CHAPTER

11

INHERITANCE AND POLYMORPHISM

Objectives

- To define 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 and 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 construct an array list from an array, to sort and shuffle a list, and to obtain max and min element from a list (§11.12).
- To implement a `Stack` class using `ArrayList` (§11.13).
- To enable data and methods in a superclass accessible from subclasses using the `protected` visibility modifier (§11.14).
- To prevent class extending and method overriding using the `final` modifier (§11.15).

11.1 Introduction

Key Point

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

inheritance

As discussed in the preceding chapter, 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 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 special type of `GeometricObject`. Figure 11.1 shows the relationship among these classes. A triangular arrow pointing to the generalized class 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. Therefore, `Circle` and `Rectangle` are subclasses of `GeometricObject`, and `GeometricObject` is the superclass for `Circle` and `Rectangle`. 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

is-a relationship

The subclass and its superclass are said to form a *is-a* relationship. A `Circle` object is a special type of general `GeometricObject`. 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.

width and height

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()`

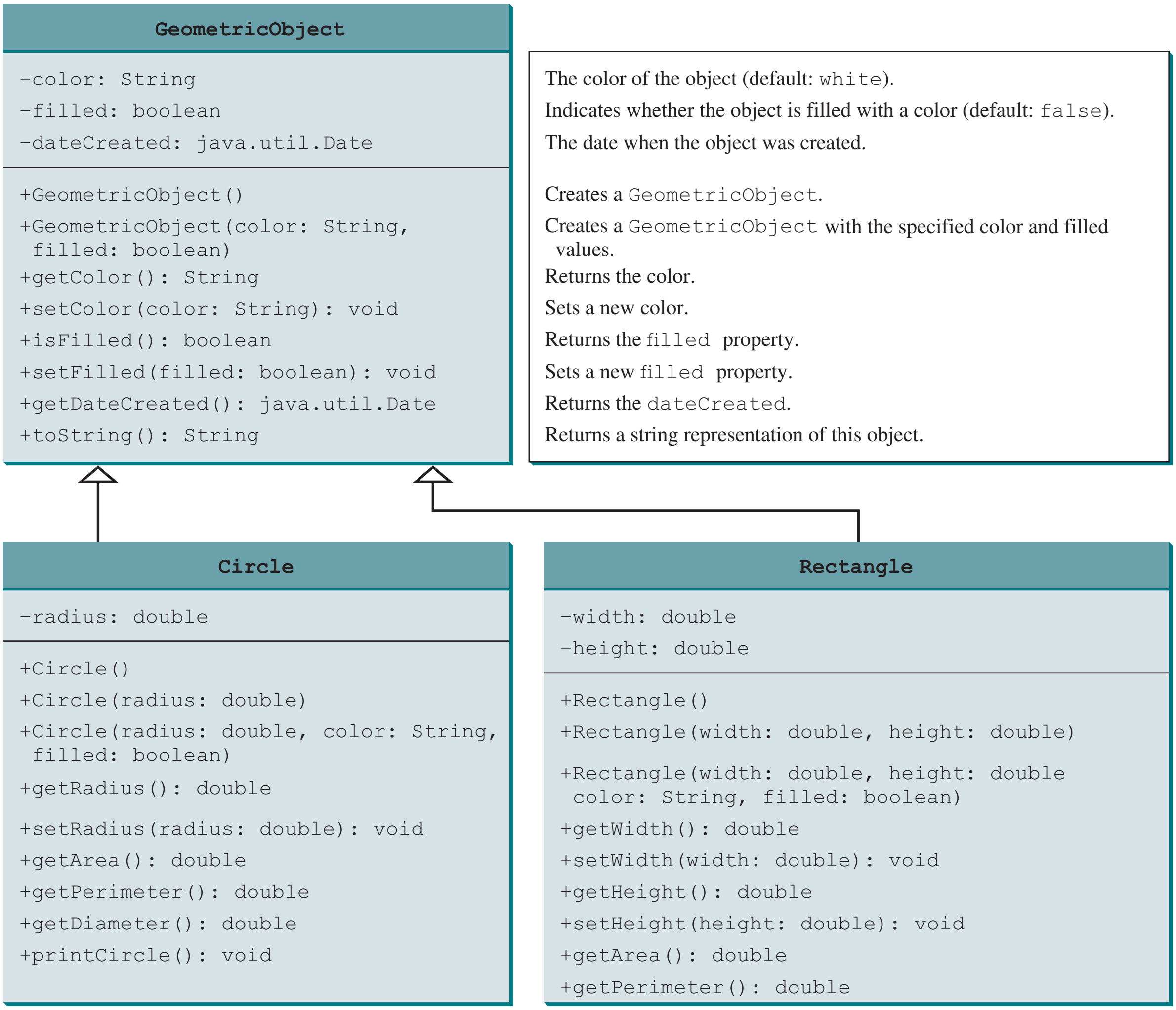

GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
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.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

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

methods for returning the area and perimeter of the rectangle. Note that you may have used the terms width and length to describe the sides of a rectangle in geometry. The common terms used in computer science are width and height, where width refers to the horizontal length, and height to the vertical length.

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

LISTING 11.1 GeometricObject.java

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

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

```

data fields

constructor

date constructed

```
  /** Construct a geometric object with the specified color
    *  and filled value */
  public GeometricObject(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 Circle.java

extends superclass

data fields

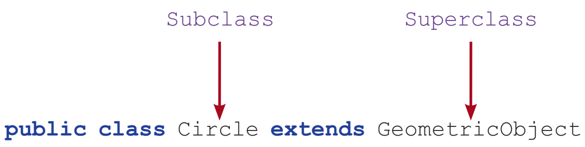
constructor

```
public class Circle extends GeometricObject {
  private double radius;

  public Circle() {
  }

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

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

```

methods

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

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



```
public class Circle extends GeometricObject
```

The keyword **extends** (lines 1 and 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 14 and 15). The public methods defined in the superclass **GeometricObject** are inherited in **Circle**, so they can be used in the **Circle** class.

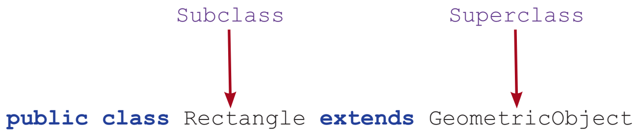
You might attempt to use the data fields **color** and **filled** directly in the constructor as follows:

private member in superclass

```
public Circle(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:



```
public class Rectangle extends GeometricObject
```

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

LISTING 11.3 Rectangle.java

extends superclass

data fields

constructor

methods

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

```java
  /** 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 11).

Listing 11.4 TestCircleRectangle.java

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

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 13.5.

Check Point

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

11.2.2 What keyword do you use to define a subclass?

11.2.3 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:

1. To call a superclass constructor
2. 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(arguments);
```

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 11–16 in Listing 11.2 can be replaced by the following code:

```
public Circle(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.

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:

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

Equivalent

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

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

Equivalent

```
public ClassName(parameters) {
  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

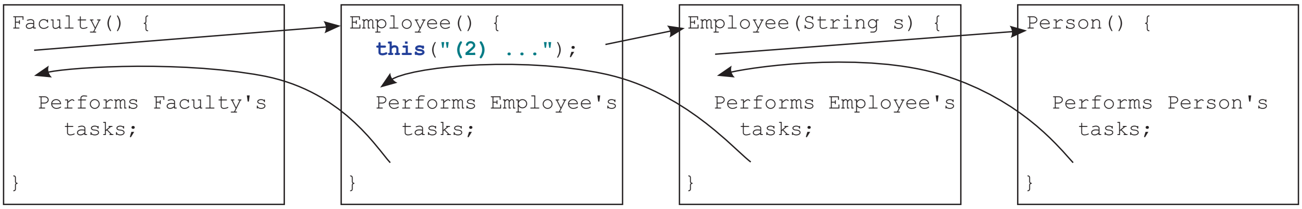
Consider the following code:

```
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) Invokes 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");
  }
}
```

invoke overloaded constructor

```
(1) Performs Person's tasks
(2) Invokes 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.



```
Faculty() {

  Performs Faculty's
    tasks;
}
```

```
Employee() {
  this("(2) ...");
  Performs Employee's
    tasks;
}
```

```
Employee(String s) {

  Performs Employee's
    tasks;
}
```

```
Person() {

  Performs Person's
    tasks;
}
```

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(arguments);
```

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.

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

Check Point

```
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)

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

11.3.3 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 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:

```
1  public class Circle extends GeometricObject {
2    // Other methods are omitted
3
4    // Override the toString method defined in the superclass
5    public String toString() {
6      return super.toString() + "\nradius is " + radius;
7    }
8  }
```

`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 6).

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

- The overriding method must have the same signature as the overridden method and same or compatible return type. Compatible means that the overriding method's return type is a subtype of the overridden method's return type.
- 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.

cannot override static method

Check Point

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

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

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

11.4.4 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 or compatible 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 TestOverriding {
  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 TestOverloading {
  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 `TestOverriding` 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 `TestOverloading` 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 in different classes related by inheritance.
- Overridden methods have the same signature; overloaded methods have the same name but different parameter lists.

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

override annotation

```
public class Circle extends GeometricObject {
  // 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 its 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 compiler won't report an error. Using the `@Override` annotation avoids mistakes.

Check Point

11.5.1 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;
  }
}
```

11.5.2 Explain the difference between method overloading and method overriding.

11.5.3 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?

11.5.4 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?

11.5.5 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?

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

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 in (a) and (b) are the same:

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

(a)

Equivalent

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

(b)

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;
}
```

print object

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)`.

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.

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 Circle(1, "red", false));
 6      displayObject(new Rectangle(1, 1, "black", true));
 7    }
 8
 9    /** Display geometric object properties */
10    public static void displayObject(GeometricObject object) {
11      System.out.println("Created on " + object.getDateCreated() +
12        ". Color is " + object.getColor());
13    }
14  }
```

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 10) takes a parameter of the `GeometricObject` type. You can invoke `displayObject` by passing any instance of `GeometricObject` (e.g., `new Circle(1, "red", false)` and `new Rectangle(1, 1, "black", true)` in lines 5 and 6). 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?

11.7.1 What are the three pillars of object-oriented programming? What is polymorphism?

Check Point

11.8 Dynamic Binding

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

Key Point

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());
```

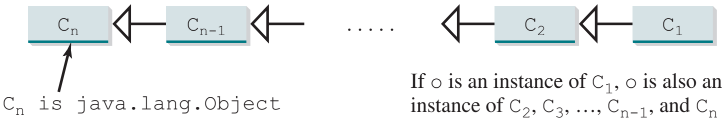
declared type

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 at runtime. 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 that 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.



C_n is java.lang.Object

If o is an instance of C_1, o is also an instance of C_2, C_3, …, C_{n-1}, and C_n

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()

override toString()

```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 {
  @Override
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  @Override
26    public String toString() {
27      return "Person";
28    }
29  }
```

```
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 `toString` method is implemented in `Student`, `Person`, and `Object`. 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

11.8.1 What is dynamic binding?

11.8.2 Describe the difference between method matching and method binding.

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

11.8.4 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();
  }
}
```

11.8.5 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)

11.8.6 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?

11.8.7 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

In the preceding section, the statement

casting object

```
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 `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

ClassCastException

instanceof

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 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 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
void someMethod(Object myObject) {
  ... // 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 declare **myObject** as a **Circle** type in the first place? To enable generic programming, it is a good practice to declare a variable with a supertype that 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 and 6), a **circle** and a **rectangle**, and invokes the **displayObject** method to display them (lines 9 and 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
 1  public class CastingDemo {
 2    /** Main method */
 3    public static void main(String[] args) {
 4      // Create and initialize two objects
 5      Object object1 = new Circle(1);
 6      Object object2 = new Rectangle(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 Circle) {
16        System.out.println("The circle area is " +
17          ((Circle)object).getArea());
18        System.out.println("The circle diameter is " +
19          ((Circle)object).getDiameter());
20      }
21      else if (object instanceof Rectangle) {
22        System.out.println("The rectangle area is " +
23          ((Rectangle)object).getArea());
24      }
25    }
26  }
```

polymorphic call

polymorphic call

```
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 and 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 and 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 (**.**) has higher precedence than 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

11.9.1 Indicate true or false for the following statements:

a. You can always successfully cast an instance of a subclass to a superclass.

b. You can always successfully cast an instance of a superclass to a subclass.

11.9.2 For the **GeometricObject** and **Circle** classes in Listings 11.1 and 11.2, answer the following questions:

a. Assume that **circle** and **object1** are created as follows:

```
Circle circle = new Circle(1);
GeometricObject object1 = new GeometricObject();
```

Are the following Boolean expressions true or false?

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

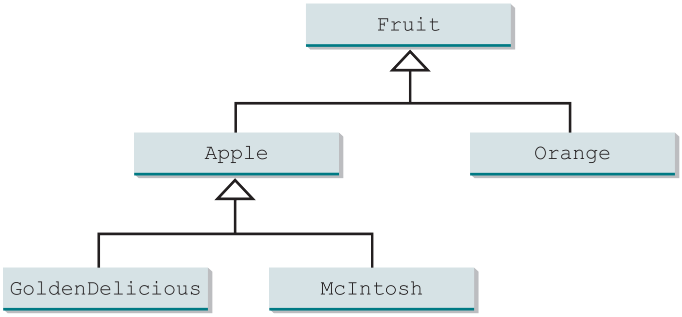
b. Can the following statements be compiled?

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

c. Can the following statements be compiled?

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

11.9.3 Suppose **Fruit**, **Apple**, **Orange**, **GoldenDelicious**, and **McIntosh** are defined in the following inheritance hierarchy:



Assume 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?

11.9.4 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

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

Key Point

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:

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

== 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 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 CheckPoint Question 11.10.2.

equals(Object)

Check Point

11.10.1 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?

11.10.2 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 following 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 o) {
    return this.radius ==
      ((Circle)o).radius;
  }
}
```

(b)

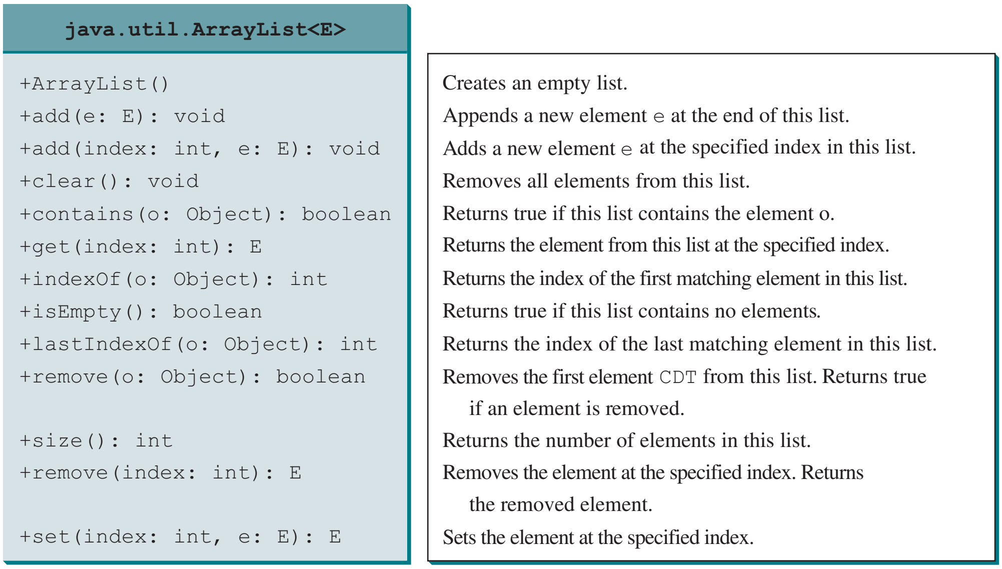
If **Object** is replaced by **Circle** in the **Test** class, what would be the output to run **Test** using the **Circle** class in (a) and (b), respectively?

11.11 The ArrayList Class

Key Point

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

Now we are ready to introduce a very useful class for storing objects. You can create an array to store objects. However, 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(e: E): void	Appends a new element e at the end of this list.
+add(index: int, e: E): void	Adds a new element e at the specified index in this list.
+clear(): void	Removes all 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 CDT from this list. Returns true if an element is removed.
+size(): int	Returns the number of elements in this list.
+remove(index: int): E	Removes the element at the specified index. Returns the removed element.
+set(index: int, e: 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<>();
```

type inference

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.

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

LISTING 11.8 TestArrayList.java

import ArrayList

create ArrayList

add element

list size

contains element?

element index

is empty?

remove element

remove element

toString()

get element

create ArrayList

```
import java.util.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<Circle> list = new ArrayList<>();

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

    // Display the area of the first circle in the list
    System.out.println("The area of the circle? " +
59         list.get(0).getArea());
60     }
61   }
```

```
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
```

add(Object)

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. Thus, after cityList.add("London") (line 9), the list contains

```
[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]
```

size()

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.

add(index, Object)

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

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

remove(Object)

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

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

remove(index)

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

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

The statement in line 43 is same as

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

toString()

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.

get(index)

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

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.

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 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> listOfIntegers = 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> listOfIntegers = new ArrayList<>();
```

remove(int) vs. remove(Integer)

Note the `remove(int index)` method removes an element at the specified index. To remove an integer value v from `listOfIntegers`, you need to use `listOfIntegers.remove(Integer.valueOf(v))`. This is not a good design in the Java API because it could easily lead to mistakes. It would be much better if `remove(int)` is renamed `removeAt(int)`.

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 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

```
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

15          if (!list.contains(value) && value != 0)
16            list.add(value); // Add the value if it is not in the list
17        } while (value != 0);
18
19        // Display the distinct numbers
20        System.out.print("The distinct integers are: ");
21        for (int i = 0; i < list.size(); i++)
22          System.out.print(list.get(i) + " ");
23      }
24    }
```

contained in list?

add to list

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

1. 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.

2. **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.

foreach loop

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 and 21 using the following code:

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

or

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

Note the elements in **list** are **Integer** objects. They are automatically unboxed into **int** in this foreach loop.

Check Point

11.11.1 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 a list?

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

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

11.11.2 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));
```

11.11.3 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");
```

11.11.4 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);
```

How do you remove integer value `3` from the list?

11.11.5 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 for 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);
```

sort a list

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

shuffle method

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 some examples:

```
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

11.12.1 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));
```

11.12.2 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

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.

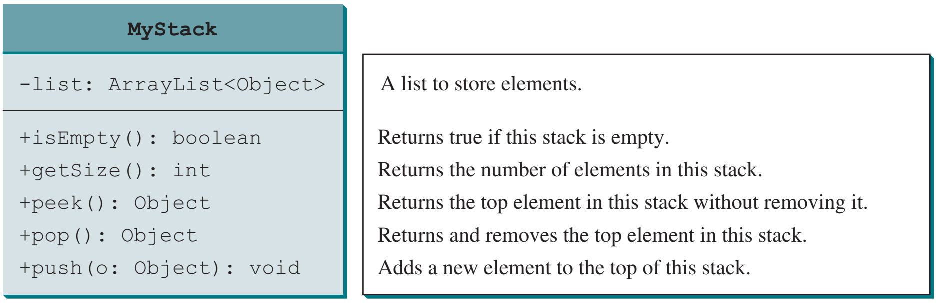

MyStack
-list: ArrayList<Object>
+isEmpty(): boolean
+getSize(): int
+peek(): Object
+pop(): Object
+push(o: Object): void
A list to store elements.
Returns true if this stack is empty.
Returns the number of elements in this stack.
Returns the top element in this stack without removing it.
Returns and removes the top element in this stack.
Adds a new element to the top of this stack.

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();
```

array list

stack empty?

get stack size

peek stack

remove

push

```
 8    }
 9
10    public int getSize() {
11      return list.size();
12    }
13
14    public Object peek() {
15       return list.get(getSize() - 1);
16    }
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

has-a

Design Guide

In Listing 11.10, `MyStack` contains `ArrayList`. The relationship between `MyStack` and `ArrayList` is *composition*. Composition essentially means declaring an instance variable for referencing an object. This object is said to be composed. 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`.

Check Point

11.13.1 Write statements that create a `MyStack` and add number `11` to the stack.

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 in different packages 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

→

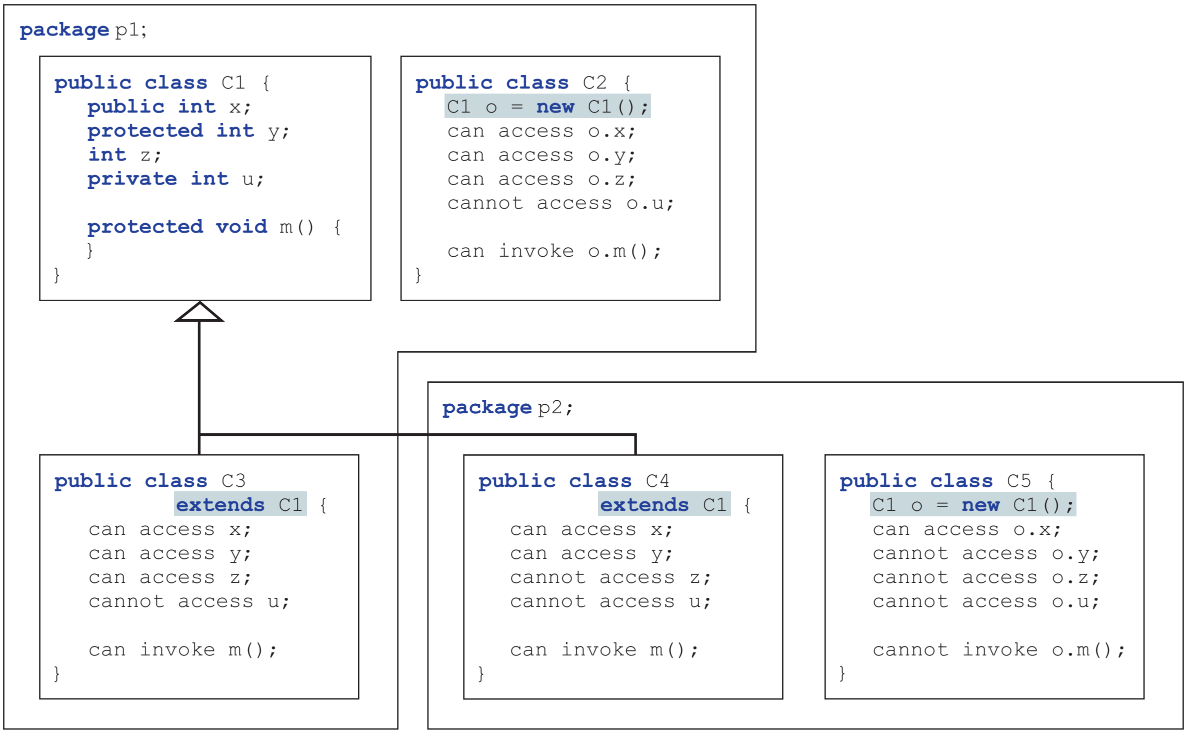
private, default (no modifier), protected, public

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, a subclass **`C3`** in the same package, a subclass **`C4`** in a different package, and a class **`C5`** in a different package.

Use the **`private`** modifier to hide the members of the class completely so 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	✓	–	–	–



```
package p1;

public class C1 {
   public int x;
   protected int y;
   int z;
   private int u;

   protected void m() {
   }
}
```

```
public class C2 {
   C1 o = new C1();
   can access o.x;
   can access o.y;
   can access o.z;
   cannot access o.u;

   can invoke o.m();
}
```

```
public class C3
      extends C1 {
   can access x;
   can access y;
   can access z;
   cannot access u;

   can invoke m();
}
```

```
package p2;

public class C4
      extends C1 {
   can access x;
   can access y;
   cannot access z;
   cannot access u;

   can invoke m();
}
```

```
public class C5 {
   C1 o = new C1();
   can access o.x;
   cannot access o.y;
   cannot access o.z;
   cannot access o.u;

   cannot invoke o.m();
}
```

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

11.14.1 What modifier should you use on a class so a class in the same package can access it, but a class in a different package cannot access it?

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

11.14.3 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)

11.14.4 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 a class is final and cannot be a parent class. The **Math** class is a final class. The **String**, **StringBuilder**, and **StringBuffer** classes, and all wrapper classes for primitive data types 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

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

11.15.2 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	426
casting objects	429
constructor chaining	419
declared type	426
dynamic binding	426
inheritance	412
`instanceof`	430
is-a relationship	412
method overriding	421
multiple inheritance	418
override	421
polymorphism	425
`protected`	442
single inheritance	418
subclass	412
subtype	412
superclass	412
supertype	412
type inference	435

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`.

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 and the same or compatible return type 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 superclass 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`). This is known as polymorphism.

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*. This is known as dynamic binding.

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

13. You can use the `ArrayList` class to create an object to store a list of objects.

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

15. You can use the `final` modifier to indicate a class is final and cannot be extended and to indicate a method is final and cannot be overridden.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

MyProgrammingLab™

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 a 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 Programming Exercise 2.19. The `toString()` method is implemented as follows:

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

Draw the UML diagrams for the classes `Triangle` and `GeometricObject` and implement the classes. Write a test program that prompts the user to enter three sides of the triangle, a color, and a Boolean value to indicate whether the triangle is filled. The program should create a `Triangle` object with these sides and set the `color` and `filled` properties using the input. The program should display the area, perimeter, color, and true or false to indicate whether it is filled or not.

Sections 11.5–11.14

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 e-mail 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. Use the `MyDate` class defined in Programming Exercise 10.14 to create an object for date hired. 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 and implement them. Write a test program that creates a `Person`, `Student`, `Employee`, `Faculty`, and `Staff`, and invokes their `toString()` methods.

11.3 (*Subclasses of `Account`*) In Programming Exercise 9.7, 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 and implement them. Write a test program that creates objects of `Account`, `SavingsAccount`, and `CheckingAccount` and invokes their `toString()` methods.

11.4 (*Maximum element in* ***ArrayList***) Write the following method that returns the maximum value in an **ArrayList** of integers. The method returns **null** if the list is **null** or the list size is **0**.

```
public static Integer max(ArrayList<Integer> list)
```

Write a test program that prompts the user to enter a sequence of numbers ending with **0** and invokes this method to return the largest number in the input.

11.5 (*The* ***Course*** *class*) Rewrite the **Course** class in Listing 10.6. Use an **ArrayList** to replace an array to store students. Draw the new UML diagram for the class. You should not change the original contract of the **Course** class (i.e., the definition of the constructors and methods should not be changed, but the private members may be changed.)

11.6 (*Use* ***ArrayList***) Write a program that creates an **ArrayList** and adds a **Loan** object, a **Date** object, a string, 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.

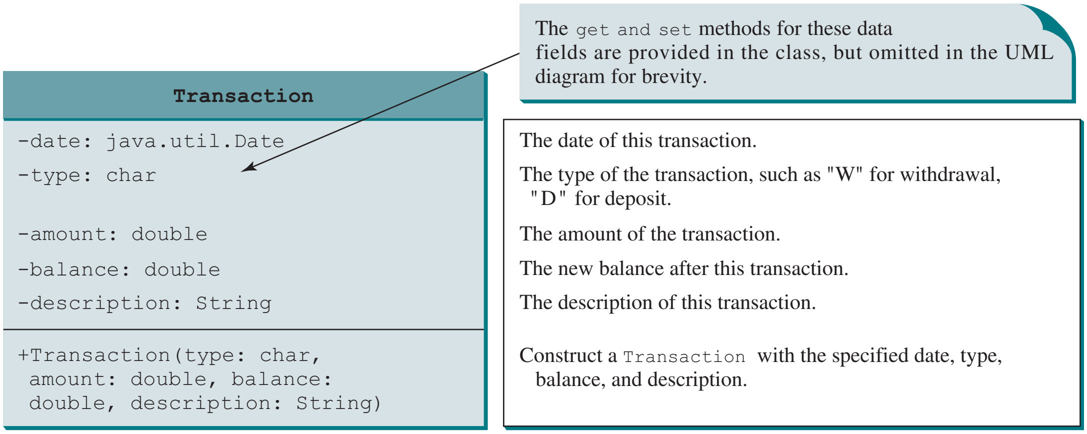
11.7 (*Shuffle* ***ArrayList***) Write the following method that shuffles the elements in an **ArrayList** of integers:

```
public static void shuffle(ArrayList<Integer> list)
```

VideoNote
New Account class

****11.8** (*New* ***Account*** *class*) An **Account** class was specified in Programming Exercise 9.7. 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, which is defined as shown in Figure 11.6.


The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

```
Transaction
-date: java.util.Date
-type: char

-amount: double
-balance: double
-description: String
+Transaction(type: char,
 amount: double, balance:
 double, description: String)
```

The date of this transaction.

The type of the transaction, such as "W" for withdrawal, "D" for deposit.

The amount of the transaction.

The new balance after this transaction.

The description of this transaction.

Construct a Transaction with the specified date, type, balance, and description.

FIGURE 11.6 The **Transaction** class describes a transaction for a bank account.

- Modify the **withdraw** and **deposit** methods to add a transaction to the **transactions** array list.
- All other properties and methods are the same as in Programming Exercise 9.7.

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

***11.9** (*Largest rows and columns*) Write a program that randomly fills in **0**s and **1**s into an n-by-n matrix, prints the matrix, and finds the rows and columns with the most **1**s. (*Hint*: Use two **ArrayList**s to store the row and column indices with the most **1**s.) Here is a sample run of the program:

```
Enter the array size n: 4 ↵Enter
The random array is
0011
0011
1101
1010
The largest row index: 2
The largest column index: 2, 3
```

11.10 (*Implement **MyStack** using inheritance*) In Listing 11.10, **MyStack** is implemented using composition. Define a new stack class that extends **ArrayList**.

Draw the UML diagram for the classes then implement **MyStack**. Write a test program that prompts the user to enter five strings and displays them in reverse order.

11.11 (*Sort **ArrayList***) Write the following method that sorts an **ArrayList** of numbers:

```
public static void sort(ArrayList<Integer> list)
```

Write a test program that prompts the user to enter five numbers, stores them in an array list, and displays them in increasing order.

11.12 (*Sum **ArrayList***) Write the following method that returns the sum of all numbers in an **ArrayList**:

```
public static double sum(ArrayList<Double> list)
```

Write a test program that prompts the user to enter five numbers, stores them in an array list, and displays their sum.

***11.13** (*Remove duplicates*) Write a method that removes the duplicate elements from an array list of integers using the following header:

```
public static void removeDuplicate(ArrayList<Integer> list)
```

Write a test program that prompts the user to enter 10 integers to a list and displays the distinct integers in their input order and separated by exactly one space. Here is a sample run:

```
Enter 10 integers: 34 5 3 5 6 4 33 2 2 4 ↵Enter
The distinct integers are 34 5 3 6 4 33 2
```

11.14 (*Combine two lists*) Write a method that returns the union of two array lists of integers using the following header:

```
public static ArrayList<Integer> union(
  ArrayList<Integer> list1, ArrayList<Integer> list2)
```

For example, the addition of two array lists {2, 3, 1, 5} and {3, 4, 6} is {2, 3, 1, 5, 3, 4, 6}. Write a test program that prompts the user to enter two lists, each with five integers, and displays their union. The numbers are separated by exactly one space. Here is a sample run:

```
Enter five integers for list1: 3 5 45 4 3 ↵Enter
Enter five integers for list2: 33 51 5 4 13 ↵Enter
The combined list is 3 5 45 4 3 33 51 5 4 13
```

***11.15** (*Area of a convex polygon*) A polygon is convex if it contains any line segments that connects two points of the polygon. Write a program that prompts the user to enter the number of points in a convex polygon, enter the points clockwise, then displays the area of the polygon. For the formula for computing the area of a polygon, see http://www.mathwords.com/a/area_convex_polygon.htm. Here is a sample run of the program:

```
Enter the number of points: 7 ↵Enter
Enter the coordinates of the points:
  -12 0 -8.5 10 0 11.4 5.5 7.8 6 -5.5 0 -7 -3.5 -5.5 ↵Enter
The total area is 244.57
```

****11.16** (*Addition quiz*) Rewrite Listing 5.1, RepeatAdditionQuiz.java, to alert the user if an answer is entered again. (*Hint:* use an array list to store answers.) Here is a sample run of the program:

```
What is 5 + 9? 12 ↵Enter
Wrong answer. Try again. What is 5 + 9? 34 ↵Enter
Wrong answer. Try again. What is 5 + 9? 12 ↵Enter
You already entered 12
Wrong answer. Try again. What is 5 + 9? 14 ↵Enter
You got it!
```

****11.17** (*Algebra: perfect square*) Write a program that prompts the user to enter an integer `m` and find the smallest integer `n` such that `m * n` is a perfect square. (*Hint:* Store all smallest factors of `m` into an array list. `n` is the product of the factors that appear an odd number of times in the array list. For example, consider `m` = 90, store the factors 2, 3, 3, and 5 in an array list. 2 and 5 appear an odd number of times in the array list. Thus, `n` is 10.) Here is a sample run of the program:

```
Enter an integer m: 1500 ↵Enter
The smallest number n for m * n to be a perfect square is 15
m * n is 22500
```

```
Enter an integer m: 63 ↵Enter
The smallest number n for m * n to be a perfect square is 7
m * n is 441
```

****11.18** (*ArrayList of Character*) Write a method that returns an array list of `Character` from a string using the following header:

```
public static ArrayList<Character> toCharacterArray(String s)
```

For example, `toCharacterArray("abc")` returns an array list that contains characters `'a'`, `'b'`, and `'c'`.

****11.19** (*Bin packing using first fit*) The bin packing problem is to pack the objects of various weights into containers. Assume each container can hold a maximum of 10 pounds. The program uses an algorithm that places an object into the first bin in which it would fit. Your program should prompt the user to enter the total number of objects and the weight of each object. The program displays the total number of containers needed to pack the objects and the contents of each container. Here is a sample run of the program:

```
Enter the number of objects: 6
Enter the weights of the objects: 7 5 2 3 5 8
Container 1 contains objects with weight 7 2
Container 2 contains objects with weight 5 3
Container 3 contains objects with weight 5
Container 4 contains objects with weight 8
```

Does this program produce an optimal solution, that is, finding the minimum number of containers to pack the objects?

CHAPTER

12

Exception Handling and Text I/O

Objectives

- To get an overview of exceptions and exception handling (§12.2).
- To explore the advantages of using exception handling (§12.2).
- To distinguish exception types: `Error` (fatal) vs. `Exception` (nonfatal) and checked vs. unchecked (§12.3).
- To declare exceptions in a method header (§12.4.1).
- To throw exceptions in a method (§12.4.2).
- To write a `try-catch` block to handle exceptions (§12.4.3).
- To explain how an exception is propagated (§12.4.3).
- To obtain information from an exception object (§12.4.4).
- To develop applications with exception handling (§12.4.5).
- To use the `finally` clause in a `try-catch` block (§12.5).
- To use exceptions only for unexpected errors (§12.6).
- To rethrow exceptions in a `catch` block (§12.7).
- To create chained exceptions (§12.8).
- To define custom exception classes (§12.9).
- To discover file/directory properties, to delete and rename files/directories, and to create directories using the `File` class (§12.10).
- To write data to a file using the `PrintWriter` class (§12.11.1).
- To use try-with-resources to ensure that the resources are closed automatically (§12.11.2).
- To read data from a file using the `Scanner` class (§12.11.3).
- To understand how data is read using a `Scanner` (§12.11.4).
- To develop a program that replaces text in a file (§12.11.5).
- To read data from the Web (§12.12).
- To develop a Web crawler (§12.13).

12.1 Introduction

Exceptions are runtime errors. Exception handling enables a program to deal with runtime errors and continue its normal execution.

Runtime errors occur while a program is running if the JVM detects an operation that is impossible to carry out. For example, if you access an array using an index that is out of bounds, you will get a runtime error with an **ArrayIndexOutOfBoundsException**. If you enter a **double** value when your program expects an integer, you will get a runtime error with an **InputMismatchException**.

exception

In Java, runtime errors are thrown as exceptions. An *exception* is an object that represents an error or a condition that prevents execution from proceeding normally. If the exception is not handled, the program will terminate abnormally. How can you handle the exception so the program can continue to run or else terminate gracefully? This chapter introduces this subject, and text input and output.

12.2 Exception-Handling Overview

Exceptions are thrown from a method. The caller of the method can catch and handle the exception.

VideoNote
Exception-handling advantages

To demonstrate exception handling, including how an exception object is created and thrown, let's begin with the example in Listing 12.1, which reads in two integers and displays their quotient.

Listing 12.1 Quotient.java

read two integers

integer division

```
import java.util.Scanner;

public class Quotient {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    System.out.println(number1 + " / " + number2 + " is " +
      (number1 / number2));
  }
}
```

```
Enter two integers: 5 2 ↵Enter
5 / 2 is 2
```

```
Enter two integers: 3 0 ↵Enter
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Quotient.main(Quotient.java:13)
```

If you entered **0** for the second number, a runtime error would occur, because you cannot divide an integer by **0**. (*Note a floating-point number divided by* **0** *does not raise an exception.*) A simple way to fix this error is to add an **if** statement to test the second number, as shown in Listing 12.2.

Listing 12.2 QuotientWithIf.java

```java
import java.util.Scanner;

public class QuotientWithIf {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    if (number2 != 0)
      System.out.println(number1 + " / " + number2
        + " is " + (number1 / number2));
    else
      System.out.println("Divisor cannot be zero ");
  }
}
```

read two integers

test number2

```
Enter two integers: 5 0 ↵Enter
Divisor cannot be zero
```

Before introducing exception handling, let us rewrite Listing 12.2 to compute a quotient using a method, as shown in Listing 12.3.

Listing 12.3 QuotientWithMethod.java

```java
import java.util.Scanner;

public class QuotientWithMethod {
  public static int quotient(int number1, int number2) {
    if (number2 == 0) {
      System.out.println("Divisor cannot be zero");
      System.exit(1);
    }

    return number1 / number2;
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    int result = quotient(number1, number2);
    System.out.println(number1 + " / " + number2 + " is "
      + result);
  }
}
```

quotient method

terminate the program

read two integers

invoke method

```
Enter two integers: 5 3 ↵Enter
5 / 3 is 1
```

```
Enter two integers: 5 0 ↵Enter
Divisor cannot be zero
```

The method **quotient** (lines 4–11) returns the quotient of two integers. If **number2** is **0**, it cannot return a value, so the program is terminated in line 7. This is clearly a problem. You should not let the method terminate the program—the *caller* should decide whether to terminate the program.

How can a method notify its caller when an exception has occurred? Java enables a method to throw an exception that can be caught and handled by the caller. Listing 12.3 can be rewritten, as shown in Listing 12.4.

Listing 12.4 QuotientWithException.java

quotient method
throw exception
read two integers
try block
invoke method
catch block

```
import java.util.Scanner;

public class QuotientWithException {
  public static int quotient(int number1, int number2) {
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");

    return number1 / number2;
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    try {
      int result = quotient(number1, number2);
      System.out.println(number1 + " / " + number2 + " is "
        + result);
    }
    catch (ArithmeticException ex) {
      System.out.println("Exception: an integer " +
        "cannot be divided by zero ");
    }

    System.out.println("Execution continues ...");
  }
}
```

If an Arithmetic Exception occurs

```
Enter two integers: 5 3 ↵Enter
5 / 3 is 1
Execution continues ...
```

```
Enter two integers: 5 0 ↵Enter
Exception: an integer cannot be divided by zero
Execution continues ...
```

If number2 is 0, the method throws an exception (line 6) by executing

throw statement

```
throw new ArithmeticException("Divisor cannot be zero");
```

exception

throw exception

The value thrown, in this case new ArithmeticException("Divisor cannot be zero"), is called an *exception*. The execution of a throw statement is called *throwing an exception*. The exception is an object created from an exception class. In this case, the exception class is java.lang.ArithmeticException. The constructor ArithmeticException(str) is invoked to construct an exception object, where str is a message that describes the exception.

handle exception

When an exception is thrown, the normal execution flow is interrupted. As the name suggests, to “throw an exception” is to pass the exception from one place to another. The statement for invoking the method is contained in a try block. The try block (lines 19–23) contains the code that is executed in normal circumstances. The exception is caught by the catch block. The code in the catch block is executed to *handle the exception*. Afterward, the statement (line 29) after the catch block is executed.

The throw statement is analogous to a method call, but instead of calling a method, it calls a catch block. In this sense, a catch block is like a method definition with a parameter that matches the type of the value being thrown. Unlike a method, however, after the catch block is executed, the program control does not return to the throw statement; instead, it executes the next statement after the catch block.

The identifier ex in the catch–block header

```
catch (ArithmeticException ex)
```

catch–block parameter

acts very much like a parameter in a method. Thus, this parameter is referred to as a catch–block parameter. The type (e.g., ArithmeticException) preceding ex specifies what kind of exception the catch block can catch. Once the exception is caught, you can access the thrown value from this parameter in the body of a catch block.

In summary, a template for a try-throw-catch block may look as follows:

```
try {
  Code to run;
  A statement or a method that may throw an exception;
  More code to run;
}
catch (type ex) {
  Code to process the exception;
}
```

An exception may be thrown directly by using a throw statement in a try block, or by invoking a method that may throw an exception.

The main method invokes quotient (line 20). If the quotient method executes normally, it returns a value to the caller. If the quotient method encounters an exception, it throws the exception back to its caller. The caller’s catch block handles the exception.

advantage

Now you can see the *advantage* of using exception handling: It enables a method to throw an exception to its caller, enabling the caller to handle the exception. Without this capability, the called method itself must handle the exception or terminate the program. Often the called method does not know what to do in case of error. This is typically the case for the library methods. The library method can detect the error, but only the caller knows what needs to be done when an error occurs. The key benefit of exception handling is separating the detection of an error (done in a called method) from the handling of an error (done in the calling method).

Many library methods throw exceptions. Listing 12.5 gives an example that handles an `InputMismatchException` when reading an input.

LISTING 12.5 InputMismatchExceptionDemo.java

create a Scanner

try block

catch block

```
 1  import java.util.*;
 2
 3  public class InputMismatchExceptionDemo {
 4    public static void main(String[] args) {
 5      Scanner input = new Scanner(System.in);
 6      boolean continueInput = true;
 7
 8      do {
 9        try {
10          System.out.print("Enter an integer: ");
11          int number = input.nextInt();
12
13          // Display the result
14          System.out.println(
15            "The number entered is " + number);
16
17          continueInput = false;
18        }
19        catch (InputMismatchException ex) {
20          System.out.println("Try again. (" +
21            "Incorrect input: an integer is required)");
22          input.nextLine(); // Discard input
23        }
24      } while (continueInput);
25    }
26  }
```

If an InputMismatch Exception occurs

```
Enter an integer: 3.5 ↵Enter
Try again. (Incorrect input: an integer is required)
Enter an integer: 4 ↵Enter
The number entered is 4
```

When executing `input.nextInt()` (line 11), an `InputMismatchException` occurs if the input entered is not an integer. Suppose `3.5` is entered. An `InputMismatchException` occurs and the control is transferred to the `catch` block. The statements in the `catch` block are now executed. The statement `input.nextLine()` in line 22 discards the current input line so the user can enter a new line of input. The variable `continueInput` controls the loop. Its initial value is `true` (line 6) and it is changed to `false` (line 17) when a valid input is received. Once a valid input is received, there is no need to continue the input.

Check Point

12.2.1 What is the advantage of using exception handling?

12.2.2 Which of the following statements will throw an exception?

```
System.out.println(1 / 0);
System.out.println(1.0 / 0);
```

12.2.3 Point out the problem in the following code. Does the code throw any exceptions?

```
long value = Long.MAX_VALUE + 1;
System.out.println(value);
```

12.2.4 What does the JVM do when an exception occurs? How do you catch an exception?

12.2.5 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    try {
      int value = 30;
      if (value < 40)
        throw new Exception("value is too small");
    }
    catch (Exception ex) {
      System.out.println(ex.getMessage());
    }
    System.out.println("Continue after the catch block");
  }
}
```

What would be the output if the line

```
int value = 30;
```

were changed to

```
int value = 50;
```

12.2.6 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    for (int i = 0; i < 2; i++) {
      System.out.print(i + " ");
      try {
        System.out.println(1 / 0);
      }
      catch (Exception ex) {
      }
    }
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    try {
      for (int i = 0; i < 2; i++) {
        System.out.print(i + " ");
        System.out.println(1 / 0);
      }
    }
    catch (Exception ex) {
    }
  }
}
```

(b)

12.3 Exception Types

Key Point

Exceptions are objects, and objects are defined using classes. The root class for exceptions is `java.lang.Throwable`.

The preceding section used the classes `ArithmeticException` and `InputMismatchException`. Are there any other types of exceptions you can use? Can you define your own exception classes? Yes. There are many predefined exception classes in the Java API. Figure 12.1 shows some of them, and in Section 12.9, you will learn how to define your own exception classes.

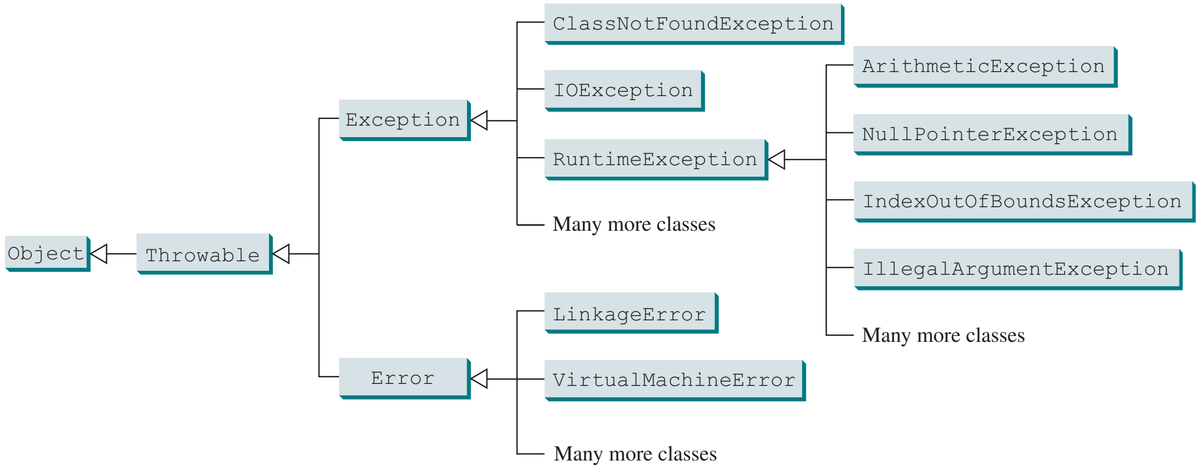


FIGURE 12.1 Exceptions thrown are instances of the classes shown in this diagram, or of subclasses of one of these classes.

Note

The class names `Error`, `Exception`, and `RuntimeException` are somewhat confusing. All three of these classes are exceptions and all of the errors occur at runtime.

The `Throwable` class is the root of exception classes. All Java exception classes inherit directly or indirectly from `Throwable`. You can create your own exception classes by extending `Exception` or a subclass of `Exception`.

The exception classes can be classified into three major types: system errors, exceptions, and runtime exceptions.

system error

- *System errors* are thrown by the JVM and are represented in the `Error` class. The `Error` class describes internal system errors, though such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully. Examples of subclasses of `Error` are listed in Table 12.1.

TABLE 12.1 Examples of Subclasses of `Error`

Class	*Reasons for Exception*
`LinkageError`	A class has some dependency on another class, but the latter class has changed incompatibly after the compilation of the former class.
`VirtualMachineError`	The JVM is broken or has run out of the resources it needs in order to continue operating.

exception

- *Exceptions* are represented in the `Exception` class, which describes errors caused by your program and by external circumstances. These errors can be caught and handled by your program. Examples of subclasses of `Exception` are listed in Table 12.2.

TABLE 12.2 Examples of Subclasses of `Exception`

Class	*Reasons for Exception*
`ClassNotFoundException`	Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the `java` command or if your program were composed of, say, three class files, only two of which could be found.
`IOException`	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file. Examples of subclasses of `IOException` are `InterruptedIOException`, `EOFException` (EOF is short for End of File), and `FileNotFoundException`.

- *Runtime exceptions* are represented in the RuntimeException class, which describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors. Runtime exceptions normally indicate programming errors. Examples of subclasses are listed in Table 12.3.

runtime exception

TABLE 12.3 Examples of Subclasses of RuntimeException

Class	*Reasons for Exception*
ArithmeticException	Dividing an integer by zero. Note floating-point arithmetic does not throw exceptions (see Appendix E, Special Floating-Point Values).
NullPointerException	Attempt to access an object through a null reference variable.
IndexOutOfBoundsException	Index to an array is out of range.
IllegalArgumentException	A method has passed an argument that is illegal or inappropriate.

RuntimeException, Error, and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning the compiler forces the programmer to check and deal with them in a try-catch block or declare it in the method header. Declaring an exception in the method header will be covered in Section 12.4.

unchecked exception

checked exception

In most cases, unchecked exceptions reflect programming logic errors that are unrecoverable. For example, a NullPointerException is thrown if you access an object through a reference variable before an object is assigned to it; an IndexOutOfBoundsException is thrown if you access an element in an array outside the bounds of the array. These are logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in a program. To avoid cumbersome overuse of try-catch blocks, Java does not mandate that you write code to catch or declare unchecked exceptions.

Check Point

12.3.1 Describe the Java Throwable class, its subclasses, and the types of exceptions.

12.3.2 What RuntimeException will the following programs throw, if any?

```java
public class Test {
  public static void main(String[] args) {
    System.out.println(1 / 0);
  }
}
```

(a)

```java
public class Test {
  public static void main(String[] args) {
    int[] list = new int[5];
    System.out.println(list[5]);
  }
}
```

(b)

```java
public class Test {
  public static void main(String[] args) {
    String s = "abc";
    System.out.println(s.charAt(3));
  }
}
```

(c)

```java
public class Test {
  public static void main(String[] args) {
    Object o = new Object();
    String d = (String)o;
  }
}
```

(d)

```java
public class Test {
  public static void main(String[] args) {
    Object o = null;
    System.out.println(o.toString());
  }
}
```

(e)

```java
public class Test {
  public static void main(String[] args) {
    System.out.println(1.0 / 0);
  }
}
```

(f)

12.3.3 What is a checked exception and what is an unchecked exception?

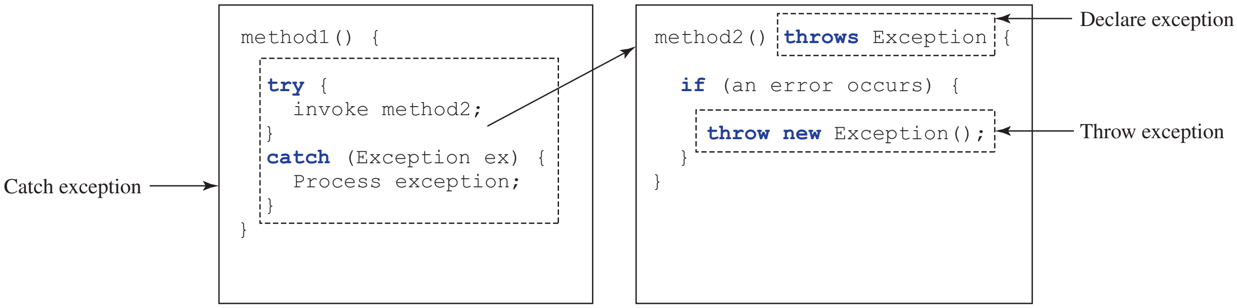
12.4 Declaring, Throwing, and Catching Exceptions

Key Point

A handler for an exception is found by propagating the exception backward through a chain of method calls, starting from the current method.

The preceding sections gave you an overview of exception handling and introduced several predefined exception types. This section provides an in-depth discussion of exception handling.

Java's exception-handling model is based on three operations: *declaring an exception*, *throwing an exception*, and *catching an exception*, as shown in Figure 12.2.



```
method1() {

  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

FIGURE 12.2 Exception handling in Java consists of declaring exceptions, throwing exceptions, and catching and processing exceptions.

12.4.1 Declaring Exceptions

declare exception

In Java, the statement currently being executed belongs to a method. The Java interpreter invokes the `main` method to start executing a program. Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*. Because system errors and runtime errors can happen to any code, Java does not require that you declare `Error` and `RuntimeException` (unchecked exceptions) explicitly in the method. However, all other exceptions thrown by the method must be explicitly declared in the method header so the caller of the method is informed of the exception.

To declare an exception in a method, use the `throws` keyword in the method header, as in this example:

```
public void myMethod() throws IOException
```

The `throws` keyword indicates `myMethod` might throw an `IOException`. If the method might throw multiple exceptions, add a list of the exceptions, separated by commas, after `throws`:

```
public void myMethod()
  throws Exception1, Exception2, ..., ExceptionN
```

Note

If a method does not declare exceptions in the superclass, you cannot override it to declare exceptions in the subclass.

12.4.2 Throwing Exceptions

throw exception

A program that detects an error can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example: Suppose the program detects that an argument passed to the method violates the method contract (e.g., the argument must be nonnegative, but a negative argument is passed); the program can create an instance of **IllegalArgumentException** and throw it, as follows:

```
IllegalArgumentException ex =
  new IllegalArgumentException("Wrong Argument");
throw ex;
```

Or, if you prefer, you can use the following:

```
throw new IllegalArgumentException("Wrong Argument");
```

Note

IllegalArgumentException is an exception class in the Java API. In general, each exception class in the Java API has at least two constructors: a no-arg constructor and a constructor with a **String** argument that describes the exception. This argument is called the *exception message*, which can be obtained by invoking **getMessage()** from an exception object.

exception message

Tip

The keyword to declare an exception is **throws**, and the keyword to throw an exception is **throw**.

throws vs. throw

12.4.3 Catching Exceptions

You now know how to declare an exception and how to throw an exception. When an exception is thrown, it can be caught and handled in a **try-catch** block, as follows:

catch exception

```
try {
  statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVarN) {
  handler for exceptionN;
}
```

If no exceptions arise during the execution of the **try** block, the **catch** blocks are skipped.

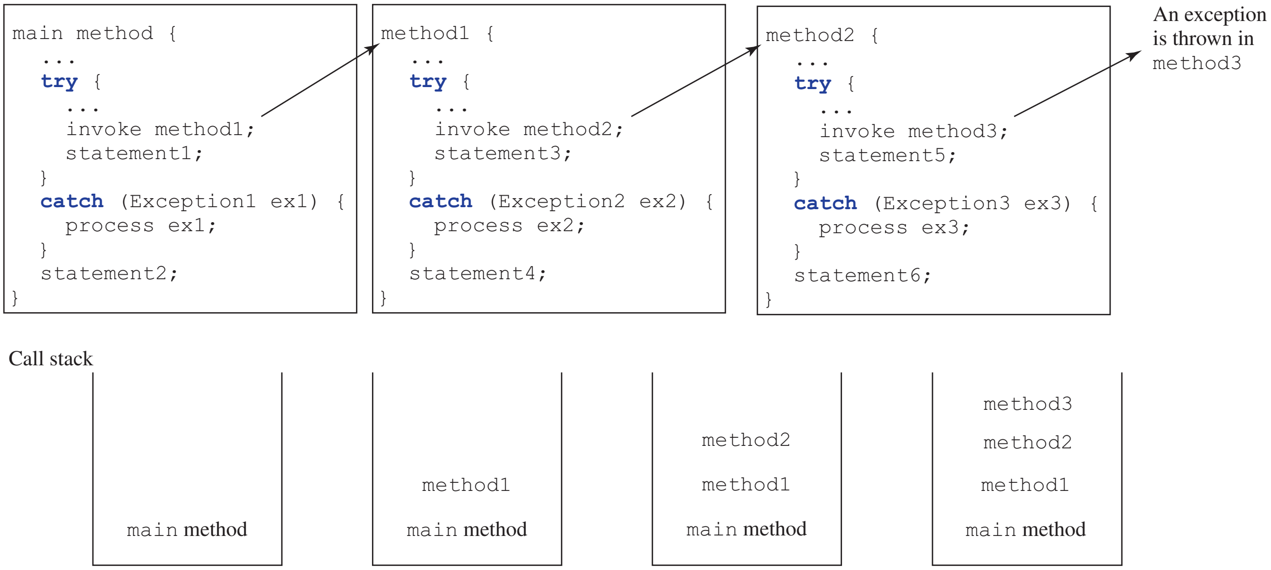
If one of the statements inside the **try** block throws an exception, Java skips the remaining statements in the **try** block and starts the process of finding the code to handle the exception. The code that handles the exception is called the *exception handler*; it is found by *propagating the exception* backward through a chain of method calls, starting from the current method. Each **catch** block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the **catch** block. If so, the exception object is assigned to the variable declared and the code in the **catch** block is executed. If no handler is found, Java exits this method, passes the exception to the method's caller, and continues the same process to find a handler. If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console. The process of finding a handler is called *catching an exception*.

exception handler

exception propagation

Suppose the main method invokes method1, method1 invokes method2, method2 invokes method3, and method3 throws an exception, as shown in Figure 12.3. Consider the following scenario:

- If the exception type is Exception3, it is caught by the catch block for handling exception ex3 in method2. statement5 is skipped and statement6 is executed.
- If the exception type is Exception2, method2 is aborted, the control is returned to method1, and the exception is caught by the catch block for handling exception ex2 in method1. statement3 is skipped and statement4 is executed.
- If the exception type is Exception1, method1 is aborted, the control is returned to the main method, and the exception is caught by the catch block for handling exception ex1 in the main method. statement1 is skipped and statement2 is executed.
- If the exception type is not caught in method2, method1, or main, the program terminates and statement1 and statement2 are not executed.



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    process ex3;
  }
  statement6;
}
```

Figure 12.3 If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the main method.

catch block

Note

Various exception classes can be derived from a common superclass. If a catch block catches exception objects of a superclass, it can catch all the exception objects of the subclasses of that superclass.

order of exception handlers

Note

The order in which exceptions are specified in catch blocks is important. A compile error will result if a catch block for a superclass type appears before a catch block for a subclass type. For example, the ordering in (a) below is erroneous, because RuntimeException is a subclass of Exception. The correct ordering should be as shown in (b).

```
try {
  ...
}
catch (Exception ex) {
  ...
}
catch (RuntimeException ex) {
  ...
}
```

(a) Wrong order

```
try {
  ...
}
catch (RuntimeException ex) {
  ...
}
catch (Exception ex) {
  ...
}
```

(b) Correct order

Note

catch or declare checked exceptions

Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than **Error** or **RuntimeException**), you must invoke it in a **try-catch** block or declare to throw the exception in the calling method. For example, suppose method **p1** invokes method **p2** and **p2** may throw a checked exception (e.g., **IOException**); you have to write the code as shown in (a) or (b) below.

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a) Catch exception

```
void p1() throws IOException {

  p2();

}
```

(b) Throw exception

Note

JDK 7 multicatch

You can use the new JDK 7 multicatch feature to simplify coding for the exceptions with the same handling code. The syntax is:

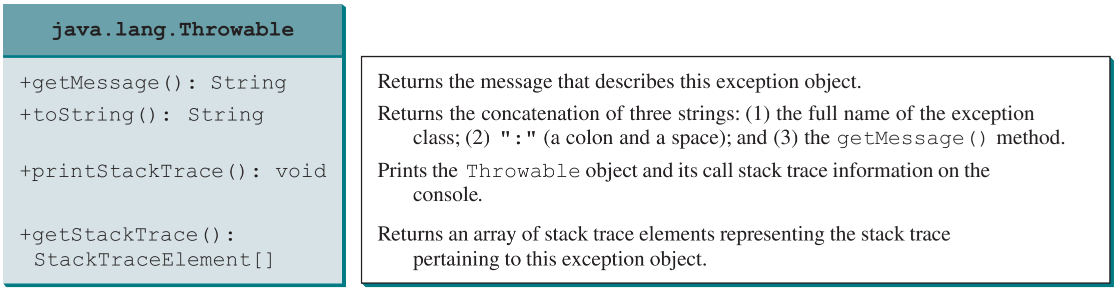
```
catch (Exception1 | Exception2 | ... | Exceptionk ex) {
  // Same code for handling these exceptions
}
```

Each exception type is separated from the next with a vertical bar (**|**). If one of the exceptions is caught, the handling code is executed.

12.4.4 Getting Information from Exceptions

methods in Throwable

An exception object contains valuable information about the exception. You may use the following instance methods in the **java.lang.Throwable** class to get information regarding the exception, as shown in Figure 12.4. The **printStackTrace()** method prints stack trace information on the console. The stack trace lists all the methods in the call stack, which provides



java.lang.Throwable

+getMessage(): String — Returns the message that describes this exception object.

+toString(): String — Returns the concatenation of three strings: (1) the full name of the exception class; (2) ": " (a colon and a space); and (3) the getMessage() method.

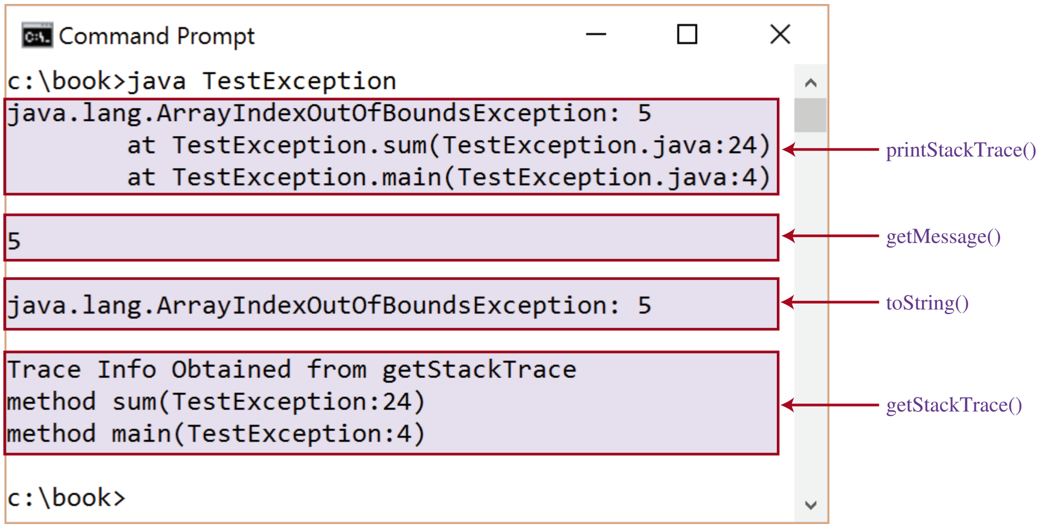
+printStackTrace(): void — Prints the Throwable object and its call stack trace information on the console.

+getStackTrace(): StackTraceElement[] — Returns an array of stack trace elements representing the stack trace pertaining to this exception object.

FIGURE 12.4 **Throwable** is the root class for all exception objects.

valuable information for debugging runtime errors. The `getStackTrace()` method provides programmatic access to the stack trace information printed by `printStackTrace()`.

Listing 12.6 gives an example that uses the methods in `Throwable` to display exception information. Line 4 invokes the `sum` method to return the sum of all the elements in the array. There is an error in line 23 that causes the `ArrayIndexOutOfBoundsException`, a subclass of `IndexOutOfBoundsException`. This exception is caught in the `try-catch` block. Lines 7, 8, and 9 display the stack trace, exception message, and exception object and message using the `printStackTrace()`, `getMessage()`, and `toString()` methods, as shown in Figure 12.5. Line 12 brings stack trace elements into an array. Each element represents a method call. You can obtain the method (line 14), class name (line 15), and exception line number (line 16) for each element.



```
c:\book>java TestException
java.lang.ArrayIndexOutOfBoundsException: 5
        at TestException.sum(TestException.java:24)
        at TestException.main(TestException.java:4)

5

java.lang.ArrayIndexOutOfBoundsException: 5

Trace Info Obtained from getStackTrace
method sum(TestException:24)
method main(TestException:4)

c:\book>
```

Figure 12.5 You can use the `printStackTrace()`, `getMessage()`, `toString()`, and `getStackTrace()` methods to obtain information from exception objects.

Listing 12.6 TestException.java

```
 1 public class TestException  {
 2   public static void main(String[] args) {
 3     try {
 4       System.out.println(sum(new int[] {1, 2, 3, 4, 5}));
 5     }
 6     catch (Exception ex) {
 7       ex.printStackTrace();
 8       System.out.println("\n" + ex.getMessage());
 9       System.out.println("\n" + ex.toString());
10
11       System.out.println("\nTrace Info Obtained from getStackTrace");
12       StackTraceElement[] traceElements = ex.getStackTrace();
13       for (int i = 0; i < traceElements.length; i++) {
14         System.out.print("method " + traceElements[i].getMethodName());
15         System.out.print("(" + traceElements[i].getClassName() + ":");
16         System.out.println(traceElements[i].getLineNumber() + ")");
17       }
18     }
19   }
20
21   private static int sum(int[] list) {
22     int result = 0;
23     for (int i = 0; i <= list.length; i++)
```

invoke sum

printStackTrace()
getMessage()
toString()

getStackTrace()

cause an exception

```
      result += list[i];
    return result;
  }
}
```

12.4.5 Example: Declaring, Throwing, and Catching Exceptions

This example demonstrates declaring, throwing, and catching exceptions by modifying the **setRadius** method in the **Circle** class in Listing 9.8, Circle.java (CircleWithPrivateDataField). The new **setRadius** method throws an exception if the radius is negative.

Listing 12.7 defines a new circle class named **CircleWithException**, which is the same as **Circle** in Listing 9.8 except that the **setRadius(double newRadius)** method throws an **IllegalArgumentException** if the argument **newRadius** is negative.

Listing 12.7 CircleWithException.java

```
public class CircleWithException {
  /** The radius of the circle */
  private double radius;

  /** The number of the objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public CircleWithException() {
    this(1.0);
  }

  /** Construct a circle with a specified radius */
  public CircleWithException(double newRadius) {
    setRadius(newRadius);
    numberOfObjects++;
  }

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

  /** Set a new radius */
  public void setRadius(double newRadius)
      throws IllegalArgumentException {
    if (newRadius >= 0)
      radius = newRadius;
    else
      throw new IllegalArgumentException(
        "Radius cannot be negative");
  }

  /** Return numberOfObjects */
  public static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
  public double findArea() {
    return radius * radius * 3.14159;
  }
}
```

declare exception

throw exception

A test program that uses the new Circle class is given in Listing 12.8.

Listing 12.8 TestCircleWithException.java

try

catch

```
 1  public class TestCircleWithException {
 2    public static void main(String[] args) {
 3      try {
 4        CircleWithException c1 = new CircleWithException(5);
 5        CircleWithException c2 = new CircleWithException(-5);
 6        CircleWithException c3 = new CircleWithException(0);
 7      }
 8      catch (IllegalArgumentException ex) {
 9        System.out.println(ex);
10      }
11
12      System.out.println("Number of objects created: " +
13        CircleWithException.getNumberOfObjects());
14    }
15  }
```

```
java.lang.IllegalArgumentException: Radius cannot be negative
Number of objects created: 1
```

The original Circle class remains intact except that the class name is changed to CircleWithException, a new constructor CircleWithException(newRadius) is added, and the setRadius method now declares an exception and throws it if the radius is negative.

The setRadius method declares to throw IllegalArgumentException in the method header (lines 25–32 in Listing 12.7 CircleWithException.java). The CircleWithException class would still compile if the throws IllegalArgumentException clause (line 26) were removed from the method declaration, since it is a subclass of RuntimeException and every method can throw RuntimeException (an unchecked exception) regardless of whether it is declared in the method header.

The test program creates three CircleWithException objects—c1, c2, and c3—to test how to handle exceptions. Invoking new CircleWithException(-5) (line 5 in Listing 12.8) causes the setRadius method to be invoked, which throws an IllegalArgumentException, because the radius is negative. In the catch block, the type of the object ex is IllegalArgumentException, which matches the exception object thrown by the setRadius method, so this exception is caught by the catch block.

The exception handler prints a short message, ex.toString() (line 9 in Listing 12.8), about the exception, using System.out.println(ex).

Note that the execution continues in the event of the exception. If the handlers had not caught the exception, the program would have abruptly terminated.

The test program would still compile if the try statement were not used, because the method throws an instance of IllegalArgumentException, a subclass of RuntimeException (an unchecked exception).

Check Point

12.4.1 What is the purpose of declaring exceptions? How do you declare an exception and where? Can you declare multiple exceptions in a method header?

12.4.2 How do you throw an exception? Can you throw multiple exceptions in one throw statement?

12.4.3 What is the keyword throw used for? What is the keyword throws used for?

12.4.4 Suppose statement2 causes an exception in the following try-catch block:

```
try {
  statement1;
  statement2;
  statement3;
}
catch (Exception1 ex1) {
}
catch (Exception2 ex2) {
}

statement4;
```

Answer the following questions:

- Will `statement3` be executed?
- If the exception is not caught, will `statement4` be executed?
- If the exception is caught in the `catch` block, will `statement4` be executed?

12.4.5 What is displayed when running the following program?

```
public class Test {
  public static void main(String[] args) {
    try {
      int[] list = new int[10];
      System.out.println("list[10] is " + list[10]);
    }
    catch (ArithmeticException ex) {
      System.out.println("ArithmeticException");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException");
    }
    catch (Exception ex) {
      System.out.println("Exception");
    }
  }
}
```

12.4.6 What is displayed when running the following program?

```
public class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (ArithmeticException ex) {
     System.out.println("ArithmeticException");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException");
    }
    catch (Exception e) {
      System.out.println("Exception");
    }
  }

  static void method() throws Exception {
     System.out.println(1 / 0);
   }
 }
```

12.4.7 What is displayed when running the following program?

```
public class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in main");
    }
    catch (Exception ex) {
      System.out.println("Exception in main");
    }
  }

  static void method() throws Exception {
    try {
      String s ="abc";
      System.out.println(s.charAt(3));
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in method()");
    }
    catch (Exception ex) {
     System.out.println("Exception in method()");
    }
  }
}
```

12.4.8 What does the method `getMessage()` do?

12.4.9 What does the method `printStackTrace()` do?

12.4.10 Does the presence of a `try-catch` block impose overhead when no exception occurs?

12.4.11 Correct a compile error in the following code:

```
public void m(int value) {
  if (value < 40)
    throw new Exception("value is too small");
}
```

12.5 The `finally` Clause

Key Point

The `finally` clause is always executed regardless of whether an exception occurred or not.

Occasionally, you may want some code to be executed regardless of whether an exception occurs or is caught. Java has a `finally` clause that can be used to accomplish this objective. The syntax for the `finally` clause might look like this:

```
try {
  statements;
}
catch (TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}
```

The code in the finally block is executed under all circumstances, regardless of whether an exception occurs in the try block or is caught. Consider three possible cases:

1. If no exception arises in the try block, finalStatements is executed and the next statement after the try statement is executed.
2. If a statement causes an exception in the try block that is caught in a catch block, the rest of the statements in the try block are skipped, the catch block is executed, and the finally clause is executed. The next statement after the try statement is executed.
3. If one of the statements causes an exception that is not caught in any catch block, the other statements in the try block are skipped, the finally clause is executed, and the exception is passed to the caller of this method.

The code in the finally clause is often for closing files and for cleaning up resources. The finally block executes even if there is a return statement prior to reaching the finally block.

Note

The catch block may be omitted when the finally clause is used, as shown in the following code:

omit catch block

```
try {
  code may throw a non-checked exception; regardless of whether an
  exception occurs, finalStatements are executed.
}
finally {
  finalStatements;
}
```

12.5.1 Suppose you run the following code:

Check Point

```
public static void main(String[] args) throws Exception2 {
  m();
  statement7;
}

public static void m() {
  try {
    statement1;
    statement2;
    statement3;
  }
  catch (Exception1 ex1) {
    statement4;
  }
  finally {
    statement5;
  }
  statement6;
}
```

Answer the following questions:

a. If no exception occurs, which statements are executed?

b. If statement2 throws an exception of type Exception1, which statements are executed?

c. If statement2 throws an exception of type Exception2, which statements are executed?

d. If statement2 throws an exception that is neither Exception1 nor Exception2, which statements are executed?

12.6 When to Use Exceptions

A method should throw an exception if the error needs to be handled by its caller.

The `try` block contains the code that is executed in normal circumstances. The `catch` block contains the code that is executed in exceptional circumstances. Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify. Be aware, however, that exception handling usually requires more time and resources, because it requires instantiating a new exception object, rolling back the call stack, and propagating the exception through the chain of method calls to search for the handler.

An exception occurs in a method. If you want the exception to be processed by its caller, you should create an exception object and throw it. If you can handle the exception in the method where it occurs, there is no need to throw or use exceptions.

In general, common exceptions that may occur in multiple classes in a project are candidates for exception classes. Simple errors that may occur in individual methods are best handled without throwing exceptions. This can be done by using `if` statements to check for errors.

When should you use a `try-catch` block in the code? Use it when you have to deal with unexpected error conditions. Do not use a `try-catch` block to deal with simple, expected situations. For example, the following code:

```
try {
  System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```

is better replaced by

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```

Which situations are exceptional and which are expected is sometimes difficult to decide. The point is not to abuse exception handling as a way to deal with a simple logic test.

12.6.1 The following method checks whether a string is a numeric string:

```
public static boolean isNumeric(String token) {
  try {
    Double.parseDouble(token);
    return true;
  }
  catch (java.lang.NumberFormatException ex) {
    return false;
  }
}
```

Is it correct? Rewrite it without using exceptions.

12.7 Rethrowing Exceptions

Java allows an exception handler to rethrow the exception if the handler cannot process the exception, or simply wants to let its caller be notified of the exception.

The syntax for rethrowing an exception may look like this:

```
try {
  statements;
  }
  catch (TheException ex) {
    perform operations before exits;
    throw ex;
  }
```

The statement **throw ex** rethrows the exception to the caller so other handlers in the caller get a chance to process the exception **ex**.

12.7.1 Suppose that **statement2** may cause an exception in the following code:

```
try {
  statement1;
  statement2;
  statement3;
}
catch (Exception1 ex1) {
}
catch (Exception2 ex2) {
  throw ex2;
}
finally {
  statement4;
}
statement5;
```

Answer the following questions:

a. If no exception occurs, will **statement4** or **statement5** be executed?

b. If the exception is of type **Exception1**, will **statement4** or **statement5** be executed?

c. If the exception is of type **Exception2**, will **statement4** or **statement5** be executed?

d. If the exception is not **Exception1** nor **Exception2**, will **statement4** or **statement5** be executed?

12.8 Chained Exceptions

Throwing an exception along with another exception forms a chained exception.

In the preceding section, the **catch** block rethrows the original exception. Sometimes, you may need to throw a new exception (with additional information) along with the original exception. This is called *chained exceptions*. Listing 12.9 illustrates how to create and throw chained exceptions.

chained exception

Listing 12.9 ChainedExceptionDemo.java

```
public class ChainedExceptionDemo {
  public static void main(String[] args) {
    try {
      method1();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static void method1() throws Exception {
    try {
      method2();
```

stack trace

chained exception

throw exception

```
14       }
15       catch (Exception ex) {
16         throw new Exception("New info from method1", ex);
17       }
18     }
19
20     public static void method2() throws Exception {
21       throw new Exception("New info from method2");
22     }
23   }
```

```
java.lang.Exception: New info from method1
  at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:16)
  at ChainedExceptionDemo.main(ChainedExceptionDemo.java:4)
Caused by: java.lang.Exception: New info from method2
  at ChainedExceptionDemo.method2(ChainedExceptionDemo.java:21)
  at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:13)
  ... 1 more
```

The `main` method invokes `method1` (line 4), `method1` invokes `method2` (line 13), and `method2` throws an exception (line 21). This exception is caught in the `catch` block in `method1` and is wrapped in a new exception in line 16. The new exception is thrown and caught in the catch block in the `main` method in line 6. The sample output shows the output from the `printStackTrace()` method in line 7. The new exception thrown from `method1` is displayed first, followed by the original exception thrown from `method2`.

Check Point

12.8.1 What would be the output if line 16 of Listing 12.9 is replaced by the following line?

```
throw new Exception("New info from method1");
```

12.9 Defining Custom Exception Classes

Key Point

You can define a custom exception class by extending the `java.lang.Exception` class.

VideoNote
Create custom exception classes

Java provides quite a few exception classes. Use them whenever possible instead of defining your own exception classes. However, if you run into a problem that cannot be adequately described by the predefined exception classes, you can create your own exception class, derived from `Exception` or from a subclass of `Exception`, such as `IOException`.

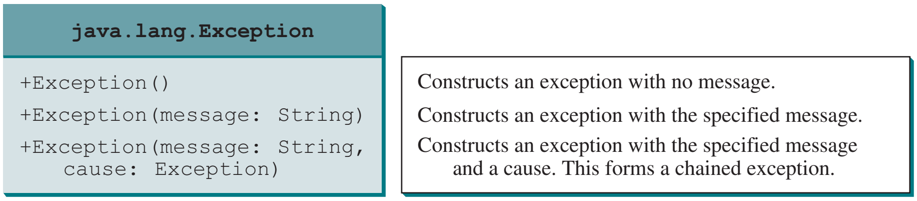
In Listing 12.7, CircleWithException.java, the `setRadius` method throws an exception if the radius is negative. Suppose you wish to pass the radius to the handler. In that case, you can define a custom exception class, as shown in Listing 12.10.

LISTING 12.10 InvalidRadiusException.java

extends `Exception`

```
public class InvalidRadiusException extends Exception {
  private double radius;

  /** Construct an exception */
  public InvalidRadiusException(double radius) {
    super("Invalid radius " + radius);
    this.radius = radius;
  }

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

This custom exception class extends `java.lang.Exception` (line 1). The `Exception` class extends `java.lang.Throwable`. All the methods (e.g., `getMessage()`, `toString()`, and `printStackTrace()`) in `Exception` are inherited from `Throwable`. The `Exception` class contains four constructors. Among them, the following constructors are often used:



```
java.lang.Exception

+Exception()
+Exception(message: String)
+Exception(message: String,
   cause: Exception)
```

Constructs an exception with no message.

Constructs an exception with the specified message.

Constructs an exception with the specified message and a cause. This forms a chained exception.

Line 6 invokes the superclass's constructor with a message. This message will be set in the exception object and can be obtained by invoking `getMessage()` on the object.

Tip

Most exception classes in the Java API contain two constructors: a no-arg constructor and a constructor with a message parameter.

To create an `InvalidRadiusException`, you have to pass a radius. Therefore, the `setRadius` method in Listing 12.7 can be modified as shown in Listing 12.11.

Listing 12.11 TestCircleWithCustomException.java

```java
public class TestCircleWithCustomException {
  public static void main(String[] args) {
    try {
      new CircleWithCustomException(5);
      new CircleWithCustomException(-5);
      new CircleWithCustomException(0);
    }
    catch (InvalidRadiusException ex) {
      System.out.println(ex);
    }

    System.out.println("Number of objects created: " +
      CircleWithCustomException.getNumberOfObjects());
  }
}

class CircleWithCustomException {
  /** The radius of the circle */
  private double radius;

  /** The number of objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public CircleWithCustomException() throws InvalidRadiusException {
    this(1.0);
  }

  /** Construct a circle with a specified radius */
  public CircleWithCustomException(double newRadius)
      throws InvalidRadiusException {
    setRadius(newRadius);
    numberOfObjects++;
  }

```

declare exception

throw exception

```
36    /** Return radius */
37    public double getRadius() {
38      return radius;
39    }
40
41    /** Set a new radius */
42    public void setRadius(double newRadius)
43        throws InvalidRadiusException {
44      if (newRadius >= 0)
45        radius = newRadius;
46      else
47        throw new InvalidRadiusException(newRadius);
48    }
49
50    /** Return numberOfObjects */
51    public static int getNumberOfObjects() {
52      return numberOfObjects;
53    }
54
55    /** Return the area of this circle */
56    public double findArea() {
57      return radius * radius * 3.14159;
58    }
59  }
```

```
InvalidRadiusException: Invalid radius -5.0
Number of objects created: 1
```

The setRadius method in CircleWithCustomException throws an InvalidRadiusException when radius is negative (line 47). Since InvalidRadiusException is a checked exception, the setRadius method must declare it in the method header (line 43). Since the constructors for CircleWithCustomException invoke the setRadius method to set a new radius, and it may throw an InvalidRadiusException, the constructors are declared to throw InvalidRadiusException (lines 25 and 31).

Invoking new CircleWithCustomException(-5) (line 5) throws an InvalidRadiusException, which is caught by the handler. The handler displays the radius in the exception object ex.

checked custom exception

Tip

Can you define a custom exception class by extending RuntimeException? Yes, but it is not a good way to go because it makes your custom exception unchecked. It is better to make a custom exception checked, so the compiler can force these exceptions to be caught in your program.

Check Point

12.9.1 How do you define a custom exception class?

12.9.2 Suppose that the setRadius method throws the InvalidRadiusException defined in Listing 12.10. What is displayed when running the following program?

```
public class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in main");
    }
    catch (Exception ex) {
      System.out.println("Exception in main");
    }
  }

  static void method() throws Exception {
    try {
      Circle c1 = new Circle(1);
      c1.setRadius(-1);
      System.out.println(c1.getRadius());
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in method()");
    }
    catch (Exception ex) {
      System.out.println("Exception in method()");
      throw ex;
    }
  }
}
```

12.10 The File Class

The File class contains the methods for obtaining the properties of a file/directory, and for renaming and deleting a file/directory.

Key Point

why file?

Having learned exception handling, you are ready to step into file processing. Data stored in the program are temporary; they are lost when the program terminates. To permanently store the data created in a program, you need to save them in a file on a disk or other permanent storage device. The file can then be transported and read later by other programs. Since data are stored in files, this section introduces how to use the File class to obtain file/directory properties, to delete and rename files/directories, and to create directories. The next section introduces how to read/write data from/to text files.

absolute file name

directory path

Every file is placed in a directory in the file system. An *absolute file name* (or *full name*) contains a file name with its complete path and drive letter. For example, **c:\book\Welcome.java** is the absolute file name for the file **Welcome.java** on the Windows operating system. Here, **c:\book** is referred to as the *directory path* for the file. Absolute file names are machine dependent. On the UNIX platform, the absolute file name may be **/home/liang/book/Welcome.java**, where **/home/liang/book** is the directory path for the file **Welcome.java**.

relative file name

A *relative file name* is in relation to the current working directory. The complete directory path for a relative file name is omitted. For example, **Welcome.java** is a relative file name. If the current working directory is **c:\book**, the absolute file name would be **c:\book\Welcome.java**.

The File class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion. The File class contains the methods for obtaining file and directory properties, and for renaming and deleting files and directories, as shown in Figure 12.6. However, *the File class does not contain the methods for reading and writing file contents*.

The file name is a string. The File class is a wrapper class for the file name and its directory path. For example, new File("c:\\book") creates a File object for the directory **c:\book** and new File("c:\\book\\test.dat") creates a File object for the file **c:\book\test.dat**, both on Windows. You can use the File class's isDirectory() method to check whether the object represents a directory, and the isFile() method to check whether the object represents a file.

java.io.File	
+File(pathname: String)	Creates a File object for the specified path name. The path name may be a directory or a file.
+File(parent: String, child: String)	Creates a File object for the child under the directory parent. The child may be a file name or a subdirectory.
+File(parent: File, child: String)	Creates a File object for the child under the directory parent. The parent is a File object. In the preceding constructor, the parent is a string.
+exists(): boolean	Returns true if the file or the directory represented by the File object exists.
+canRead(): boolean	Returns true if the file represented by the File object exists and can be read.
+canWrite(): boolean	Returns true if the file represented by the File object exists and can be written.
+isDirectory(): boolean	Returns true if the File object represents a directory.
+isFile(): boolean	Returns true if the File object represents a file.
+isAbsolute(): boolean	Returns true if the File object is created using an absolute path name.
+isHidden(): boolean	Returns true if the file represented in the File object is hidden. The exact definition of *hidden* is system dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period (.) character.
+getAbsolutePath(): String	Returns the complete absolute file or directory name represented by the File object.
+getCanonicalPath(): String	Returns the same as getAbsolutePath() except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
+getName(): String	Returns the last name of the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getName() returns test.dat.
+getPath(): String	Returns the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getPath() returns c:\book\test.dat.
+getParent(): String	Returns the complete parent directory of the current directory or the file represented by the File object. For example, new File("c:\\book\\test.dat").getParent() returns c:\book.
+lastModified(): long	Returns the time that the file was last modified.
+length(): long	Returns the size of the file, or 0 if it does not exist or if it is a directory.
+listFile(): File[]	Returns the files under the directory for a directory File object.
+delete(): boolean	Deletes the file or directory represented by this File object. The method returns true if the deletion succeeds.
+renameTo(dest: File): boolean	Renames the file or directory represented by this File object to the specified name represented in dest. The method returns true if the operation succeeds.
+mkdir(): boolean	Creates a directory represented in this File object. Returns true if the the directory is created successfully.
+mkdirs(): boolean	Same as mkdir() except that it creates directory along with its parent directories if the parent directories do not exist.

FIGURE 12.6 The **File** class can be used to obtain file and directory properties, to delete and rename files and directories, and to create directories.

\ in file names

Caution

The directory separator for Windows is a backslash (****). The backslash is a special character in Java and should be written as \\ in a string literal (see Table 4.5).

Note

*Constructing a **File** instance does not create a file on the machine*. You can create a **File** instance for any file name regardless of whether it exists or not. You can invoke the **exists()** method on a **File** instance to check whether the file exists.

relative file name

Java directory separator (/)

Do not use absolute file names in your program. If you use a file name such as **c:\\book\\Welcome.java**, it will work on Windows but not on other platforms. You should use a file name relative to the current directory. For example, you may create a **File** object using **new File("Welcome.java")** for the file **Welcome.java** in the current directory. You may create a **File** object using **new File("image/us.gif")** for the file **us.gif** under the **image** directory in the current directory. The forward slash (**/**) is the Java directory separator, which is the same as on UNIX. The statement `new File("image/us.gif")` works on Windows, UNIX, and any other platform.

Listing 12.12 demonstrates how to create a `File` object and use the methods in the `File` class to obtain its properties. The program creates a `File` object for the file **us.gif**. This file is stored under the **image** directory in the current directory.

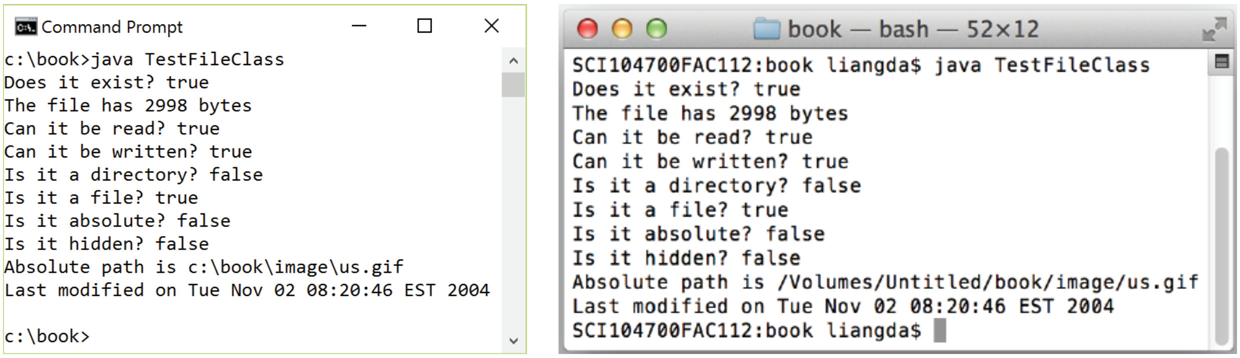
Listing 12.12 TestFileClass.java

```java
 1  public class TestFileClass {
 2    public static void main(String[] args) {
 3      java.io.File file = new java.io.File("image/us.gif");
 4      System.out.println("Does it exist? " + file.exists());
 5      System.out.println("The file has " + file.length() + " bytes");
 6      System.out.println("Can it be read? " + file.canRead());
 7      System.out.println("Can it be written? " + file.canWrite());
 8      System.out.println("Is it a directory? " + file.isDirectory());
 9      System.out.println("Is it a file? " + file.isFile());
10      System.out.println("Is it absolute? " + file.isAbsolute());
11      System.out.println("Is it hidden? " + file.isHidden());
12      System.out.println("Absolute path is " +
13        file.getAbsolutePath());
14      System.out.println("Last modified on " +
15        new java.util.Date(file.lastModified()));
16    }
17  }
```

create a `File`
`exists()`
`length()`
`canRead()`
`canWrite()`
`isDirectory()`
`isFile()`
`isAbsolute()`
`isHidden()`
`getAbsolutePath()`
`lastModified()`

The `lastModified()` method returns the date and time when the file was last modified, measured in milliseconds since the beginning of UNIX time (00:00:00 GMT, January 1, 1970). The `Date` class is used to display it in a readable format in lines 14 and 15.

Figure 12.7a shows a sample run of the program on Windows and Figure 12.7b, a sample run on UNIX. As shown in the figures, the path-naming conventions on Windows are different from those on UNIX.



```
c:\book>java TestFileClass
Does it exist? true
The file has 2998 bytes
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
Absolute path is c:\book\image\us.gif
Last modified on Tue Nov 02 08:20:46 EST 2004

c:\book>
```

(a) On Windows

```
SCI104700FAC112:book liangda$ java TestFileClass
Does it exist? true
The file has 2998 bytes
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
Absolute path is /Volumes/Untitled/book/image/us.gif
Last modified on Tue Nov 02 08:20:46 EST 2004
SCI104700FAC112:book liangda$
```

(b) On UNIX

Figure 12.7 The program creates a `File` object and displays file properties.

Check Point

12.10.1 What is wrong about creating a `File` object using the following statement?
`new File("c:\book\test.dat");`

12.10.2 How do you check whether a file already exists? How do you delete a file? How do you rename a file? Can you find the file size (the number of bytes) using the `File` class? How do you create a directory?

12.10.3 Can you use the `File` class for I/O? Does creating a `File` object create a file on the disk?

12.11 File Input and Output

Key Point

*Use the **Scanner** class for reading text data from a file, and the **PrintWriter** class for writing text data to a file.*

VideoNote
Write and read data

A `File` object encapsulates the properties of a file or a path, but it does not contain the methods for writing/reading data to/from a file (referred to as data *input* and *output*, or *I/O* for short). In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file. There are two types of files: text and binary. Text files are essentially characters on disk. This section introduces how to read/write strings and numeric values from/to a text file using the `Scanner` and `PrintWriter` classes. Binary files will be introduced in Chapter 17.

12.11.1 Writing Data Using PrintWriter

The `java.io.PrintWriter` class can be used to create a file and write data to a text file. First, you have to create a `PrintWriter` object for a text file as follows:

```
PrintWriter output = new PrintWriter(filename);
```

Then, you can invoke the `print`, `println`, and `printf` methods on the `PrintWriter` object to write data to a file. Figure 12.8 summarizes frequently used methods in `PrintWriter`.

java.io.PrintWriter	
+PrintWriter(file: File)	Creates a PrintWriter object for the specified file object.
+PrintWriter(filename: String)	Creates a PrintWriter object for the specified file name string.
+print(s: String): void	Writes a string to the file.
+print(c: char): void	Writes a character to the file.
+print(cArray: char[]): void	Writes an array of characters to the file.
+print(i: int): void	Writes an int value to the file.
+print(l: long): void	Writes a long value to the file.
+print(f: float): void	Writes a float value to the file.
+print(d: double): void	Writes a double value to the file.
+print(b: boolean): void	Writes a boolean value to the file.
Also contains the overloaded println methods.	A println method acts like a print method; additionally, it prints a line separator. The line-separator string is defined by the system. It is \r\n on Windows and \n on Unix.
Also contains the overloaded printf methods.	The printf method was introduced in §4.6, "Formatting Console Output."

FIGURE 12.8 The `PrintWriter` class contains the methods for writing data to a text file.

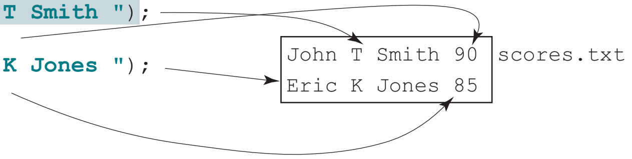
Listing 12.13 gives an example that creates an instance of `PrintWriter` and writes two lines to the file **scores.txt**. Each line consists of a first name (a string), a middle-name initial (a character), a last name (a string), and a score (an integer).

LISTING 12.13 WriteData.java

throws an exception
create File object
file exist?

```
public class WriteData {
  public static void main(String[] args) throws java.io.IOException {
    java.io.File file = new java.io.File("scores.txt");
    if (file.exists()) {
      System.out.println("File already exists");
      System.exit(1);
    }

```

```
 9     // Create a file
10     java.io.PrintWriter output = new java.io.PrintWriter(file);
11
12     // Write formatted output to the file
13     output.print("John T Smith ");
14     output.println(90);
15     output.print("Eric K Jones ");
16     output.println(85);
17
18     // Close the file
19     output.close();
20   }
21 }
```



create PrintWriter

print data

close file

Lines 4–7 check whether the file **scores.txt** exists. If so, exit the program (line 6).

create a file

Invoking the constructor of `PrintWriter` will create a new file if the file does not exist. If the file already exists, the current content in the file will be discarded without verifying with the user.

throws IOException

Invoking the constructor of `PrintWriter` may throw an I/O exception. Java forces you to write the code to deal with this type of exception. For simplicity, we declare `throws IOException` in the main method header (line 2).

print method

You have used the `System.out.print`, `System.out.println`, and `System.out.printf` methods to write text to the console output. `System.out` is a standard Java object for the console. You can create `PrintWriter` objects for writing text to any file using `print`, `println`, and `printf` (lines 13–16).

close file

The `close()` method must be used to close the file (line 19). If this method is not invoked, the data may not be saved properly in the file.

Note

You can append data to an existing file using new PrintWriter(new FileOutputStream(file, true)) to create a PrintWriter object. FileOutputStream will be introduced in Chapter 17.

Tip

When the program writes data to a file, it first stores the data temporarily in a buffer in the memory. When the buffer is full, the data are automatically saved to the file on the disk. Once you close the file, all the data left in the buffer are saved to the file on the disk. Therefore, you must close the file to ensure that all data are saved to the file.

12.11.2 Closing Resources Automatically Using try-with-resources

Programmers often forget to close the file. JDK 7 provides the following try-with-resources syntax that automatically closes the files.

```
try (declare and create resources) {
  Use the resource to process the file;
}
```

Using the try-with-resources syntax, we rewrite the code in Listing 12.13 as shown in Listing 12.14.

Listing 12.14 WriteDataWithAutoClose.java

```
public class WriteDataWithAutoClose {
  public static void main(String[] args) throws Exception {
    java.io.File file = new java.io.File("scores.txt");
    if (file.exists()) {
      System.out.println("File already exists");
      System.exit(0);
```

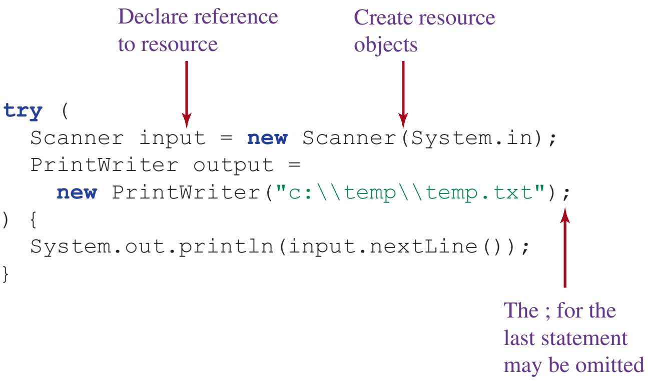
declare/create resource

use the resource

```
 7      }
 8
 9      try (
10        // Create a file
11        java.io.PrintWriter output = new java.io.PrintWriter(file);
12      ) {
13        // Write formatted output to the file
14        output.print("John T Smith ");
15        output.println(90);
16        output.print("Eric K Jones ");
17        output.println(85);
18      }
19    }
20  }
```

A resource is declared and created in the parentheses following the keyword **try**. The resources must be a subtype of **AutoCloseable** such as a **PrinterWriter** that has the **close()** method. A resource must be declared and created in the same statement, and multiple resources can be declared and created inside the parentheses. The statements in the block (lines 12–18) immediately following the resource declaration use the resource. After the block is finished, the resource's **close()** method is automatically invoked to close the resource. Using try-with-resources can not only avoid errors, but also make the code simpler. Note the catch clause may be omitted in a try-with-resources statement.

Note that (1) you have to declare the resource reference variable and create the resource altogether in the **try(...)** clause; (2) the semicolon (**;**) in last statement in the **try(...)** clause may be omitted; (3) You may create multiple **AutoCloseable** resources in the the **try(...)** clause; (4) The **try(...)** clause can contain only the statements for creating resources. Here is an example.



```
try (
  Scanner input = new Scanner(System.in);
  PrintWriter output =
    new PrintWriter("c:\\temp\\temp.txt");
) {
  System.out.println(input.nextLine());
}
```

Declare reference to resource
Create resource objects
The ; for the last statement may be omitted

12.11.3 Reading Data Using Scanner

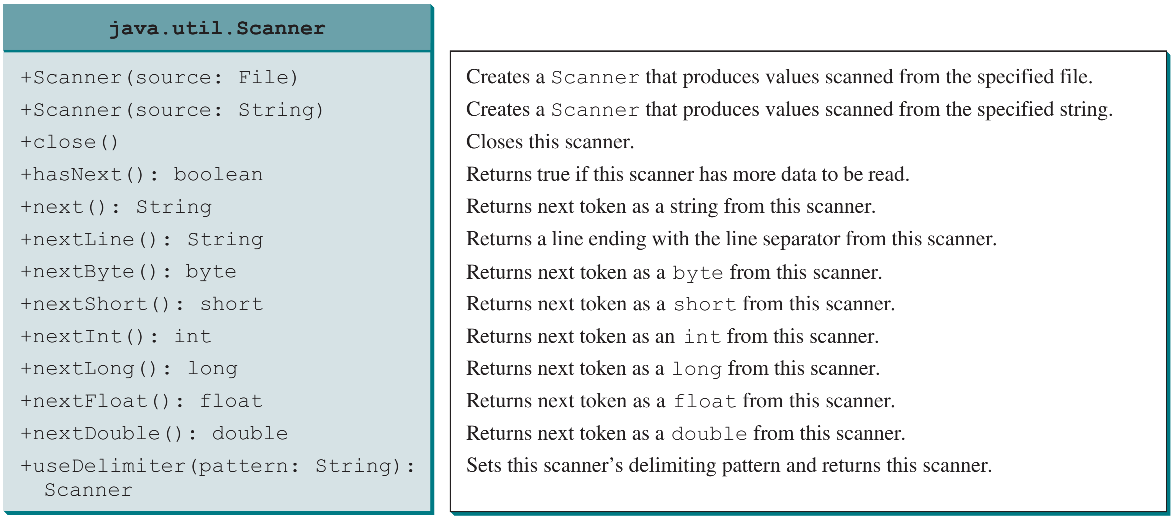
The **java.util.Scanner** class was used to read strings and primitive values from the console in Section 2.3, Reading Input from the Console. A **Scanner** breaks its input into tokens delimited by whitespace characters. To read from the keyboard, you create a **Scanner** for **System.in**, as follows:

```
Scanner input = new Scanner(System.in);
```

To read from a file, create a **Scanner** for a file, as follows:

```
Scanner input = new Scanner(new File(filename));
```

Figure 12.9 summarizes frequently used methods in **Scanner**.



java.util.Scanner	
+Scanner(source: File)	Creates a Scanner that produces values scanned from the specified file.
+Scanner(source: String)	Creates a Scanner that produces values scanned from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has more data to be read.
+next(): String	Returns next token as a string from this scanner.
+nextLine(): String	Returns a line ending with the line separator from this scanner.
+nextByte(): byte	Returns next token as a byte from this scanner.
+nextShort(): short	Returns next token as a short from this scanner.
+nextInt(): int	Returns next token as an int from this scanner.
+nextLong(): long	Returns next token as a long from this scanner.
+nextFloat(): float	Returns next token as a float from this scanner.
+nextDouble(): double	Returns next token as a double from this scanner.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern and returns this scanner.

FIGURE 12.9 The Scanner class contains the methods for scanning data.

Listing 12.15 gives an example that creates an instance of Scanner and reads data from the file **scores.txt**.

LISTING 12.15 ReadData.java

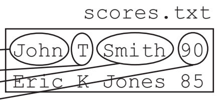
```
 1 import java.util.Scanner;
 2
 3 public class ReadData {
 4   public static void main(String[] args) throws Exception {
 5     // Create a File instance
 6     java.io.File file = new java.io.File("scores.txt");
 7
 8     // Create a Scanner for the file
 9     Scanner input = new Scanner(file);
10
11     // Read data from a file
12     while (input.hasNext()) {
13       String firstName = input.next();
14       String mi = input.next();
15       String lastName = input.next();
16       int score = input.nextInt();
17       System.out.println(
18         firstName +" " + mi + " " + lastName + " " + score);
19     }
20
21     // Close the file
22     input.close();
23   }
24 }
```

create a File

create a Scanner

has next?
read items

close file



Note new Scanner(String) creates a Scanner for a given string. To create a Scanner to read data from a file, you have to use the java.io.File class to create an instance of the File using the constructor new File(filename) (line 6) and use new Scanner(File) to create a Scanner for the file (line 9).

File class

Invoking the constructor new Scanner(File) may throw an I/O exception, so the main method declares throws Exception in line 4.

throws Exception

Each iteration in the while loop reads the first name, middle initial, last name, and score from the text file (lines 12–19). The file is closed in line 22.

close file

It is not necessary to close the input file (line 22), but it is a good practice to do so to release the resources occupied by the file. You can rewrite this program using the try-with-resources syntax. See liveexample.pearsoncmg.com/html/ReadDataWithAutoClose.html.

12.11.4 How Does **Scanner** Work?

change delimiter

Section 4.5.5 introduced token-based and line-based input. The token-based input methods `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, and `next()` read input separated by delimiters. By default, the delimiters are whitespace characters. You can use the `useDelimiter(String regex)` method to set a new pattern for delimiters.

How does an input method work? A token-based input first skips any delimiters (whitespace characters by default) then reads a token ending at a delimiter. The token is then automatically converted into a value of the `byte`, `short`, `int`, `long`, `float`, or `double` type for `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, and `nextDouble()`, respectively. For the `next()` method, no conversion is performed. If the token does not match the expected type, a runtime exception `java.util.InputMismatchException` will be thrown.

`InputMismatchException`

`next()` vs. `nextLine()`

Both methods `next()` and `nextLine()` read a string. The `next()` method reads a string separated by delimiters and `nextLine()` reads a line ending with a line separator.

line separator

Note

The line-separator string is defined by the system. It is `\r\n` on Windows and `\n` on UNIX. To get the line separator on a particular platform, use

```
String lineSeparator = System.getProperty("line.separator");
```

If you enter input from a keyboard, a line ends with the *Enter* key, which corresponds to the `\n` character.

behavior of `nextLine()`

The token-based input method does not read the delimiter after the token. If the `nextLine()` method is invoked after a token-based input method, this method reads characters that start from this delimiter and end with the line separator. The line separator is read, but it is not part of the string returned by `nextLine()`.

input from file

Suppose a text file named **test.txt** contains a line

```
34 567
```

After the following code is executed,

```
Scanner input = new Scanner(new File("test.txt"));
int intValue = input.nextInt();
String line = input.nextLine();
```

`intValue` contains `34` and `line` contains the characters `' '`, `5`, `6`, and `7`.

What happens if the input is *entered from the keyboard*? Suppose you enter `34`, press the *Enter* key, then enter `567` and press the *Enter* key for the following code:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
String line = input.nextLine();
```

You will get `34` in `intValue` and an empty string in `line`. Why? Here is the reason. The token-based input method `nextInt()` reads in `34` and stops at the delimiter, which in this case is a line separator (the *Enter* key). The `nextLine()` method ends after reading the line separator and returns the string read before the line separator. Since there are no characters before the line separator, `line` is empty. For this reason, *you should not use a line-based input after a token-based input.*

scan a string

You can read data from a file or from the keyboard using the `Scanner` class. You can also scan data from a string using the `Scanner` class. For example, the following code:

```
Scanner input = new Scanner("13 14");
int sum = input.nextInt() + input.nextInt();
System.out.println("Sum is " + sum);
```

displays

```
Sum is 27
```

12.11.5 Case Study: Replacing Text

Suppose you are to write a program named `ReplaceText` that replaces all occurrences of a string in a text file with a new string. The file name and strings are passed as command-line arguments as follows:

```
java ReplaceText sourceFile targetFile oldString newString
```

For example, invoking

```
java ReplaceText FormatString.java t.txt StringBuilder StringBuffer
```

replaces all the occurrences of `StringBuilder` by `StringBuffer` in the file **FormatString.java** and saves the new file in **t.txt**.

Listing 12.16 gives the program. The program checks the number of arguments passed to the `main` method (lines 7–11), checks whether the source and target files exist (lines 14–25), creates a `Scanner` for the source file (line 29), creates a `PrintWriter` for the target file (line 30), and repeatedly reads a line from the source file (line 33), replaces the text (line 34), and writes a new line to the target file (line 35).

LISTING 12.16 `ReplaceText.java`

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class ReplaceText {
 5    public static void main(String[] args) throws Exception {
 6      // Check command line parameter usage
 7      if (args.length != 4) {
 8        System.out.println(
 9          "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
10        System.exit(1);
11      }
12
13      // Check if source file exists
14      File sourceFile = new File(args[0]);
15      if (!sourceFile.exists()) {
16        System.out.println("Source file " + args[0] + " does not exist");
17        System.exit(2);
18      }
19
20      // Check if target file exists
21      File targetFile = new File(args[1]);
22      if (targetFile.exists()) {
23        System.out.println("Target file " + args[1] + " already exists");
24        System.exit(3);
25      }
26
27      try (
28        // Create input and output files
29        Scanner input = new Scanner(sourceFile);
30        PrintWriter output = new PrintWriter(targetFile);
```

check command usage

source file exists?

target file exists?

try-with-resources

create a `Scanner`

create a `PrintWriter`

has next?
read a line

```
31       ) {
32         while (input.hasNext()) {
33           String s1 = input.nextLine();
34           String s2 = s1.replaceAll(args[2], args[3]);
35           output.println(s2);
36         }
37       }
38     }
39   }
```

In a normal situation, the program is terminated after a file is copied. The program is terminated abnormally if the command-line arguments are not used properly (lines 7–11), if the source file does not exist (lines 14–18), or if the target file already exists (lines 22–25). The exit status codes 1, 2, and 3 are used to indicate these abnormal terminations (lines 10, 17, and 24).

Check Point

12.11.1 How do you create a `PrintWriter` to write data to a file? What is the reason to declare `throws Exception` in the main method in Listing 12.13, WriteData.java? What would happen if the `close()` method were not invoked in Listing 12.13?

12.11.2 Show the contents of the file **temp.txt** after the following program is executed:

```
public class Test {
  public static void main(String[] args) throws Exception {
    java.io.PrintWriter output = new
      java.io.PrintWriter("temp.txt");
    output.printf("amount is %f %e\r\n", 32.32, 32.32);
    output.printf("amount is %5.4f %5.4e\r\n", 32.32, 32.32);
    output.printf("%6b\r\n", (1 > 2));
    output.printf("%6s\r\n", "Java");
    output.close();
  }
}
```

12.11.3 Rewrite the code in the preceding question using a try-with-resources syntax.

12.11.4 How do you create a `Scanner` to read data from a file? What is the reason to define `throws Exception` in the main method in Listing 12.15, ReadData.java? What would happen if the `close()` method were not invoked in Listing 12.15?

12.11.5 What will happen if you attempt to create a `Scanner` for a nonexistent file? What will happen if you attempt to create a `PrintWriter` for an existing file?

12.11.6 Is the line separator the same on all platforms? What is the line separator on Windows?

12.11.7 Suppose you enter `45 57.8 789`, then press the *Enter* key. Show the contents of the variables after the following code is executed:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
double doubleValue = input.nextDouble();
String line = input.nextLine();
```

12.11.8 Suppose you enter `45`, press the *Enter* key, enter `57.8`, press the *Enter* key, and enter `789`, press the *Enter* key. Show the contents of the variables after the following code is executed:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
double doubleValue = input.nextDouble();
String line = input.nextLine();
```

12.12 Reading Data from the Web

Just like you can read data from a file on your computer, you can read data from a file on the Web.

Key Point

In addition to reading data from a local file on a computer or file server, you can also access data from a file that is on the Web if you know the file's URL (Uniform Resource Locator—the unique address for a file on the Web). For example, www.google.com/index.html is the URL for the file **index.html** located on the Google web server. When you enter the URL in a Web browser, the Web server sends the data to your browser, which renders the data graphically. Figure 12.10 illustrates how this process works.

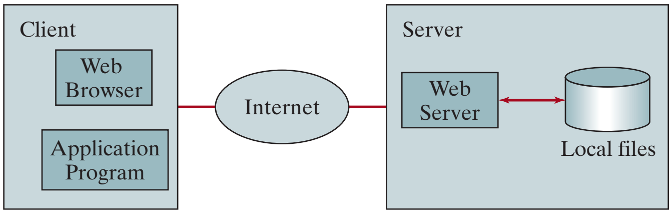


FIGURE 12.10 The client retrieves files from a Web server.

For an application program to read data from a URL, you first need to create a `URL` object using the `java.net.URL` class with this constructor:

```
public URL(String spec) throws MalformedURLException
```

For example, the following statement creates a URL object for http://www.google.com/index.html.

```
1  try {
2    URL url = new URL("http://www.google.com/index.html");
3  }
4  catch (MalformedURLException ex) {
5    ex.printStackTrace();
6  }
```

A `MalformedURLException` is thrown if the URL string has a syntax error. For example, the URL string http:www.google.com/index.html would cause a `MalformedURLException` runtime error because two slashes (`//`) are required after the colon (`:`). Note the `http://` prefix is required for the `URL` class to recognize a valid URL. It would be wrong if you replace line 2 with the following code:

```
URL url = new URL("www.google.com/index.html");
```

After a `URL` object is created, you can use the `openStream()` method defined in the `URL` class to open an input stream and use this stream to create a `Scanner` object as follows:

```
Scanner input = new Scanner(url.openStream());
```

Now you can read the data from the input stream just like from a local file. The example in Listing 12.17 prompts the user to enter a URL and displays the size of the file.

LISTING 12.17 ReadFileFromURL.java

```
import java.util.Scanner;

public class ReadFileFromURL {
  public static void main(String[] args) {
```

enter a URL

create a URL object

create a Scanner object

more to read?

read a line

MalformedURLException

IOException

```
 5      System.out.print("Enter a URL: ");
 6      String URLString = new Scanner(System.in).next();
 7
 8      try {
 9        java.net.URL url = new java.net.URL(URLString);
10        int count = 0;
11        Scanner input = new Scanner(url.openStream());
12        while (input.hasNext()) {
13          String line = input.nextLine();
14          count += line.length();
15        }
16
17        System.out.println("The file size is " + count + " characters");
18      }
19      catch (java.net.MalformedURLException ex) {
20        System.out.println("Invalid URL");
21      }
22      catch (java.io.IOException ex) {
23        System.out.println("I/O Errors: no such file");
24      }
25    }
26  }
```

```
Enter a URL: http://liveexample.pearsoncmg.com/data/Lincoln.txt ↵Enter
The file size is 1469 characters
```

```
Enter a URL: http://www.yahoo.com ↵Enter
The file size is 190006 characters
```

MalformedURLException

The program prompts the user to enter a URL string (line 6) and creates a **URL** object (line 9). The constructor will throw a **java.net.MalformedURLException** (line 19) if the URL isn't formed correctly.

The program creates a **Scanner** object from the input stream for the URL (line 11). If the URL is formed correctly but does not exist, an **IOException** will be thrown (line 22). For example, http://google.com/index1.html uses the appropriate form, but the URL itself does not exist. An **IOException** would be thrown if this URL was used for this program.

Check Point

12.12.1 How do you create a **Scanner** object for reading text from a URL?

12.13 Case Study: Web Crawler

Key Point

This case study develops a program that travels the Web by following hyperlinks.

web crawler

The World Wide web, abbreviated as WWW, W3, or Web, is a system of interlinked hypertext documents on the Internet. With a web browser, you can view a document and follow the hyperlinks to view other documents. In this case study, we will develop a program that automatically traverses the documents on the Web by following the hyperlinks. This type of program is commonly known as a *web crawler*. For simplicity, our program follows the hyperlink that starts with **http://**. Figure 12.11 shows an example of traversing the Web. We start from a Webpage that contains three URLs named **URL1**, **URL2**, and **URL3**. Following **URL1** leads to the page that contains three URLs named **URL11**, **URL12**, and **URL13**. Following **URL2** leads to the page that contains two URLs named **URL21** and **URL22**. Following **URL3** leads to the page that contains four URLs named **URL31**, **URL32**, **URL33**, and **URL34**. Continue to traverse

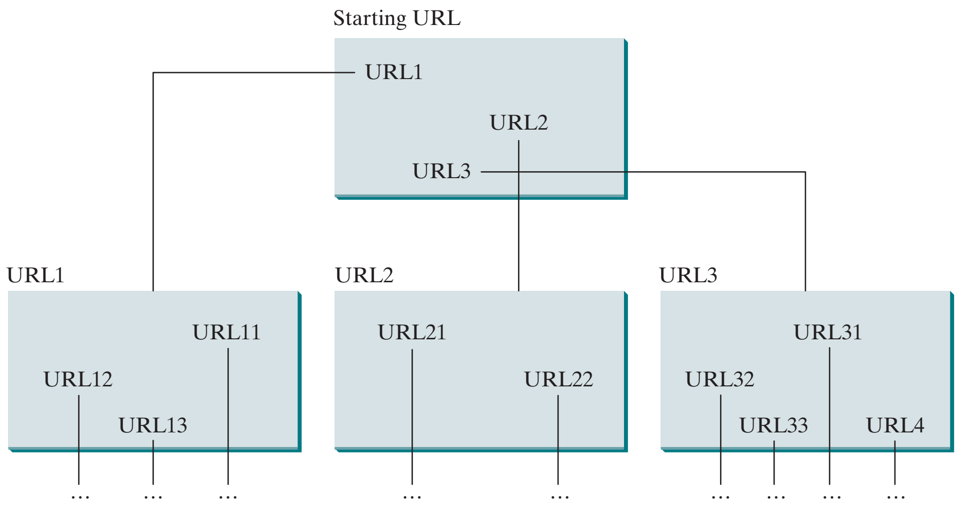


FIGURE 12.11 Web crawler explores the web through hyperlinks.

the Web following the new hyperlinks. As you see, this process may continue forever, but we will exit the program once we have traversed 100 pages.

The program follows the URLs to traverse the Web. To ensure that each URL is traversed only once, the program maintains two lists of URLs. One list stores the URLs pending for traversing, and the other stores the URLs that have already been traversed. The algorithm for this program can be described as follows:

```
Add the starting URL to a list named listOfPendingURLs;
while listOfPendingURLs is not empty and size of listOfTraversedURLs
<= 100 {
  Remove a URL from listOfPendingURLs;
  if this URL is not in listOfTraversedURLs {
    Add it to listOfTraversedURLs;
    Display this URL;
    Read the page from this URL and for each URL contained in the page {
      Add it to listOfPendingURLs if it is not in listOfTraversedURLs;
    }
  }
}
```

Listing 12.18 gives the program that implements this algorithm.

LISTING 12.18 WebCrawler.java

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

public class WebCrawler {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a URL: ");
    String url = input.nextLine();
    crawler(url); // Traverse the Web from the a starting url
  }

  public static void crawler(String startingURL) {
    ArrayList<String> listOfPendingURLs = new ArrayList<>();
    ArrayList<String> listOfTraversedURLs = new ArrayList<>();
```

enter a URL
crawl from this URL
list of pending URLs
list of traversed URLs

add starting URL (line 16)
get the first URL (line 19)
URL traversed (line 21)
add a new URL (line 26)
read a line (line 40)
search for a URL (line 41)
end of a URL (line 42)
URL ends with " (line 44)
extract a URL (line 45)
search for next URL (line 46)
return URLs (line 57)

```
15
16     listOfPendingURLs.add(startingURL);
17     while (!listOfPendingURLs.isEmpty() &&
18         listOfTraversedURLs.size() <= 100) {
19      String urlString = listOfPendingURLs.remove(0);
20      if (!listOfTraversedURLs.contains(urlString)) {
21       listOfTraversedURLs.add(urlString);
22       System.out.println("Crawl " + urlString);
23
24       for (String s: getSubURLs(urlString)) {
25        if (!listOfTraversedURLs.contains(s))
26         listOfPendingURLs.add(s);
27       }
28      }
29     }
30    }
31
32    public static ArrayList<String> getSubURLs(String urlString) {
33     ArrayList<String> list = new ArrayList<>();
34
35     try {
36      java.net.URL url = new java.net.URL(urlString);
37      Scanner input = new Scanner(url.openStream());
38      int current = 0;
39      while (input.hasNext()) {
40       String line = input.nextLine();
41       current = line.indexOf("http:", current);
42       while (current > 0) {
43        int endIndex = line.indexOf("\"", current);
44        if (endIndex > 0) { // Ensure that a correct URL is found
45         list.add(line.substring(current, endIndex));
46         current = line.indexOf("http:", endIndex);
47        }
48        else
49         current = -1;
50       }
51      }
52     }
53     catch (Exception ex) {
54      System.out.println("Error: " + ex.getMessage());
55     }
56
57     return list;
58    }
59   }
```

```
Enter a URL: http://cs.armstrong.edu/liang ↵Enter
Crawl http://www.cs.armstrong.edu/liang
Crawl http://www.cs.armstrong.edu
Crawl http://www.armstrong.edu
Crawl http://www.pearsonhighered.com/liang
...
```

The program prompts the user to enter a starting URL (lines 7 and 8) and invokes the `crawler(url)` method to traverse the Web (line 9).

The `crawler(url)` method adds the starting url to `listOfPendingURLs` (line 16) and repeatedly process each URL in `listOfPendingURLs` in a while loop (lines 17–29). It removes the first URL in the list (line 19) and processes the URL if it has not been processed (lines 20–28).

To process each URL, the program first adds the URL to `listOfTraversedURLs` (line 21). This list stores all the URLs that have been processed. The `getSubURLs(url)` method returns a list of URLs in the webpage for the specified URL (line 24). The program uses a foreach loop to add each URL in the page into `listOfPendingURLs` if it is not in `listOfTraversedURLs` (lines 24–27).

The `getSubURLs(url)` method reads each line from the webpage (line 40) and searches for the URLs in the line (line 41). Note a correct URL cannot contain line break characters. Therefore, it is sufficient to limit the search for a URL in one line of the text in a webpage. For simplicity, we assume that a URL ends with a quotation mark `"` (line 43). The method obtains a URL and adds it to a list (line 45). A line may contain multiple URLs. The method continues to search for the next URL (line 46). If no URL is found in the line, current is set to `-1` (line 49). The URLs contained in the page are returned in the form of a list (line 57).

The program terminates when the number of traversed URLs reaches 100 (line 18).

This is a simple program to traverse the Web. Later, you will learn the techniques to make the program more efficient and robust.

Check Point

12.13.1 Before a URL is added to `listOfPendingURLs`, line 25 checks whether it has been traversed. Is it possible that `listOfPendingURLs` contains duplicate URLs? If so, give an example.

12.13.2 Simplify the code in lines 20-28 as follows: 1. Delete lines 20 and 28; 2. Add an additional condition `!listOfPendingURLs.contains(s)` to the if statement in line 25. Write the complete new code for the while loop in lines 17-29. Does this revision work?

Key Terms

absolute file name	477
chained exception	473
checked exception	461
declare exception	467
directory path	477
exception	491
exception propagation	463
relative file name	477
throw exception	457
unchecked exception	489

Chapter Summary

1. Exception handling enables a method to throw an exception to its caller.

2. A Java *exception* is an instance of a class derived from `java.lang.Throwable`. Java provides a number of predefined exception classes, such as `Error`, `Exception`, `RuntimeException`, `ClassNotFoundException`, `NullPointerException`, and `ArithmeticException`. You can also define your own exception class by extending `Exception`.

3. Exceptions occur during the execution of a method. `RuntimeException` and `Error` are *unchecked exceptions*; all other exceptions are *checked*.

4. When *declaring a method*, you have to declare a checked exception if the method might throw it, thus telling the compiler what can go wrong.

5. The keyword for declaring an exception is `throws`, and the keyword for throwing an exception is `throw`.

6. To invoke the method that declares checked exceptions, enclose it in a `try` statement. When an exception occurs during the execution of the method, the `catch` block catches and handles the exception.

7. If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.

8. Various exception classes can be derived from a common superclass. If a `catch` block catches the exception objects of a superclass, it can also catch all the exception objects of the subclasses of that superclass.

9. The order in which exceptions are specified in a `catch` block is important. A compile error will result if you specify an exception object of a class after an exception object of the superclass of that class.

10. When an exception occurs in a method, the method exits immediately if it does not catch the exception. If the method is required to perform some task before exiting, you can catch the exception in the method and then rethrow it to its caller.

11. The code in the `finally` block is executed under all circumstances, regardless of whether an exception occurs in the `try` block, or whether an exception is caught if it occurs.

12. Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.

13. Exception handling should not be used to replace simple tests. You should perform simple test using `if` statements whenever possible and reserve exception handling for dealing with situations that cannot be handled with `if` statements.

14. The `File` class is used to obtain file properties and manipulate files. It does not contain the methods for creating a file or for reading/writing data from/to a file.

15. You can use `Scanner` to read string and primitive data values from a text file and use `PrintWriter` to create a file and write data to a text file.

16. You can read from a file on the Web using the `URL` class.

Quiz

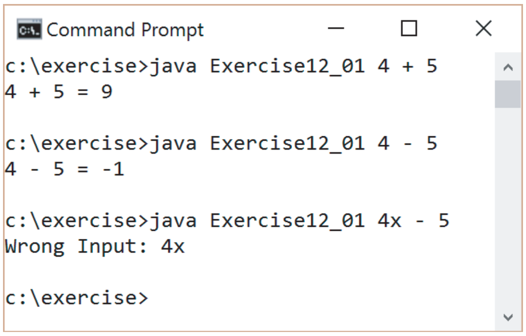
Answer the quiz for this chapter online at the Companion Website.

MyProgrammingLab™

Programming Exercises

Sections 12.2–12.9

***12.1** (*`NumberFormatException`*) Listing 7.9, Calculator.java, is a simple command-line calculator. Note the program terminates if any operand is nonnumeric. Write a program with an exception handler that deals with nonnumeric operands; then write another program without using an exception handler to achieve the same objective. Your program should display a message that informs the user of the wrong operand type before exiting (see Figure 12.12).



```
c:\exercise>java Exercise12_01 4 + 5
4 + 5 = 9

c:\exercise>java Exercise12_01 4 - 5
4 - 5 = -1

c:\exercise>java Exercise12_01 4x - 5
Wrong Input: 4x

c:\exercise>
```

FIGURE 12.12 The program performs arithmetic operations and detects input errors.

***12.2** (*InputMismatchException*) Write a program that prompts the user to read two integers and displays their sum. Your program should prompt the user to read the number again if the input is incorrect.

***12.3** (*ArrayIndexOutOfBoundsException*) Write a program that meets the following requirements:

- Creates an array with 100 randomly chosen integers.
- Prompts the user to enter the index of the array, then displays the corresponding element value. If the specified index is out of bounds, display the message "Out of Bounds".

***12.4** (*IllegalArgumentException*) Modify the Loan class in Listing 10.2 to throw IllegalArgumentException if the loan amount, interest rate, or number of years is less than or equal to zero.

***12.5** (*IllegalTriangleException*) Programming Exercise 11.1 defined the Triangle class with three sides. In a triangle, the sum of any two sides is greater than the other side. The Triangle class must adhere to this rule. Create the IllegalTriangleException class, and modify the constructor of the Triangle class to throw an IllegalTriangleException object if a triangle is created with sides that violate the rule, as follows:

```
/** Construct a triangle with the specified sides */
public Triangle(double side1, double side2, double side3)
  throws IllegalTriangleException {
  // Implement it
}
```

***12.6** (*NumberFormatException*) Listing 6.8 implements the hex2Dec(String hexString) method, which converts a hex string into a decimal number. Implement the hex2Dec method to throw a NumberFormatException if the string is not a hex string. Write a test program that prompts the user to enter a hex number as a string and displays its decimal equivalent. If the method throws an exception, display “Not a hex number”.

***12.7** (*NumberFormatException*) Write the bin2Dec(String binaryString) method to convert a binary string into a decimal number. Implement the bin2Dec method to throw a NumberFormatException if the string is not a binary string. Write a test program that prompts the user to enter a binary number as a string and displays its decimal equivalent. If the method throws an exception, display “Not a binary number”.

***12.8** (*HexFormatException*) Programming Exercise 12.6 implements the hex2Dec method to throw a NumberFormatException if the string is not a hex string. Define a custom exception called HexFormatException.

VideoNote
HexFormatException

Implement the hex2Dec method to throw a HexFormatException if the string is not a hex string.

*12.9 (*BinaryFormatException*) Exercise 12.7 implements the bin2Dec method to throw a BinaryFormatException if the string is not a binary string. Define a custom exception called BinaryFormatException. Implement the bin2Dec method to throw a BinaryFormatException if the string is not a binary string.

*12.10 (*OutOfMemoryError*) Write a program that causes the JVM to throw an OutOfMemoryError and catches and handles this error.

Sections 12.10–12.12

**12.11 (*Remove text*) Write a program that removes all the occurrences of a specified string from a text file. For example, invoking

```
java Exercise12_11 John filename
```

removes the string John from the specified file. Your program should get the arguments from the command line.

**12.12 (*Reformat Java source code*) Write a program that converts the Java source code from the next-line brace style to the end-of-line brace style. For example, the following Java source in (a) uses the next-line brace style. Your program converts it to the end-of-line brace style in (b).

```
public class Test
{
  public static void main(String[] args)
  {
    // Some statements
  }
}
```

(a) Next-line brace style

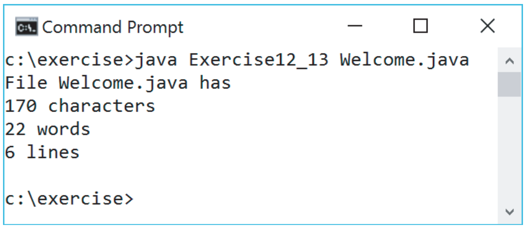
```
public class Test {
  public static void main(String[] args) {
    // Some statements
  }
}
```

(b) End-of-line brace style

Your program can be invoked from the command line with the Java source-code file as the argument. It converts the Java source code to a new format. For example, the following command converts the Java source-code file Test.java to the end-of-line brace style.

```
java Exercise12_12 Test.java
```

*12.13 (*Count characters, words, and lines in a file*) Write a program that will count the number of characters, words, and lines in a file. Words are separated by whitespace characters. The file name should be passed as a command-line argument, as shown in Figure 12.13.



```
c:\exercise>java Exercise12_13 Welcome.java
File Welcome.java has
170 characters
22 words
6 lines

c:\exercise>
```

FIGURE 12.13 The program displays the number of characters, words, and lines in the given file.

*12.14 (*Process scores in a text file*) Suppose a text file contains an unspecified number of scores separated by spaces. Write a program that prompts the user to enter the file, reads the scores from the file, and displays their total and average.

***12.15** (*Write/read data*) Write a program to create a file named **Exercise12_15.txt** if it does not exist. Write `100` integers created randomly into the file using text I/O. Integers are separated by spaces in the file. Read the data back from the file and display the data in increasing order.

****12.16** (*Replace text*) Listing 12.16, ReplaceText.java, gives a program that replaces text in a source file and saves the change into a new file. Revise the program to save the change into the original file. For example, invoking

```
java Exercise12_16 file oldString newString
```

replaces `oldString` in the source file with `newString`.

*****12.17** (*Game: hangman*) Rewrite Programming Exercise 7.35. The program reads the words stored in a text file named **hangman.txt**. Words are delimited by spaces.

****12.18** (*Add package statement*) Suppose you have Java source files under the directories `chapter1`, `chapter2`, . . . , `chapter34`. Write a program to insert the statement `package chapteri;` as the first line for each Java source file under the directory `chapteri`. Suppose `chapter1`, `chapter2`, . . . , `chapter34` are under the root directory `srcRootDirectory`. The root directory and `chapteri` directory may contain other folders and files. Use the following command to run the program:

```
java Exercise12_18 srcRootDirectory
```

***12.19** (*Count words*) Write a program that counts the number of words in President Abraham Lincoln's Gettysburg address from https://liveexample.pearsoncmg.com/data/Lincoln.txt.

****12.20** (*Remove package statement*) Suppose you have Java source files under the directories `chapter1`, `chapter2`, . . . , `chapter34`. Write a program to remove the statement `package chapteri;` in the first line for each Java source file under the directory `chapteri`. Suppose `chapter1`, `chapter2`, . . . , `chapter34` are under the root directory `srcRootDirectory`. The root directory and `chapteri` directory may contain other folders and files. Use the following command to run the program:

```
java Exercise12_20 srcRootDirectory
```

***12.21** (*Data sorted?*) Write a program that reads the strings from file **SortedStrings.txt** and reports whether the strings in the files are stored in increasing order. If the strings are not sorted in the file, it displays the first two strings that are out of the order.

****12.22** (*Replace text*) Revise Programming Exercise 12.16 to replace a string in a file with a new string for all files in the specified directory using the following command:

```
java Exercise12_22 dir oldString newString
```

****12.23** (*Process scores in a text file on the Web*) Suppose the text file on the Web http://liveexample.pearsoncmg.com/data/Scores.txt contains an unspecified number of scores separated by spaces. Write a program that reads the scores from the file and displays their total and average.

***12.24** (*Create large dataset*) Create a data file with 1,000 lines. Each line in the file consists of a faculty member's first name, last name, rank, and salary. The faculty member's first name and last name for the *i*th line are FirstName*i* and LastName*i*. The rank is randomly generated as assistant, associate, and full. The salary is randomly generated as a number with two digits after the decimal point. The salary for an assistant professor should be in the range from 50,000 to 80,000, for associate professor from 60,000 to 110,000, and for full professor from 75,000 to 130,000. Save the file in **Salary.txt**. Here are some sample data:

FirstName1 LastName1 assistant 60055.95

FirstName2 LastName2 associate 81112.45

. . .

FirstName1000 LastName1000 full 92255.21

***12.25** (*Process large dataset*) A university posts its employees' salaries at http://liveexample.pearsoncmg.com/data/Salary.txt. Each line in the file consists of a faculty member's first name, last name, rank, and salary (see Programming Exercise 12.24). Write a program to display the total salary for assistant professors, associate professors, full professors, and faculty, respectively, and display the average salary for assistant professors, associate professors, full professors, and faculty, respectively.

****12.26** (*Create a directory*) Write a program that prompts the user to enter a directory name and creates a directory using the `File`'s `mkdirs` method. The program displays the message "Directory created successfully" if a directory is created or "Directory already exists" if the directory already exists.

****12.27** (*Replace words*) Suppose you have a lot of files in a directory that contain words **Exercise*i_j***, where *i* and *j* are digits. Write a program that pads a 0 before *i* if *i* is a single digit and 0 before *j* if *j* is a single digit. For example, the word **Exercise2_1** in a file will be replaced by **Exercise02_01**. In Java, when you pass the symbol `*` from the command line, it refers to all files in the directory (see Supplement III.V). Use the following command to run your program:

```
java Exercise12_27 *
```

****12.28** (*Rename files*) Suppose you have a lot of files in a directory named **Exercise*i_j***, where *i* and *j* are digits. Write a program that pads a 0 before *i* if *i* is a single digit. For example, a file named **Exercise2_1** in a directory will be renamed to **Exercise02_1**. In Java, when you pass the symbol `*` from the command line, it refers to all files in the directory (see Supplement III.V). Use the following command to run your program:

```
java Exercise12_28 *
```

****12.29** (*Rename files*) Suppose you have several files in a directory named **Exercise*i_j***, where *i* and *j* are digits. Write a program that pads a 0 before *j* if *j* is a single digit. For example, a file named **Exercise2_1** in a directory will be renamed to **Exercise2_01**. In Java, when you pass the symbol `*` from the command line, it refers to all files in the directory (see Supplement III.V). Use the following command to run your program:

```
java Exercise12_29 *
```

****12.30** (*Occurrences of each letter*) Write a program that prompts the user to enter a file name and displays the occurrences of each letter in the file. Letters are case insensitive. Here is a sample run:

```
Enter a filename: Lincoln.txt ↵Enter
Number of As: 56
Number of Bs: 134
...
Number of Zs: 9
```

***12.31** (*Baby name popularity ranking*) The popularity ranking of baby names from years 2001 to 2010 is downloaded from www.ssa.gov/oact/babynames and stored in files named **babynameranking2001.txt**, **babynameranking2002.txt**, . . . , **babynameranking2010.txt**. You can download these files using the URL such as http://liveexample.pearsoncmg.com/data/babynamesranking2001.txt. Each file contains 1,000 lines. Each line contains a ranking, a boy's name, number for the boy's name, a girl's name, and number for the girl's name. For example, the first two lines in the file **babynameranking2010.txt** are as follows:

```
1       Jacob      21,875       Isabella       22,731
2       Ethan      17,866       Sophia         20,477
```

Therefore, the boy's name Jacob and girl's name Isabella are ranked #1 and the boy's name Ethan and girl's name Sophia are ranked #2; 21,875 boys are named Jacob, and 22,731 girls are named Isabella. Write a program that prompts the user to enter the year, gender, followed by a name, and displays the ranking of the name for the year. Your program should read the data directly from the Web. Here are some sample runs:

```
Enter the year: 2010 ↵Enter
Enter the gender: M ↵Enter
Enter the name: Javier ↵Enter
Javier is ranked #190 in year 2010
```

```
Enter the year: 2010 ↵Enter
Enter the gender: F ↵Enter
Enter the name: ABC ↵Enter
The name ABC is not ranked in year 2010
```

***12.32** (*Ranking summary*) Write a program that uses the files described in Programming Exercise 12.31 and displays a ranking summary table for the first five girl's and boy's names as follows:

```
Year Rank 1    Rank 2    Rank 3    Rank 4    Rank 5    Rank 1   Rank 2    Rank 3    Rank 4    Rank 5
2010 Isabella  Sophia    Emma      Olivia    Ava       Jacob    Ethan     Michael   Jayden    William
2009 Isabella  Emma      Olivia    Sophia    Ava       Jacob    Ethan     Michael   Alexander William
2008 Emma      Isabella  Emily     Olivia    Ava       Jacob    Michael   Ethan     Joshua    Daniel
2007 Emily     Isabella  Emma      Ava       Madison   Jacob    Michael   Ethan     Joshua    Daniel
2006 Emily     Emma      Madison   Isabella  Ava       Jacob    Michael   Joshua    Ethan     Matthew
2005 Emily     Emma      Madison   Abigail   Olivia    Jacob    Michael   Joshua    Matthew   Ethan
2004 Emily     Emma      Madison   Olivia    Hannah    Jacob    Michael   Joshua    Matthew   Ethan
2003 Emily     Emma      Madison   Hannah    Olivia    Jacob    Michael   Joshua    Matthew   Andrew
2002 Emily     Madison   Hannah    Emma      Alexis    Jacob    Michael   Joshua    Matthew   Ethan
2001 Emily     Madison   Hannah    Ashley    Alexis    Jacob    Michael   Matthew   Joshua    Christopher
```

****12.33** (*Search Web*) Modify Listing 12.18 WebCrawler.java to search for the word (e.g., Computer Programming) starting from a URL (e.g., http://cs.armstrong.edu/liang). Your program prompts the user to enter the word and the starting URL and terminates once the word is found. Display the URL for the page that contains the word.