrayList<java.util.Date> ();
```

Note

Since JDK 7, the statement

```
ArrayList<AConcreteType> list = new ArrayList<AConcreteType>();
```

can be simplified by

```
ArrayList<AConcreteType> list = new ArrayList<>();
```

The concrete type is no longer required in the constructor thanks to a feature called *type inference*. The compiler is able to infer the type from the variable declaration. More discussions on generics including how to define custom generic classes and methods will be introduced in Chapter 19, Generics.

type inference

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

LISTING 11.8 TestArrayList.java

```
import java.util.ArrayList;

```

import ArrayList

create ArrayList
add element
list size
contains element?
element index
is empty?
remove element
remove element
toString()
get element
create ArrayList

```
public class TestArrayList {
  public static void main(String[] args) {
    // Create a list to store cities
    ArrayList<String> cityList = new ArrayList<>();

    // Add some cities in the list
    cityList.add("London");
    // cityList now contains [London]
    cityList.add("Denver");
    // cityList now contains [London, Denver]
    cityList.add("Paris");
    // cityList now contains [London, Denver, Paris]
    cityList.add("Miami");
    // cityList now contains [London, Denver, Paris, Miami]
    cityList.add("Seoul");
    // Contains [London, Denver, Paris, Miami, Seoul]
    cityList.add("Tokyo");
    // Contains [London, Denver, Paris, Miami, Seoul, Tokyo]

    System.out.println("List size? " + cityList.size());
    System.out.println("Is Miami in the list? " +
      cityList.contains("Miami"));
    System.out.println("The location of Denver in the list? "
      + cityList.indexOf("Denver"));
    System.out.println("Is the list empty? " +
      cityList.isEmpty()); // Print false

    // Insert a new city at index 2
    cityList.add(2, "Xian");
    // Contains [London, Denver, Xian, Paris, Miami, Seoul, Tokyo]

    // Remove a city from the list
    cityList.remove("Miami");
    // Contains [London, Denver, Xian, Paris, Seoul, Tokyo]

    // Remove a city at index 1
    cityList.remove(1);
    // Contains [London, Xian, Paris, Seoul, Tokyo]

    // Display the contents in the list
    System.out.println(cityList.toString());

    // Display the contents in the list in reverse order
    for (int i = cityList.size() - 1; i >= 0; i--)
      System.out.print(cityList.get(i) + " ");
    System.out.println();

    // Create a list to store two circles
    ArrayList<CircleFromSimpleGeometricObject> list
      = new ArrayList<>();

    // Add two circles
    list.add(new CircleFromSimpleGeometricObject(2));
    list.add(new CircleFromSimpleGeometricObject(3));

    // Display the area of the first circle in the list
    System.out.println("The area of the circle? " +
      list.get(0).getArea());
  }
}
```

```
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
```

Since the **ArrayList** is in the **java.util** package, it is imported in line 1. The program creates an **ArrayList** of strings using its no-arg constructor and assigns the reference to **cityList** (line 6). The **add** method (lines 9–19) adds strings to the end of list. So, after **cityList.add("London")** (line 9), the list contains

add(Object)

```
[London]
```

After **cityList.add("Denver")** (line 11), the list contains

```
[London, Denver]
```

After adding **Paris**, **Miami**, **Seoul**, and **Tokyo** (lines 13–19), the list contains

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

Invoking **size()** (line 22) returns the size of the list, which is currently **6**. Invoking **contains("Miami")** (line 24) checks whether the object is in the list. In this case, it returns **true**, since **Miami** is in the list. Invoking **indexOf("Denver")** (line 26) returns the index of **Denver** in the list, which is **1**. If **Denver** were not in the list, it would return **-1**. The **isEmpty()** method (line 28) checks whether the list is empty. It returns **false**, since the list is not empty.

size()

The statement **cityList.add(2, "Xian")** (line 31) inserts an object into the list at the specified index. After this statement, the list becomes

add(index, Object)

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

The statement **cityList.remove("Miami")** (line 35) removes the object from the list. After this statement, the list becomes

remove(Object)

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

The statement **cityList.remove(1)** (line 39) removes the object at the specified index from the list. After this statement, the list becomes

remove(index)

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

The statement in line 43 is same as

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

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

toString()

The **get(index)** method (line 47) returns the object at the specified index.

get(index)

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

array vs. ArrayList

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**.

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

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

You cannot use the **get(index)** and **set(index, element)** methods if the element is not in the list. It is easy to add, insert, 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 that you can sort an array using the **java.util.Arrays.sort(array)** method. To sort an array list, use the **java.util.Collections.sort(arraylist)** method.

Suppose you want to create an **ArrayList** for storing integers. Can you use the following code to create a list?

```
ArrayList<int> list = new ArrayList<>();
```

No. This will not work because the elements stored in an **ArrayList** must be of an object type. You cannot use a primitive data type such as **int** to replace a generic type. However, you can create an **ArrayList** for storing **Integer** objects as follows:

```
ArrayList<Integer> list = new ArrayList<>();
```

Listing 11.9 gives a program that prompts the user to enter a sequence of numbers and displays the distinct numbers in the sequence. Assume that the input ends with **0** and **0** is not counted as a number in the sequence.

LISTING 11.9 DistinctNumbers.java

create an array list

contained in list?

add to list

```
import java.util.ArrayList;
import java.util.Scanner;

public class DistinctNumbers {
  public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<>();

    Scanner input = new Scanner(System.in);
    System.out.print("Enter integers (input ends with 0): ");
    int value;

    do {
      value = input.nextInt(); // Read a value from the input

      if (!list.contains(value) && value != 0)
        list.add(value); // Add the value if it is not in the list
    } while (value != 0);
```

```
18
19        // Display the distinct numbers
20        for (int i = 0; i < list.size(); i++)
21          System.out.print(list.get(i) + " ");
22      }
23    }
```

```
Enter numbers (input ends with 0): 1 2 3 2 1 6 3 4 5 4 5 1 2 3 0 ↵
The distinct numbers are: 1 2 3 6 4 5
```

The program creates an `ArrayList` for `Integer` objects (line 6) and repeatedly reads a value in the loop (lines 12–17). For each value, if it is not in the list (line 15), add it to the list (line 16). You can rewrite this program using an array to store the elements rather than using an `ArrayList`. However, it is simpler to implement this program using an `ArrayList` for two reasons.

- First, the size of an `ArrayList` is flexible so you don't have to specify its size in advance. When creating an array, its size must be specified.
- Second, `ArrayList` contains many useful methods. For example, you can test whether an element is in the list using the `contains` method. If you use an array, you have to write additional code to implement this method.

You can traverse the elements in an array using a foreach loop. The elements in an array list can also be traversed using a foreach loop using the following syntax:

```
for (elementType element: arrayList) {
  // Process the element
}
```

For example, you can replace the code in lines 20-21 using the following code:

```
for (int number: list)
  System.out.print(number + " ");
```

11.30 How do you do the following?

a. Create an `ArrayList` for storing double values?

b. Append an object to a list?

c. Insert an object at the beginning of a list?

d. Find the number of objects in a list?

e. Remove a given object from a list?

f. Remove the last object from the list?

g. Check whether a given object is in a list?

h. Retrieve an object at a specified index from a list?

11.31 Identify the errors in the following code.

```
ArrayList<String> 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));
```

11.32 Suppose the `ArrayList list` contains `{"Dallas", "Dallas", "Houston", "Dallas"}`. What is the list after invoking `list.remove("Dallas")` one time? Does the following code correctly remove all elements with value `"Dallas"` from the list? If not, correct the code.

```
for (int i = 0; i < list.size(); i++)
  list.remove("Dallas");
```

11.33 Explain why the following code displays `[1, 3]` rather than `[2, 3]`.

```
ArrayList<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);
list.remove(1);
System.out.println(list);
```

11.34 Explain why the following code is wrong.

```
ArrayList<Double> list = new ArrayList<>();
list.add(1);
```

11.12 Useful Methods for Lists

Key Point *Java provides the methods for creating a list from an array, for sorting a list, and finding maximum and minimum element in a list, and for shuffling a list.*

array to array list

Often you need to create an array list from an array of objects or vice versa. You can write the code using a loop to accomplish this, but an easy way is to use the methods in the Java API. Here is an example to create an array list from an array:

```
String[] array = {"red", "green", "blue"};
ArrayList<String> list = new ArrayList<>(Arrays.asList(array));
```

array list to array

The static method `asList` in the `Arrays` class returns a list that is passed to the `ArrayList` constructor for creating an `ArrayList`. Conversely, you can use the following code to create an array of objects from an array list.

```
String[] array1 = new String[list.size()];
list.toArray(array1);
```

Invoking `list.toArray(array1)` copies the contents from `list` to `array1`.

sort a list

If the elements in a list are comparable such as integers, double, or strings, you can use the static `sort` method in the `java.util.Collections` class to sort the elements. Here are examples:

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));
java.util.Collections.sort(list);
System.out.println(list);
```

`max` and `min` methods

You can use the static `max` and `min` in the `java.util.Collections` class to return the maximum and minimal element in a list. Here are examples:

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));
System.out.println(java.util.Collections.max(list));
System.out.println(java.util.Collections.min(list));
```

You can use the static **shuffle** method in the **java.util.Collections** class to perform a random shuffle for the elements in a list. Here are examples:

shuffle method

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));
java.util.Collections.shuffle(list);
System.out.println(list);
```

Check Point

11.35 Correct errors in the following statements:

```
int[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));
```

11.36 Correct errors in the following statements:

```
int[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
System.out.println(java.util.Collections.max(array));
```

11.13 Case Study: A Custom Stack Class

This section designs a stack class for holding objects.

Key Point

VideoNote

The MyStack class

Section 10.6 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 11.10. The UML diagram for the class is shown in Figure 11.4.

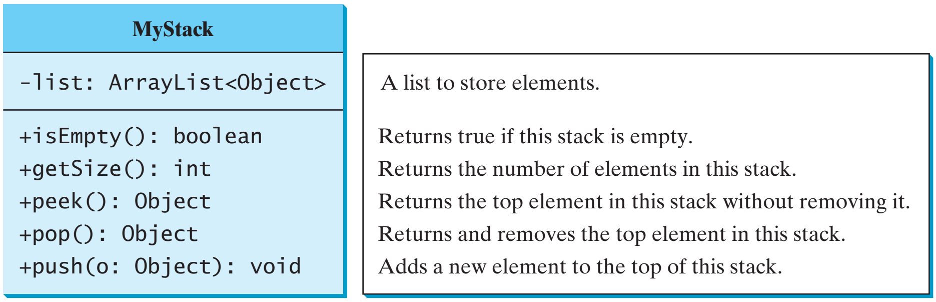


FIGURE 11.4 The **MyStack** class encapsulates the stack storage and provides the operations for manipulating the stack.

LISTING 11.10 MyStack.java

```
import java.util.ArrayList;

public class MyStack {
  private ArrayList<Object> list = new ArrayList<>();

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

  public int getSize() {
    return list.size();
  }

  public Object peek() {
    return list.get(getSize() - 1);
  }
```

array list

stack empty?

get stack size

peek stack

remove

push

```
17
18   public Object pop() {
19     Object o = list.get(getSize() - 1);
20     list.remove(getSize() - 1);
21     return o;
22   }
23
24   public void push(Object o) {
25     list.add(o);
26   }
27
28   @Override
29   public String toString() {
30     return "stack: " + list.toString();
31   }
32 }
```

An array list is created to store the elements in the stack (line 4). The **isEmpty()** method (lines 6–8) returns **list.isEmpty()**. The **getSize()** method (lines 10–12) returns **list.size()**. The **peek()** method (lines 14–16) 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 18–22) removes the top element from the stack and returns it. The **push(Object element)** method (lines 24–26) adds the specified element to the stack. The **toString()** method (lines 28–31) 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.

composition
is-a
has-a

Design Guide

In Listing 11.10, **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 could also implement **MyStack** as a subclass of **ArrayList** (see Programming Exercise 11.10). 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**.

11.14 The protected Data and Methods

Key Point

A protected member of a class can be accessed from a subclass.

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

why protected?

Often it is desirable to allow subclasses to access data fields or methods defined in the superclass, but not to allow nonsubclasses to access these data fields and methods. To accomplish this, you can use the **protected** keyword. This way you can access protected data fields or methods in a superclass from its subclasses.

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

Visibility increases

$\longrightarrow$

private, default (no modifier), 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.

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 (the default) 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.

TABLE 11.2 Data and Methods Visibility

Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

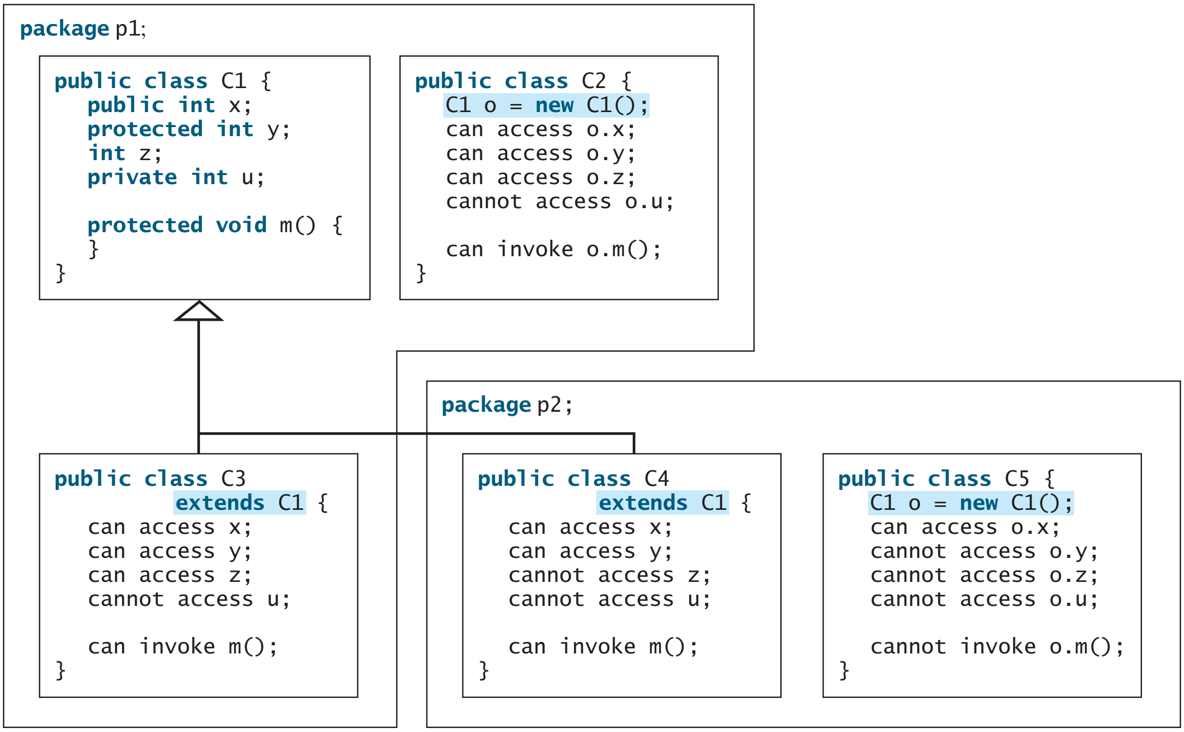


FIGURE 11.5 Visibility modifiers are used to control how data and methods are accessed.

Your class can be used in two ways: (1) for creating instances of the class and (2) 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 for 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 as on the class. A class with no modifier (i.e., not a public class) is not accessible by classes from other packages.

change visibility

Note

A subclass may override a protected method defined 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.

Check Point

11.37 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?

11.38 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?

11.39 In the following code, the classes `A` and `B` are in the same package. If the question marks in (a) 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?

```
package p1;

public class A {
  ___?___ int i;

  ___?___ void m() {
    ...
  }
}
```

(a)

```
package p1;

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

(b)

11.40 In the following code, the classes `A` and `B` are in different packages. If the question marks in (a) 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?

```
package p1;

public class A {
  ___?___ int i;

  ___?___  void m() {
    ...
  }
}
```

(a)

```
package p2;

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

(b)

11.15 Preventing Extending and Overriding

Key Point

Neither a final class nor a final method can be extended. A final data field is a constant.

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 `A` is final and cannot be extended:

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

You also can define a method to be final; a final method cannot be overridden by its subclasses.

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

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

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

Note

The modifiers `public`, `protected`, `private`, `static`, `abstract`, and `final` 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.

Check Point

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

11.42 Indicate true or false for the following statements:

a. A protected datum or method can be accessed by any class in the same package.

b. A protected datum or method can be accessed by any class in different packages.

c. A protected datum or method can be accessed by its subclasses in any package.

d. A final class can have instances.

e. A final class can be extended.

f. A final method can be overridden.

Key Terms

actual type 424
casting objects 427
constructor chaining 417
declared type 424
dynamic binding 424
inheritance 410
`instanceof` 428
is-a relationship 440
method overriding 419
multiple inheritance 416
override 000
polymorphism 423
`protected` 440
single inheritance 416
subclass 410
subtype 423
superclass 410
supertype 423
type inference 433

Chapter Summary

1. You can define 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`.