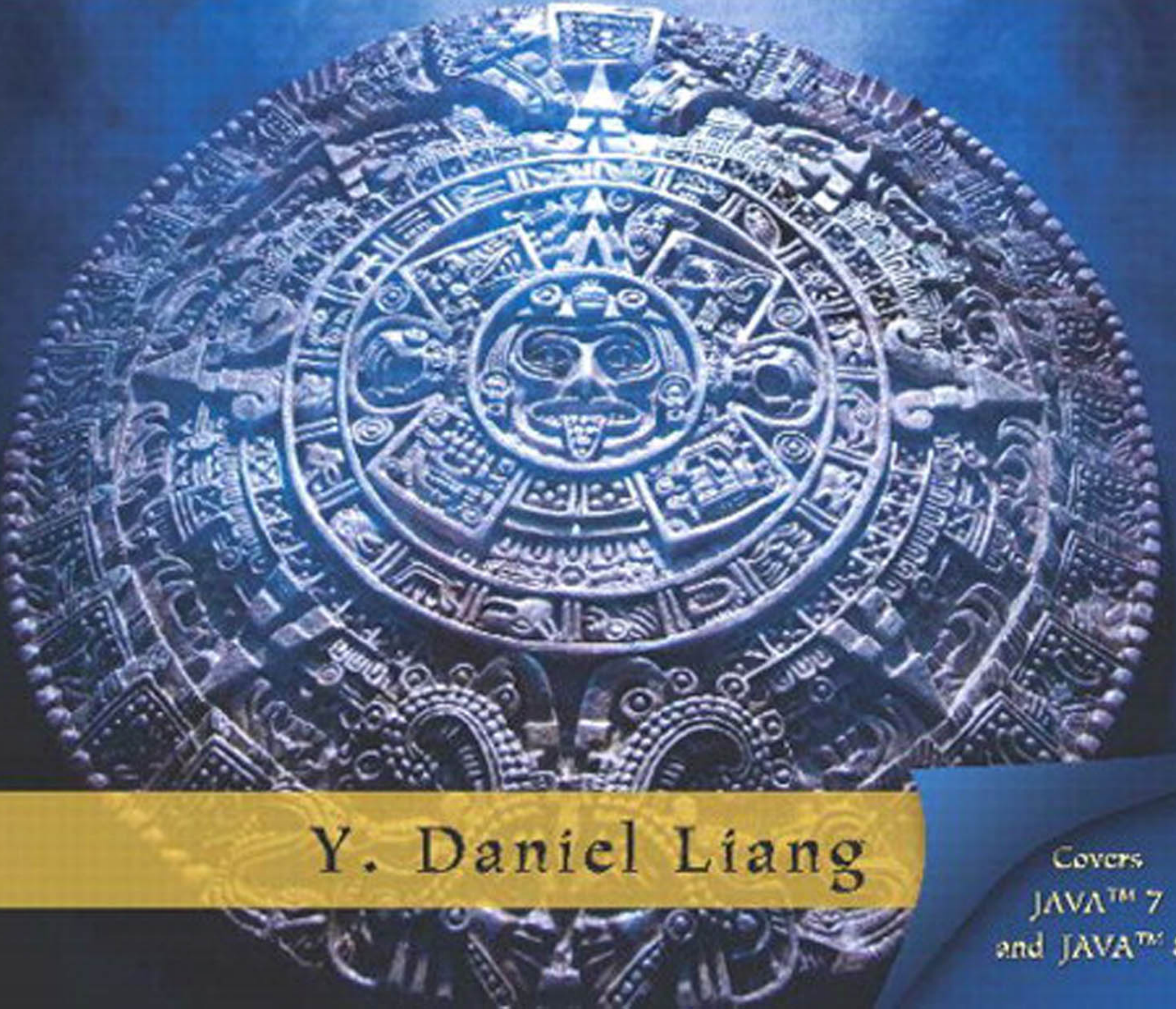
INTRODUCTION TO JAVA™ PROGRAMMING

COMPREHENSIVE VERSION

10TH EDITION



Y. Daniel Liang

Covers JAVA™ 7 and JAVA™ 8

INTRODUCTION TO JAVA® PROGRAMMING

COMPREHENSIVE VERSION

Tenth Edition

Y. Daniel Liang

Armstrong Atlantic State University

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

CHAPTER

9

Objects and Classes

Objectives

- To describe objects and classes, and use classes to model objects (§9.2).
- To use UML graphical notation to describe classes and objects (§9.2).
- To demonstrate how to define classes and create objects (§9.3).
- To create objects using constructors (§9.4).
- To access objects via object reference variables (§9.5).
- To define a reference variable using a reference type (§9.5.1).
- To access an object's data and methods using the object member access operator (`.`) (§9.5.2).
- To define data fields of reference types and assign default values for an object's data fields (§9.5.3).
- To distinguish between object reference variables and primitive data type variables (§9.5.4).
- To use the Java library classes `Date`, `Random`, and `Point2D` (§9.6).
- To distinguish between instance and static variables and methods (§9.7).
- To define private data fields with appropriate getter and setter methods (§9.8).
- To encapsulate data fields to make classes easy to maintain (§9.9).
- To develop methods with object arguments and differentiate between primitive-type arguments and object-type arguments (§9.10).
- To store and process objects in arrays (§9.11).
- To create immutable objects from immutable classes to protect the contents of objects (§9.12).
- To determine the scope of variables in the context of a class (§9.13).
- To use the keyword `this` to refer to the calling object itself (§9.14).

9.1 Introduction

Key Point

Object-oriented programming enables you to develop large-scale software and GUIs effectively.

why OOP?

Having learned the material in the preceding chapters, you are able to solve many programming problems using selections, loops, methods, and arrays. However, these Java features are not sufficient for developing graphical user interfaces and large-scale software systems. Suppose you want to develop a graphical user interface (GUI, pronounced *goo-ee*) as shown in Figure 9.1. How would you program it?

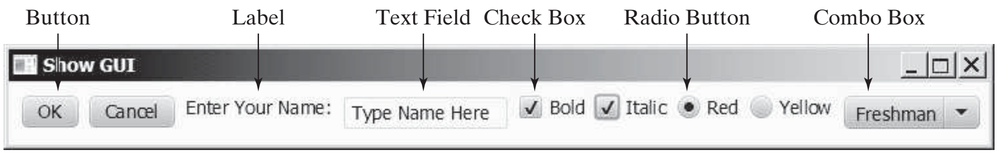


FIGURE 9.1 The GUI objects are created from classes.

This chapter introduces object-oriented programming, which you can use to develop GUI and large-scale software systems.

9.2 Defining Classes for Objects

Key Point

A class defines the properties and behaviors for objects.

VideoNote
Define classes and objects

object
state of an object
properties
attributes
data fields
behavior
actions

Object-oriented programming (OOP) involves programming using objects. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behavior.

- The *state* of an object (also known as its *properties* or *attributes*) is represented by *data fields* with their current values. A circle object, for example, has a data field `radius`, which is the property that characterizes a circle. A rectangle object has the data fields `width` and `height`, which are the properties that characterize a rectangle.
- The *behavior* of an object (also known as its *actions*) is defined by methods. To invoke a method on an object is to ask the object to perform an action. For example, you may define methods named `getArea()` and `getPerimeter()` for circle objects. A circle object may invoke `getArea()` to return its area and `getPerimeter()` to return its perimeter. You may also define the `setRadius(radius)` method. A circle object can invoke this method to change its radius.

class
contract
instantiation
instance

Objects of the same type are defined using a common class. A *class* is a template, blueprint, or *contract* that defines what an object's data fields and methods will be. An object is an instance of a class. You can create many instances of a class. Creating an instance is referred to as *instantiation*. The terms *object* and *instance* are often interchangeable. The relationship between classes and objects is analogous to that between an apple-pie recipe and apple pies: You can make as many apple pies as you want from a single recipe. Figure 9.2 shows a class named `Circle` and its three objects.

data field
method
constructors

A Java class uses variables to define data fields and methods to define actions. Additionally, a class provides methods of a special type, known as *constructors*, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects. Figure 9.3 shows an example of defining the class for circle objects.

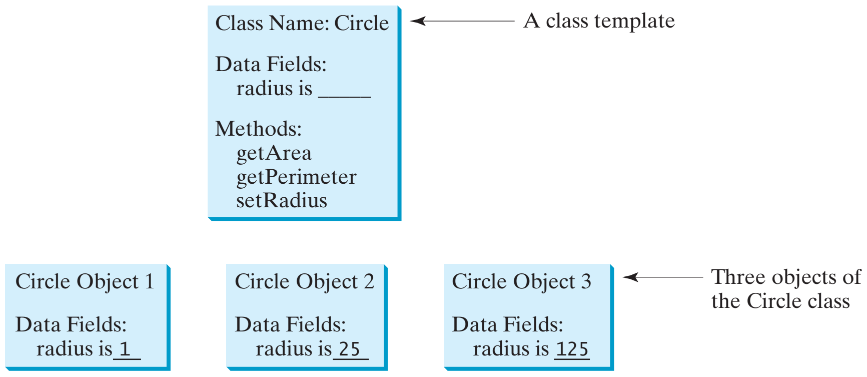
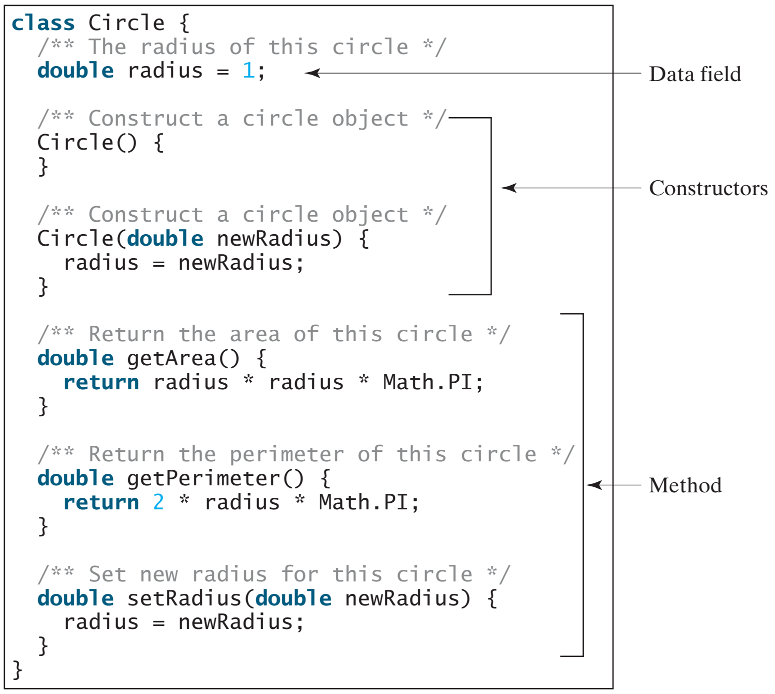


FIGURE 9.2 A class is a template for creating objects.



```
class Circle {
  /** The radius of this circle */
  double radius = 1;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }

  /** Return the perimeter of this circle */
  double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Set new radius for this circle */
  double setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

FIGURE 9.3 A class is a construct that defines objects of the same type.

The **Circle** class is different from all of the other classes you have seen thus far. It does not have a **main** method and therefore cannot be run; it is merely a definition for circle objects. The class that contains the **main** method will be referred to in this book, for convenience, as the *main class*.

main class

The illustration of class templates and objects in Figure 9.2 can be standardized using *Unified Modeling Language (UML)* notation. This notation, as shown in Figure 9.4, is called a *UML class diagram*, or simply a *class diagram*. In the class diagram, the data field is denoted as

Unified Modeling Language (UML)

class diagram

```
dataFieldName: dataFieldType
```

The constructor is denoted as

```
ClassName(parameterName: parameterType)
```

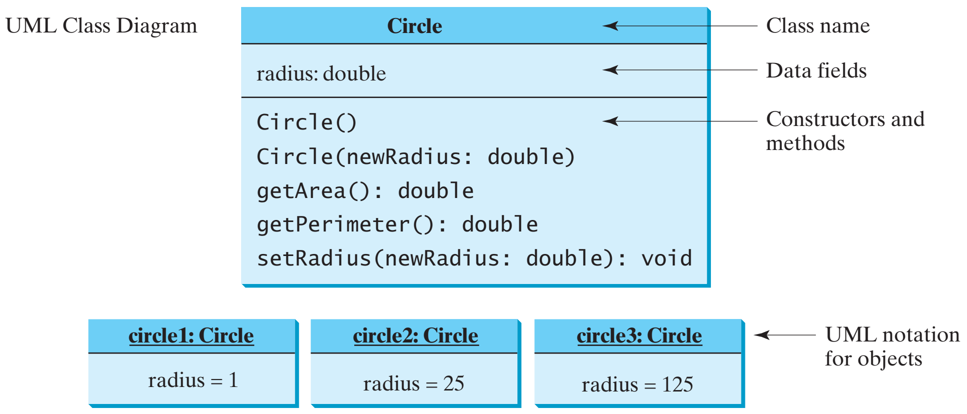


FIGURE 9.4 Classes and objects can be represented using UML notation.

The method is denoted as

```
methodName(parameterName: parameterType): returnType
```

9.3 Example: Defining Classes and Creating Objects

Key Point

Classes are definitions for objects and objects are created from classes.

This section gives two examples of defining classes and uses the classes to create objects. Listing 9.1 is a program that defines the `Circle` class and uses it to create objects. The program constructs three circle objects with radius `1`, `25`, and `125` and displays the radius and area of each of the three circles. It then changes the radius of the second object to `100` and displays its new radius and area.

avoid naming conflicts

Note

To avoid a naming conflict with several enhanced versions of the `Circle` class introduced later in the chapter, the `Circle` class in this example is named `SimpleCircle`. For simplicity, we will still refer to the class in the text as `Circle`.

LISTING 9.1 `TestSimpleCircle.java`

main class
main method
create object
create object
create object

```
public class TestSimpleCircle {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 1
    SimpleCircle circle1 = new SimpleCircle();
    System.out.println("The area of the circle of radius "
      + circle1.radius + " is " + circle1.getArea());

    // Create a circle with radius 25
    SimpleCircle circle2 = new SimpleCircle(25);
    System.out.println("The area of the circle of radius "
      + circle2.radius + " is " + circle2.getArea());

    // Create a circle with radius 125
    SimpleCircle circle3 = new SimpleCircle(125);
    System.out.println("The area of the circle of radius "
      + circle3.radius + " is " + circle3.getArea());

    // Modify circle radius
    circle2.radius = 100; // or circle2.setRadius(100)
    System.out.println("The area of the circle of radius "
      + circle2.radius + " is " + circle2.getArea());
```

```java
    }
  }

  // Define the circle class with two constructors
  class SimpleCircle {
    double radius;

    /** Construct a circle with radius 1 */
    SimpleCircle() {
      radius = 1;
    }

    /** Construct a circle with a specified radius */
    SimpleCircle(double newRadius) {
      radius = newRadius;
    }

    /** Return the area of this circle */
    double getArea() {
      return radius * radius * Math.PI;
    }

    /** Return the perimeter of this circle */
    double getPerimeter() {
      return 2 * radius * Math.PI;
    }

    /** Set a new radius for this circle */
    void setRadius(double newRadius) {
      radius = newRadius;
    }
  }
```

class `SimpleCircle`
data field
no-arg constructor
second constructor
`getArea`
`getPerimeter`
`setRadius`

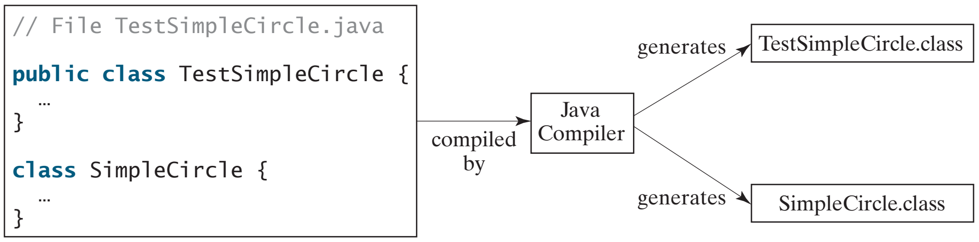
```
The area of the circle of radius 1.0 is 3.141592653589793
The area of the circle of radius 25.0 is 1963.4954084936207
The area of the circle of radius 125.0 is 49087.385212340516
The area of the circle of radius 100.0 is 31415.926535897932
```

The program contains two classes. The first of these, `TestSimpleCircle`, is the main class. Its sole purpose is to test the second class, `SimpleCircle`. Such a program that uses the class is often referred to as a *client* of the class. When you run the program, the Java runtime system invokes the `main` method in the main class.

client

You can put the two classes into one file, but only one class in the file can be a *public class*. Furthermore, the public class must have the same name as the file name. Therefore, the file name is **TestSimpleCircle.java**, since `TestSimpleCircle` is public. Each class in the source code is compiled into a **.class** file. When you compile **TestSimpleCircle.java**, two class files **TestSimpleCircle.class** and **SimpleCircle.class** are generated, as shown in Figure 9.5.

public class



```java
// File TestSimpleCircle.java

public class TestSimpleCircle {
  ...
}

class SimpleCircle {
  ...
}
```

FIGURE 9.5 Each class in the source code file is compiled into a **.class** file.

The main class contains the **main** method (line 3) that creates three objects. As in creating an array, the **new** operator is used to create an object from the constructor: **new SimpleCircle()** creates an object with radius **1** (line 5), **new SimpleCircle(25)** creates an object with radius **25** (line 10), and **new SimpleCircle(125)** creates an object with radius **125** (line 15).

These three objects (referenced by **circle1**, **circle2**, and **circle3**) have different data but the same methods. Therefore, you can compute their respective areas by using the **getArea()** method. The data fields can be accessed via the reference of the object using **circle1.radius**, **circle2.radius**, and **circle3.radius**, respectively. The object can invoke its method via the reference of the object using **circle1.getArea()**, **circle2.getArea()**, and **circle3.getArea()**, respectively.

These three objects are independent. The radius of **circle2** is changed to **100** in line 20. The object's new radius and area are displayed in lines 21–22.

There are many ways to write Java programs. For instance, you can combine the two classes in the example into one, as shown in Listing 9.2.

LISTING 9.2 SimpleCircle.java

main method

data field

no-arg constructor

second constructor

method

```
 1 public class SimpleCircle {
 2   /** Main method */
 3   public static void main(String[] args) {
 4     // Create a circle with radius 1
 5     SimpleCircle circle1 = new SimpleCircle();
 6     System.out.println("The area of the circle of radius "
 7       + circle1.radius + " is " + circle1.getArea());
 8
 9     // Create a circle with radius 25
10     SimpleCircle circle2 = new SimpleCircle(25);
11     System.out.println("The area of the circle of radius "
12       + circle2.radius + " is " + circle2.getArea());
13
14     // Create a circle with radius 125
15     SimpleCircle circle3 = new SimpleCircle(125);
16     System.out.println("The area of the circle of radius "
17       + circle3.radius + " is " + circle3.getArea());
18
19     // Modify circle radius
20     circle2.radius = 100;
21     System.out.println("The area of the circle of radius "
22       + circle2.radius + " is " + circle2.getArea());
23   }
24
25   double radius;
26
27   /** Construct a circle with radius 1 */
28   SimpleCircle() {
29     radius = 1;
30   }
31
32   /** Construct a circle with a specified radius */
33   SimpleCircle(double newRadius) {
34     radius = newRadius;
35   }
36
37   /** Return the area of this circle */
38   double getArea() {
39     return radius * radius * Math.PI;
40   }
41
```

```
  /** Return the perimeter of this circle */
  double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Set a new radius for this circle */
  void setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

Since the combined class has a **main** method, it can be executed by the Java interpreter. The **main** method is the same as that in Listing 9.1. This demonstrates that you can test a class by simply adding a **main** method in the same class.

As another example, consider television sets. Each TV is an object with states (current channel, current volume level, power on or off) and behaviors (change channels, adjust volume, turn on/off). You can use a class to model TV sets. The UML diagram for the class is shown in Figure 9.6.

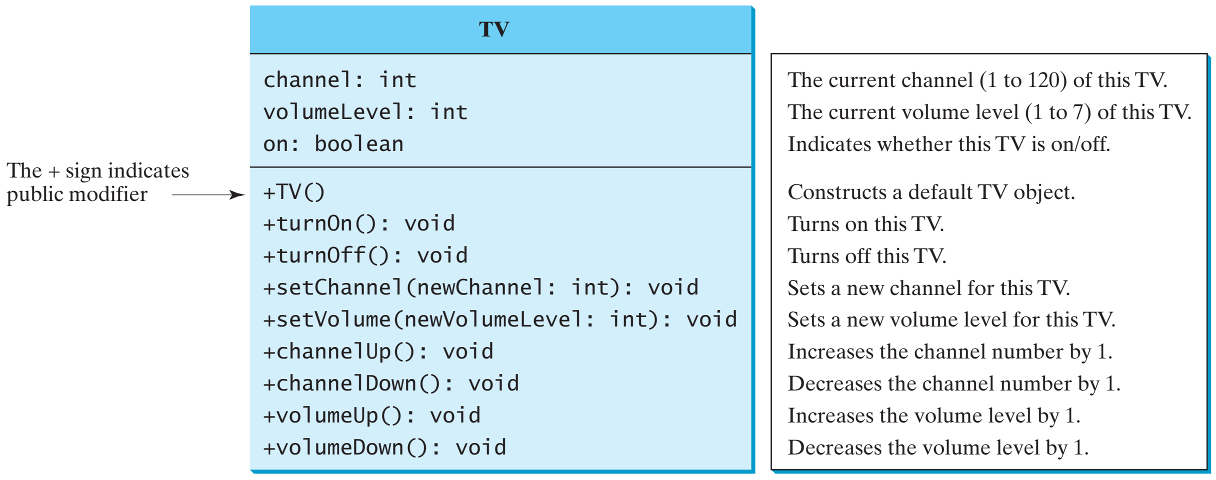

TV
channel: int
volumeLevel: int
on: boolean
+TV()
+turnOn(): void
+turnOff(): void
+setChannel(newChannel: int): void
+setVolume(newVolumeLevel: int): void
+channelUp(): void
+channelDown(): void
+volumeUp(): void
+volumeDown(): void
The current channel (1 to 120) of this TV.
The current volume level (1 to 7) of this TV.
Indicates whether this TV is on/off.
Constructs a default TV object.
Turns on this TV.
Turns off this TV.
Sets a new channel for this TV.
Sets a new volume level for this TV.
Increases the channel number by 1.
Decreases the channel number by 1.
Increases the volume level by 1.
Decreases the volume level by 1.
The + sign indicates public modifier

FIGURE 9.6 The TV class models TV sets.

Listing 9.3 gives a program that defines the **TV** class.

LISTING 9.3 TV.java

```
public class TV {
  int channel = 1; // Default channel is 1
  int volumeLevel = 1; // Default volume level is 1
  boolean on = false; // TV is off

  public TV() {
  }

  public void turnOn() {
    on = true;
  }

  public void turnOff() {
```

data fields

constructor

turn on TV

turn off TV

```java
      on = false;
    }

    public void setChannel(int newChannel) {      // set a new channel
      if (on && newChannel >= 1 && newChannel <= 120)
        channel = newChannel;
    }

    public void setVolume(int newVolumeLevel) {   // set a new volume
      if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)
        volumeLevel = newVolumeLevel;
    }

    public void channelUp() {                     // increase channel
      if (on && channel < 120)
        channel++;
    }

    public void channelDown() {                   // decrease channel
      if (on && channel > 1)
        channel--;
    }

    public void volumeUp() {                      // increase volume
      if (on && volumeLevel < 7)
        volumeLevel++;
    }

    public void volumeDown() {                    // decrease volume
      if (on && volumeLevel > 1)
        volumeLevel--;
    }
  }
```

set a new channel

set a new volume

increase channel

decrease channel

increase volume

decrease volume

The constructor and methods in the TV class are defined public so they can be accessed from other classes. Note that the channel and volume level are not changed if the TV is not on. Before either of these is changed, its current value is checked to ensure that it is within the correct range.

Listing 9.4 gives a program that uses the TV class to create two objects.

LISTING 9.4 TestTV.java

main method
create a TV
turn on
set a new channel
set a new volume
create a TV
turn on
increase channel
increase volume
display state

```java
public class TestTV {
  public static void main(String[] args) {
    TV tv1 = new TV();
    tv1.turnOn();
    tv1.setChannel(30);
    tv1.setVolume(3);

    TV tv2 = new TV();
    tv2.turnOn();
    tv2.channelUp();
    tv2.channelUp();
    tv2.volumeUp();

    System.out.println("tv1's channel is " + tv1.channel
      + " and volume level is " + tv1.volumeLevel);
    System.out.println("tv2's channel is " + tv2.channel
      + " and volume level is " + tv2.volumeLevel);
  }
}
```

```
tv1's channel is 30 and volume level is 3
tv2's channel is 3 and volume level is 2
```

The program creates two objects in lines 3 and 8 and invokes the methods on the objects to perform actions for setting channels and volume levels and for increasing channels and volumes. The program displays the state of the objects in lines 14–17. The methods are invoked using syntax such as `tv1.turnOn()` (line 4). The data fields are accessed using syntax such as `tv1.channel` (line 14).

These examples have given you a glimpse of classes and objects. You may have many questions regarding constructors, objects, reference variables, accessing data fields, and invoking object's methods. The sections that follow discuss these issues in detail.

Check Point

9.1 Describe the relationship between an object and its defining class.

9.2 How do you define a class?

9.3 How do you declare an object's reference variable?

9.4 How do you create an object?

9.4 Constructing Objects Using Constructors

Key Point

A constructor is invoked to create an object using the `new` *operator.*

Constructors are a special kind of method. They have three peculiarities:

constructor's name

- A constructor must have the same name as the class itself.

no return type

- Constructors do not have a return type—not even `void`.

`new` operator

- Constructors are invoked using the `new` operator when an object is created. Constructors play the role of initializing objects.

overloaded constructors

The constructor has exactly the same name as its defining class. Like regular methods, constructors can be overloaded (i.e., multiple constructors can have the same name but different signatures), making it easy to construct objects with different initial data values.

It is a common mistake to put the `void` keyword in front of a constructor. For example,

no `void`

```
public void Circle() {
}
```

In this case, `Circle()` is a method, not a constructor.

constructing objects

Constructors are used to construct objects. To construct an object from a class, invoke a constructor of the class using the `new` operator, as follows:

```
new ClassName(arguments);
```

For example, `new Circle()` creates an object of the `Circle` class using the first constructor defined in the `Circle` class, and `new Circle(25)` creates an object using the second constructor defined in the `Circle` class.

A class normally provides a constructor without arguments (e.g., `Circle()`). Such a constructor is referred to as a *no-arg* or *no-argument constructor*.

no-arg constructor

A class may be defined without constructors. In this case, a public no-arg constructor with an empty body is implicitly defined in the class. This constructor, called a *default constructor*, is provided automatically *only if no constructors are explicitly defined in the class*.

default constructor

Check Point

9.5 What are the differences between constructors and methods?

9.6 When will a class have a default constructor?

9.5 Accessing Objects via Reference Variables

Key Point

An object's data and methods can be accessed through the dot (`.`) operator via the object's reference variable.

Newly created objects are allocated in the memory. They can be accessed via reference variables.

9.5.1 Reference Variables and Reference Types

reference variable

Objects are accessed via the object's *reference variables*, which contain references to the objects. Such variables are declared using the following syntax:

```
ClassName objectRefVar;
```

reference type

A class is essentially a programmer-defined type. A class is a *reference type*, which means that a variable of the class type can reference an instance of the class. The following statement declares the variable `myCircle` to be of the `Circle` type:

```
Circle myCircle;
```

The variable `myCircle` can reference a `Circle` object. The next statement creates an object and assigns its reference to `myCircle`:

```
myCircle = new Circle();
```

You can write a single statement that combines the declaration of an object reference variable, the creation of an object, and the assigning of an object reference to the variable with the following syntax:

```
ClassName objectRefVar = new ClassName();
```

Here is an example:

```
Circle myCircle = new Circle();
```

The variable `myCircle` holds a reference to a `Circle` object.

object vs. object reference variable

Note
An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored. Therefore, it is fine, for simplicity, to say that `myCircle` is a `Circle` object rather than use the longer-winded description that `myCircle` is a variable that contains a reference to a `Circle` object.

array object

Note
Arrays are treated as objects in Java. Arrays are created using the `new` operator. An array variable is actually a variable that contains a reference to an array.

9.5.2 Accessing an Object's Data and Methods

dot operator (.)

In OOP terminology, an object's member refers to its data fields and methods. After an object is created, its data can be accessed and its methods can be invoked using the *dot operator* (`.`), also known as the *object member access operator*:

- `objectRefVar.dataField` references a data field in the object.
- `objectRefVar.method(arguments)` invokes a method on the object.

For example, **myCircle.radius** references the radius in **myCircle**, and **myCircle.getArea()** invokes the **getArea** method on **myCircle**. Methods are invoked as operations on objects.

The data field **radius** is referred to as an *instance variable*, because it is dependent on a specific instance. For the same reason, the method **getArea** is referred to as an *instance method*, because you can invoke it only on a specific instance. The object on which an instance method is invoked is called a *calling object*.

instance variable

instance method

calling object

Caution

Recall that you use **Math.methodName(arguments)** (e.g., **Math.pow(3, 2.5)**) to invoke a method in the **Math** class. Can you invoke **getArea()** using **Circle.getArea()**? The answer is no. All the methods in the **Math** class are static methods, which are defined using the **static** keyword. However, **getArea()** is an instance method, and thus nonstatic. It must be invoked from an object using **objectRefVar.methodName(arguments)** (e.g., **myCircle.getArea()**). Further explanation is given in Section 9.7, Static Variables, Constants, and Methods.

invoking methods

Note

Usually you create an object and assign it to a variable, and then later you can use the variable to reference the object. Occasionally an object does not need to be referenced later. In this case, you can create an object without explicitly assigning it to a variable using the syntax:

```
new Circle();
```

or

```
System.out.println("Area is " + new Circle(5).getArea());
```

The former statement creates a **Circle** object. The latter creates a **Circle** object and invokes its **getArea** method to return its area. An object created in this way is known as an *anonymous object*.

anonymous object

9.5.3 Reference Data Fields and the null Value

The data fields can be of reference types. For example, the following **Student** class contains a data field **name** of the **String** type. **String** is a predefined Java class.

reference data fields

```
class Student {
  String name; // name has the default value null
  int age; // age has the default value 0
  boolean isScienceMajor; // isScienceMajor has default value false
  char gender; // gender has default value '\u0000'
}
```

If a data field of a reference type does not reference any object, the data field holds a special Java value, **null**. **null** is a literal just like **true** and **false**. While **true** and **false** are Boolean literals, **null** is a literal for a reference type.

null value

The default value of a data field is **null** for a reference type, **0** for a numeric type, **false** for a **boolean** type, and **\u0000** for a **char** type. However, Java assigns no default value to a local variable inside a method. The following code displays the default values of the data fields **name**, **age**, **isScienceMajor**, and **gender** for a **Student** object:

default field values

```
class Test {
  public static void main(String[] args) {
    Student student = new Student();
    System.out.println("name? " + student.name);
```

```
      System.out.println("age? " + student.age);
      System.out.println("isScienceMajor? " + student.isScienceMajor);
      System.out.println("gender? " + student.gender);
    }
  }
```

The following code has a compile error, because the local variables `x` and `y` are not initialized:

```
  class Test {
    public static void main(String[] args) {
      int x; // x has no default value
      String y; // y has no default value
      System.out.println("x is " + x);
      System.out.println("y is " + y);
    }
  }
```

NullPointerException

Caution

`NullPointerException` is a common runtime error. It occurs when you invoke a method on a reference variable with a `null` value. Make sure you assign an object reference to the variable before invoking the method through the reference variable (See Checkpoint Question 9.11c).

9.5.4 Differences between Variables of Primitive Types and Reference Types

Every variable represents a memory location that holds a value. When you declare a variable, you are telling the compiler what type of value the variable can hold. For a variable of a primitive type, the value is of the primitive type. For a variable of a reference type, the value is a reference to where an object is located. For example, as shown in Figure 9.7, the value of `int` variable `i` is `int` value `1`, and the value of `Circle` object `c` holds a reference to where the contents of the `Circle` object are stored in memory.

When you assign one variable to another, the other variable is set to the same value. For a variable of a primitive type, the real value of one variable is assigned to the other variable. For a variable of a reference type, the reference of one variable is assigned to the other variable. As shown in Figure 9.8, the assignment statement `i = j` copies the contents of `j` into `i`

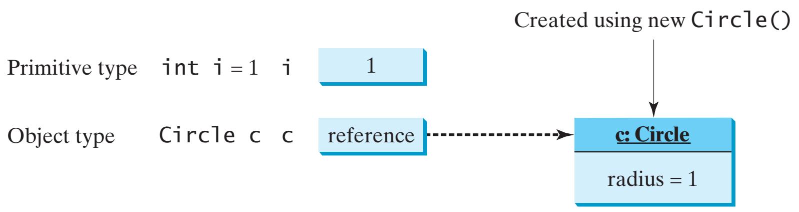


FIGURE 9.7 A variable of a primitive type holds a value of the primitive type, and a variable of a reference type holds a reference to where an object is stored in memory.

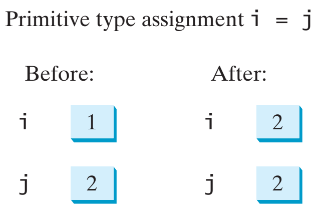


FIGURE 9.8 Primitive variable `j` is copied to variable `i`.

for primitive variables. As shown in Figure 9.9, the assignment statement `c1 = c2` copies the reference of `c2` into `c1` for reference variables. After the assignment, variables `c1` and `c2` refer to the same object.

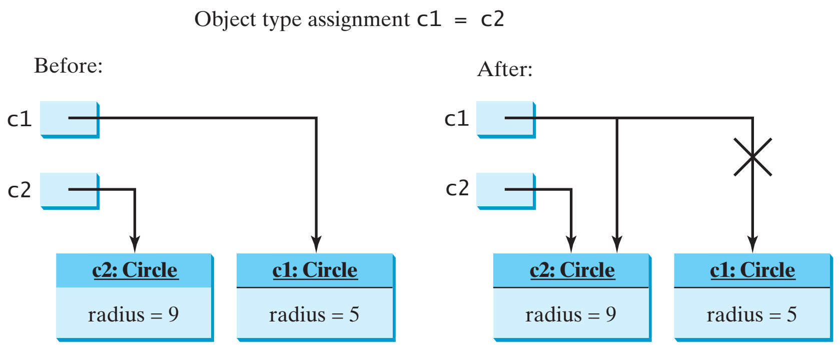


FIGURE 9.9 Reference variable `c2` is copied to variable `c1`.

Note

As illustrated in Figure 9.9, after the assignment statement `c1 = c2`, `c1` points to the same object referenced by `c2`. The object previously referenced by `c1` is no longer useful and therefore is now known as *garbage*. Garbage occupies memory space, so the Java runtime system detects garbage and automatically reclaims the space it occupies. This process is called *garbage collection*.

garbage

garbage collection

Tip

If you know that an object is no longer needed, you can explicitly assign `null` to a reference variable for the object. The JVM will automatically collect the space if the object is not referenced by any reference variable.

Check Point

9.7 Which operator is used to access a data field or invoke a method from an object?

9.8 What is an anonymous object?

9.9 What is `NullPointerException`?

9.10 Is an array an object or a primitive type value? Can an array contain elements of an object type? Describe the default value for the elements of an array.

9.11 What is wrong with each of the following programs?

```
public class ShowErrors {
  public static void main(String[] args) {
    ShowErrors t = new ShowErrors(5);
  }
}
```

(a)

```
public class ShowErrors {
  public static void main(String[] args) {
    ShowErrors t = new ShowErrors();
    t.x();
  }
}
```

(b)

```
public class ShowErrors {
  public void method1() {
    Circle c;
    System.out.println("What is radius "
      + c.getRadius());
    c = new Circle();
  }
}
```

(c)

```
public class ShowErrors {
  public static void main(String[] args) {
    C c = new C(5.0);
    System.out.println(c.value);
  }
}

class C {
  int value = 2;
}
```

(d)

9.12 What is wrong in the following code?

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

class A {
  String s;

  A(String newS) {
    s = newS;
  }

  public void print() {
    System.out.print(s);
  }
}
```

9.13 What is the output of the following code?

```
public class A {
  boolean x;

  public static void main(String[] args) {
    A a = new A();
    System.out.println(a.x);
  }
}
```

9.6 Using Classes from the Java Library

Key Point

The Java API contains a rich set of classes for developing Java programs.

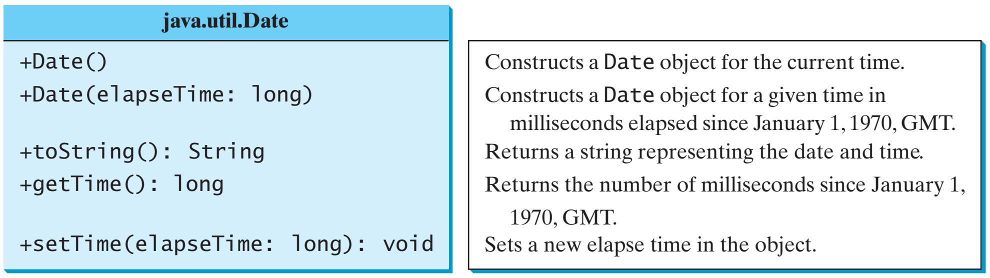
Listing 9.1 defined the `SimpleCircle` class and created objects from the class. You will frequently use the classes in the Java library to develop programs. This section gives some examples of the classes in the Java library.

VideoNote
Use classes

9.6.1 The `Date` Class

In Listing 2.7, ShowCurrentTime.java, you learned how to obtain the current time using `System.currentTimeMillis()`. You used the division and remainder operators to extract the current second, minute, and hour. Java provides a system-independent encapsulation of date and time in the `java.util.Date` class, as shown in Figure 9.10.

`java.util.Date` class



java.util.Date	
`+Date()`	Constructs a `Date` object for the current time.
`+Date(elapseTime: long)`	Constructs a `Date` object for a given time in milliseconds elapsed since January 1, 1970, GMT.
`+toString(): String`	Returns a string representing the date and time.
`+getTime(): long`	Returns the number of milliseconds since January 1, 1970, GMT.
`+setTime(elapseTime: long): void`	Sets a new elapse time in the object.

FIGURE 9.10 A `Date` object represents a specific date and time.

You can use the no-arg constructor in the **Date** class to create an instance for the current date and time, the **getTime()** method to return the elapsed time since January 1, 1970, GMT, and the **toString()** method to return the date and time as a string. For example, the following code

```
java.util.Date date = new java.util.Date();
System.out.println("The elapsed time since Jan 1, 1970 is " +
  date.getTime() + " milliseconds");
System.out.println(date.toString());
```

create object

get elapsed time

invoke toString

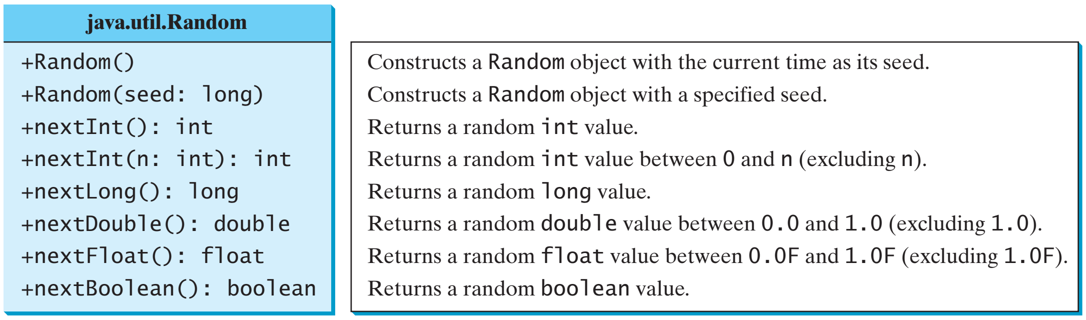
displays the output like this:

```
The elapsed time since Jan 1, 1970 is 1324903419651 milliseconds
Mon Dec 26 07:43:39 EST 2011
```

The **Date** class has another constructor, **Date(long elapseTime)**, which can be used to construct a **Date** object for a given time in milliseconds elapsed since January 1, 1970, GMT.

9.6.2 The **Random** Class

You have used **Math.random()** to obtain a random **double** value between **0.0** and **1.0** (excluding **1.0**). Another way to generate random numbers is to use the **java.util.Random** class, as shown in Figure 9.11, which can generate a random **int**, **long**, **double**, **float**, and **boolean** value.



java.util.Random	
+Random()	Constructs a Random object with the current time as its seed.
+Random(seed: long)	Constructs a Random object with a specified seed.
+nextInt(): int	Returns a random int value.
+nextInt(n: int): int	Returns a random int value between 0 and n (excluding n).
+nextLong(): long	Returns a random long value.
+nextDouble(): double	Returns a random double value between 0.0 and 1.0 (excluding 1.0).
+nextFloat(): float	Returns a random float value between 0.0F and 1.0F (excluding 1.0F).
+nextBoolean(): boolean	Returns a random boolean value.

FIGURE 9.11 A **Random** object can be used to generate random values.

When you create a **Random** object, you have to specify a seed or use the default seed. A seed is a number used to initialize a random number generator. The no-arg constructor creates a **Random** object using the current elapsed time as its seed. If two **Random** objects have the same seed, they will generate identical sequences of numbers. For example, the following code creates two **Random** objects with the same seed, **3**.

```
Random random1 = new Random(3);
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
  System.out.print(random1.nextInt(1000) + " ");

Random random2 = new Random(3);
System.out.print("\nFrom random2: ");
for (int i = 0; i < 10; i++)
  System.out.print(random2.nextInt(1000) + " ");
```

The code generates the same sequence of random **int** values:

```
From random1: 734 660 210 581 128 202 549 564 459 961
From random2: 734 660 210 581 128 202 549 564 459 961
```

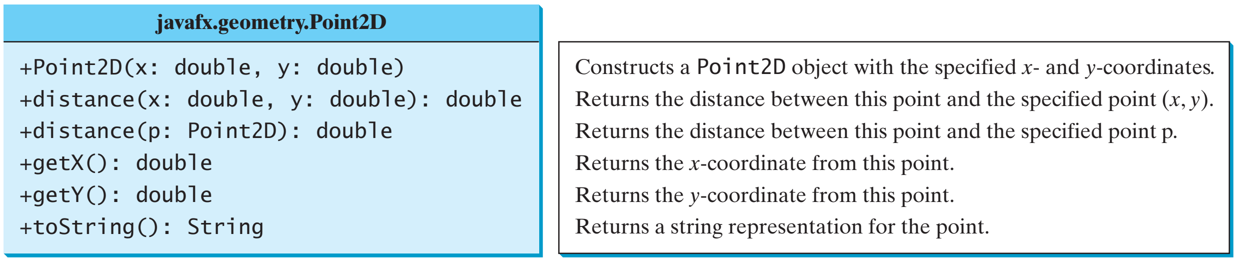
same sequence

Note

The ability to generate the same sequence of random values is useful in software testing and many other applications. In software testing, often you need to reproduce the test cases from a fixed sequence of random numbers.

9.6.3 The **Point2D** Class

Java API has a conveninent **Point2D** class in the **javafx.geometry** package for representing a point in a two-dimensional plane. The UML diagram for the class is shown in Figure 9.12.



javafx.geometry.Point2D

`+Point2D(x: double, y: double)`	Constructs a `Point2D` object with the specified *x*- and *y*-coordinates.
`+distance(x: double, y: double): double`	Returns the distance between this point and the specified point (*x*, *y*).
`+distance(p: Point2D): double`	Returns the distance between this point and the specified point p.
`+getX(): double`	Returns the *x*-coordinate from this point.
`+getY(): double`	Returns the *y*-coordinate from this point.
`+toString(): String`	Returns a string representation for the point.

FIGURE 9.12 A **Point2D** object represents a point with *x*- and *y*-coordinates.

You can create a **Point2D** object for a point with the specified *x*- and *y*-coordinates, use the **distance** method to compute the distance from this point to another point, and use the **toString()** method to return a string representation of the point. Lisitng 9.5 gives an example of using this class.

LISTING 9.5 TestPoint2D.java

create an object

invoke toString()

get distance

```java
import java.util.Scanner;
import javafx.geometry.Point2D;

public class TestPoint2D {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    System.out.print("Enter point1's x-, y-coordinates: ");
    double x1 = input.nextDouble();
    double y1 = input.nextDouble();
    System.out.print("Enter point2's x-, y-coordinates: ");
    double x2 = input.nextDouble();
    double y2 = input.nextDouble();

    Point2D p1 = new Point2D(x1, y1);
    Point2D p2 = new Point2D(x2, y2);
    System.out.println("p1 is " + p1.toString());
    System.out.println("p2 is " + p2.toString());
    System.out.println("The distance between p1 and p2 is " +
      p1.distance(p2));
  }
}
```

```
Enter point1's x-, y-coordinates: 1.5 5.5 ↵Enter
Enter point2's x-, y-coordinates: -5.3 -4.4 ↵Enter
p1 is Point2D [x = 1.5, y = 5.5]
p2 is Point2D [x = -5.3, y = -4.4]
The distance between p1 and p2 is 12.010412149464313
```

This program creates two objects of the `Point2D` class (lines 15–16). The `toString()` method returns a string that describes the object (lines 17–18). Invoking `p1.distance(p2)` returns the distance between the two points (line 20).

Check Point

9.14 How do you create a `Date` for the current time? How do you display the current time?

9.15 How do you create a `Point2D`? Suppose `p1` and `p2` are two instances of `Point2D`? How do you obtain the distance between the two points?

9.16 Which packages contain the classes `Date`, `Random`, `Point2D`, `System`, and `Math`?

9.7 Static Variables, Constants, and Methods

A static variable is shared by all objects of the class. A static method cannot access instance members of the class.

Key Point

Static vs. instance instance variable

VideoNote Static vs. instance

The data field `radius` in the circle class is known as an *instance variable*. An instance variable is tied to a specific instance of the class; it is not shared among objects of the same class. For example, suppose that you create the following objects:

```
Circle circle1 = new Circle();
Circle circle2 = new Circle(5);
```

The `radius` in `circle1` is independent of the `radius` in `circle2` and is stored in a different memory location. Changes made to `circle1`'s `radius` do not affect `circle2`'s `radius`, and vice versa.

static variable

If you want all the instances of a class to share data, use *static variables*, also known as *class variables*. Static variables store values for the variables in a common memory location. Because of this common location, if one object changes the value of a static variable, all objects of the same class are affected. Java supports static methods as well as static variables. *Static methods* can be called without creating an instance of the class.

static method

Let's modify the `Circle` class by adding a static variable `numberOfObjects` to count the number of circle objects created. When the first object of this class is created, `numberOfObjects` is `1`. When the second object is created, `numberOfObjects` becomes `2`. The UML of the new circle class is shown in Figure 9.13. The `Circle` class defines the instance variable `radius` and the static variable `numberOfObjects`, the instance methods `getRadius`, `setRadius`, and `getArea`, and the static method `getNumberOfObjects`. (Note that static variables and methods are underlined in the UML class diagram.)

To declare a static variable or define a static method, put the modifier `static` in the variable or method declaration. The static variable `numberOfObjects` and the static method `getNumberOfObjects()` can be declared as follows:

declare static variable

```
static int numberOfObjects;
```

define static method

```
static int getNumberObjects() {
  return numberOfObjects;
}
```

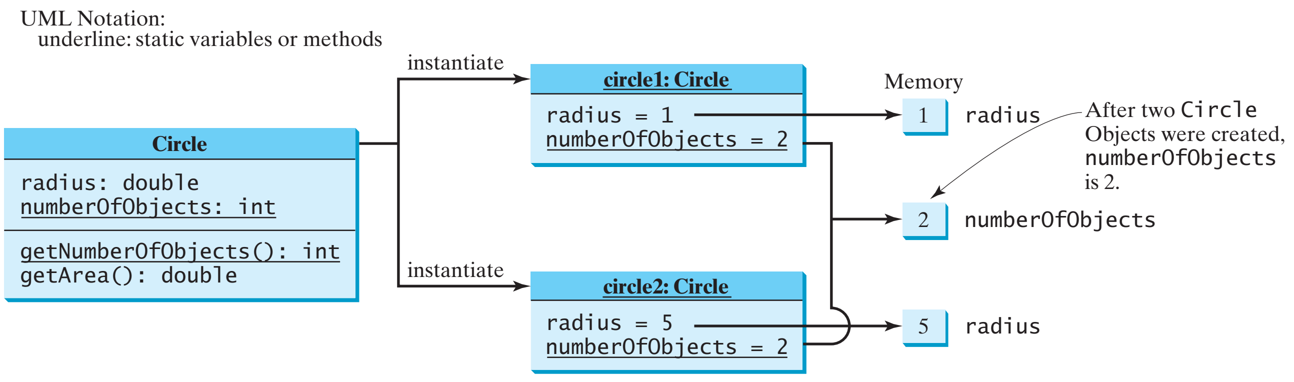


FIGURE 9.13 Instance variables belong to the instances and have memory storage independent of one another. Static variables are shared by all the instances of the same class.

declare constant

Constants in a class are shared by all objects of the class. Thus, constants should be declared as `final static`. For example, the constant `PI` in the `Math` class is defined as:

```
final static double PI = 3.14159265358979323846;
```

The new circle class, named `CircleWithStaticMembers`, is defined in Listing 9.6:

LISTING 9.6 `CircleWithStaticMembers.java`

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

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

  /** Construct a circle with radius 1 */
  CircleWithStaticMembers() {
    radius = 1;
    numberOfObjects++;
  }

  /** Construct a circle with a specified radius */
  CircleWithStaticMembers(double newRadius) {
    radius = newRadius;
    numberOfObjects++;
  }

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

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }
}
```

static variable

increase by 1

increase by 1

static method

Method `getNumberOfObjects()` in `CircleWithStaticMembers` is a static method. All the methods in the `Math` class are static. The `main` method is static, too.

Instance methods (e.g., **getArea()**) and instance data (e.g., **radius**) belong to instances and can be used only after the instances are created. They are accessed via a reference variable. Static methods (e.g., **getNumberOfObjects()**) and static data (e.g., **numberOfObjects**) can be accessed from a reference variable or from their class name.

The program in Listing 9.7 demonstrates how to use instance and static variables and methods and illustrates the effects of using them.

LISTING 9.7 TestCircleWithStaticMembers.java

```
 1  public class TestCircleWithStaticMembers {
 2    /** Main method */
 3    public static void main(String[] args) {
 4      System.out.println("Before creating objects");
 5      System.out.println("The number of Circle objects is " +
 6        CircleWithStaticMembers.numberOfObjects);              static variable
 7
 8      // Create c1
 9      CircleWithStaticMembers c1 = new CircleWithStaticMembers();
10
11      // Display c1 BEFORE c2 is created
12      System.out.println("\nAfter creating c1");
13      System.out.println("c1: radius (" + c1.radius +          instance variable
14        ") and number of Circle objects (" +
15        c1.numberOfObjects + ")");                             static variable
16
17      // Create c2
18      CircleWithStaticMembers c2 = new CircleWithStaticMembers(5);
19
20      // Modify c1
21      c1.radius = 9;                                           instance variable
22
23      // Display c1 and c2 AFTER c2 was created
24      System.out.println("\nAfter creating c2 and modifying c1");
25      System.out.println("c1: radius (" + c1.radius +
26        ") and number of Circle objects (" +
27        c1.numberOfObjects + ")");                             static variable
28      System.out.println("c2: radius (" + c2.radius +
29        ") and number of Circle objects (" +
30        c2.numberOfObjects + ")");                             static variable
31    }
32  }
```

```
Before creating objects
The number of Circle objects is 0
After creating c1
c1: radius (1.0) and number of Circle objects (1)
After creating c2 and modifying c1
c1: radius (9.0) and number of Circle objects (2)
c2: radius (5.0) and number of Circle objects (2)
```

When you compile **TestCircleWithStaticMembers.java**, the Java compiler automatically compiles **CircleWithStaticMembers.java** if it has not been compiled since the last change.

Static variables and methods can be accessed without creating objects. Line 6 displays the number of objects, which is **0**, since no objects have been created.

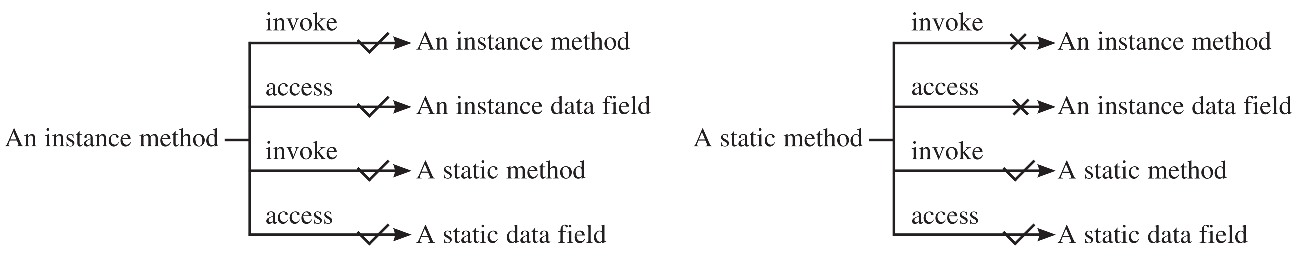
The `main` method creates two circles, `c1` and `c2` (lines 9, 18). The instance variable `radius` in `c1` is modified to become `9` (line 21). This change does not affect the instance variable `radius` in `c2`, since these two instance variables are independent. The static variable `numberOfObjects` becomes `1` after `c1` is created (line 9), and it becomes `2` after `c2` is created (line 18).

Note that `PI` is a constant defined in `Math`, and `Math.PI` references the constant. `c1.numberOfObjects` (line 27) and `c2.numberOfObjects` (line 30) are better replaced by `CircleWithStaticMembers.numberOfObjects`. This improves readability, because other programmers can easily recognize the static variable. You can also replace `CircleWithStaticMembers.numberOfObjects` with `CircleWithStaticMembers.getNumberOfObjects()`.

use class name

Tip
Use `ClassName.methodName(arguments)` to invoke a static method and `ClassName.staticVariable` to access a static variable. This improves readability, because this makes the static method and data easy to spot.

An instance method can invoke an instance or static method and access an instance or static data field. A static method can invoke a static method and access a static data field. However, a static method cannot invoke an instance method or access an instance data field, since static methods and static data fields don't belong to a particular object. The relationship between static and instance members is summarized in the following diagram:



For example, the following code is wrong.

```
public class A {
  int i = 5;
  static int k = 2;

  public static void main(String[] args) {
    int j = i; // Wrong because i is an instance variable
    m1(); // Wrong because m1() is an instance method
  }

  public void m1() {
    // Correct since instance and static variables and methods
    // can be used in an instance method
    i = i + k + m2(i, k);
  }

  public static int m2(int i, int j) {
    return (int)(Math.pow(i, j));
  }
}
```

Note that if you replace the preceding code with the following new code, the program would be fine, because the instance data field `i` and method `m1` are now accessed from an object `a` (lines 7–8):

```
 1  public class A {
 2    int i = 5;
 3    static int k = 2;
 4
 5    public static void main(String[] args) {
 6      A a = new A();
 7      int j = a.i; // OK, a.i accesses the object's instance variable
 8      a.m1(); // OK. a.m1() invokes the object's instance method
 9    }
10
11    public void m1() {
12      i = i + k + m2(i, k);
13    }
14
15    public static int m2(int i, int j) {
16      return (int)(Math.pow(i, j));
17    }
18  }
```

Design Guide

instance or static?

How do you decide whether a variable or a method should be an instance one or a static one? A variable or a method that is dependent on a specific instance of the class should be an instance variable or method. A variable or a method that is not dependent on a specific instance of the class should be a static variable or method. For example, every circle has its own radius, so the radius is dependent on a specific circle. Therefore, `radius` is an instance variable of the `Circle` class. Since the `getArea` method is dependent on a specific circle, it is an instance method. None of the methods in the `Math` class, such as `random`, `pow`, `sin`, and `cos`, is dependent on a specific instance. Therefore, these methods are static methods. The `main` method is static and can be invoked directly from a class.

Caution

common design error

It is a common design error to define an instance method that should have been defined as static. For example, the method `factorial(int n)` should be defined as static, as shown next, because it is independent of any specific instance.

```
public class Test {
  public int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i ++)
      result *= i;

    return result;
  }
}
```

(a) Wrong design

```
public class Test {
  public static int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

(b) Correct design

Check Point

9.17 Suppose that the class `F` is defined in (a). Let `f` be an instance of `F`. Which of the statements in (b) are correct?

```java
public class F {
  int i;
  static String s;

  void imethod() {
  }

  static void smethod() {
  }
}
```

(a)

```java
System.out.println(f.i);
System.out.println(f.s);
f.imethod();
f.smethod();
System.out.println(F.i);
System.out.println(F.s);
F.imethod();
F.smethod();
```

(b)

9.18 Add the `static` keyword in the place of `?` if appropriate.

```java
public class Test {
  int count;

  public ? void main(String[] args) {
    ...
  }

  public ? int getCount() {
    return count;
  }

  public ? int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

9.19 Can you invoke an instance method or reference an instance variable from a static method? Can you invoke a static method or reference a static variable from an instance method? What is wrong in the following code?

```java
public class C {
  public static void main(String[] args) {
    method1();
  }

  public void method1() {
    method2();
  }

  public static void method2() {
    System.out.println("What is radius " + c.getRadius());
  }

  Circle c = new Circle();
}
```

9.8 Visibility Modifiers

Key Point

Visibility modifiers can be used to specify the visibility of a class and its members.

package-private (or package-access)

You can use the `public` visibility modifier for classes, methods, and data fields to denote that they can be accessed from any other classes. If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. This is known as *package-private* or *package-access*.

Note

using packages

Packages can be used to organize classes. To do so, you need to add the following line as the first noncomment and nonblank statement in the program:

```
package packageName;
```

If a class is defined without the package statement, it is said to be placed in the *default package*.

Java recommends that you place classes into packages rather than using a default package. For simplicity, however, this book uses default packages. For more information on packages, see Supplement III.E, Packages.

In addition to the `public` and default visibility modifiers, Java provides the `private` and `protected` modifiers for class members. This section introduces the `private` modifier. The `protected` modifier will be introduced in Section 11.14, The `protected` Data and Methods.

The `private` modifier makes methods and data fields accessible only from within its own class. Figure 9.14 illustrates how a public, default, and private data field or method in class `C1` can be accessed from a class `C2` in the same package and from a class `C3` in a different package.

```
package p1;

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

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
package p1;

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

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```
package p2;

public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;

    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

FIGURE 9.14 The private modifier restricts access to its defining class, the default modifier restricts access to a package, and the public modifier enables unrestricted access.

If a class is not defined as public, it can be accessed only within the same package. As shown in Figure 9.15, `C1` can be accessed from `C2` but not from `C3`.

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

public class C3 {
  cannot access C1;
  can access C2;
}
```

FIGURE 9.15 A nonpublic class has package-access.

inside access

A visibility modifier specifies how data fields and methods in a class can be accessed from outside the class. There is no restriction on accessing data fields and methods from inside the class. As shown in Figure 9.16b, an object `c` of class `C` cannot access its private members, because `c` is in the `Test` class. As shown in Figure 9.16a, an object `c` of class `C` can access its private members, because `c` is defined inside its own class.

```
public class C {
  private boolean x;

  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }

  private int convert() {
    return x ? 1 : -1;
  }
}
```

(a) This is okay because object `c` is used inside the class `C`.

```
public class Test {
  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }
}
```

(b) This is wrong because `x` and `convert` are private in class `C`.

FIGURE 9.16 An object can access its private members if it is defined in its own class.

Caution

The `private` modifier applies only to the members of a class. The `public` modifier can apply to a class or members of a class. Using the modifiers `public` and `private` on local variables would cause a compile error.

Note

private constructor

In most cases, the constructor should be public. However, if you want to prohibit the user from creating an instance of a class, use a *private constructor*. For example, there is no reason to create an instance from the `Math` class, because all of its data fields and methods are static. To prevent the user from creating objects from the `Math` class, the constructor in `java.lang.Math` is defined as follows:

```
private Math() {
}
```

9.9 Data Field Encapsulation

Key Point

Making data fields private protects data and makes the class easy to maintain.

Data field encapsulation

VideoNote
Data field encapsulation

The data fields `radius` and `numberOfObjects` in the `CircleWithStaticMembers` class in Listing 9.6 can be modified directly (e.g., `c1.radius = 5` or `CircleWithStaticMembers.numberOfObjects = 10`). This is not a good practice—for two reasons:

- First, data may be tampered with. For example, `numberOfObjects` is to count the number of objects created, but it may be mistakenly set to an arbitrary value (e.g., `CircleWithStaticMembers.numberOfObjects = 10`).
- Second, the class becomes difficult to maintain and vulnerable to bugs. Suppose you want to modify the `CircleWithStaticMembers` class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the `CircleWithStaticMembers` class but also the programs that use it, because the clients may have modified the radius directly (e.g., `c1.radius = -5`).

data field encapsulation

To prevent direct modifications of data fields, you should declare the data fields private, using the `private` modifier. This is known as *data field encapsulation*.

A private data field cannot be accessed by an object from outside the class that defines the private field. However, a client often needs to retrieve and modify a data field. To make a private data field accessible, provide a *getter* method to return its value. To enable a private data field to be updated, provide a *setter* method to set a new value. A getter method is also referred to as an *accessor* and a setter to a *mutator*.

getter (or accessor)
setter (or mutator)

A getter method has the following signature:

```
public returnType getPropertyName()
```

If the `returnType` is `boolean`, the getter method should be defined as follows by convention:

boolean accessor

```
public boolean isPropertyName()
```

A setter method has the following signature:

```
public void setPropertyName(dataType propertyValue)
```

Let's create a new circle class with a private data-field radius and its associated accessor and mutator methods. The class diagram is shown in Figure 9.17. The new circle class, named `CircleWithPrivateDataFields`, is defined in Listing 9.8:

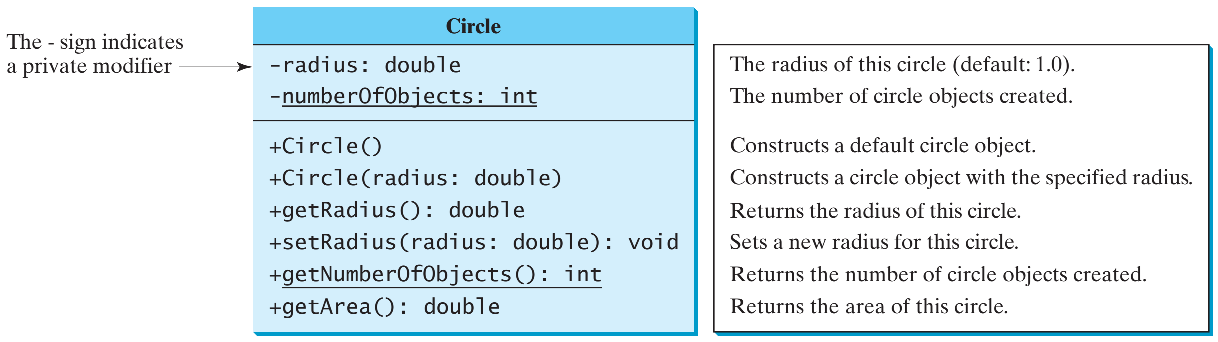

The - sign indicates a private modifier
Circle
-radius: double
-numberOfObjects: int
+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getNumberOfObjects(): int
+getArea(): double
The radius of this circle (default: 1.0).
The number of circle objects created.
Constructs a default circle object.
Constructs a circle object with the specified radius.
Returns the radius of this circle.
Sets a new radius for this circle.
Returns the number of circle objects created.
Returns the area of this circle.

FIGURE 9.17 The `Circle` class encapsulates circle properties and provides getter/setter and other methods.

LISTING 9.8 CircleWithPrivateDataFields.java

```
public class CircleWithPrivateDataFields {
  /** The radius of the circle */
  private double radius = 1;

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

  /** Construct a circle with radius 1 */
  public CircleWithPrivateDataFields() {
    numberOfObjects++;
  }

  /** Construct a circle with a specified radius */
  public CircleWithPrivateDataFields(double newRadius) {
    radius = newRadius;
    numberOfObjects++;
```

encapsulate `radius`

encapsulate `numberOfObjects`

accessor method

mutator method

accessor method

```
17    }
18
19    /** Return radius */
20    public double getRadius() {
21      return radius;
22    }
23
24    /** Set a new radius */
25    public void setRadius(double newRadius) {
26      radius = (newRadius >= 0) ? newRadius : 0;
27    }
28
29    /** Return numberOfObjects */
30    public static int getNumberOfObjects() {
31      return numberOfObjects;
32    }
33
34    /** Return the area of this circle */
35    public double getArea() {
36      return radius * radius * Math.PI;
37    }
38  }
```

The **getRadius()** method (lines 20–22) returns the radius, and the **setRadius(newRadius)** method (line 25–27) sets a new radius for the object. If the new radius is negative, **0** is set as the radius for the object. Since these methods are the only ways to read and modify the radius, you have total control over how the **radius** property is accessed. If you have to change the implementation of these methods, you don't need to change the client programs. This makes the class easy to maintain.

Listing 9.9 gives a client program that uses the **Circle** class to create a **Circle** object and modifies the radius using the **setRadius** method.

Listing 9.9 TestCircleWithPrivateDataFields.java

invoke public method

invoke public method

invoke public method

```
public class TestCircleWithPrivateDataFields {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 5.0
    CircleWithPrivateDataFields myCircle =
      new CircleWithPrivateDataFields(5.0);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    // Increase myCircle's radius by 10%
    myCircle.setRadius(myCircle.getRadius() * 1.1);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    System.out.println("The number of objects created is "
      + CircleWithPrivateDataFields.getNumberOfObjects());
  }
}
```

The data field **radius** is declared private. Private data can be accessed only within their defining class, so you cannot use **myCircle.radius** in the client program. A compile error would occur if you attempted to access private data from a client.

Since **numberOfObjects** is private, it cannot be modified. This prevents tampering. For example, the user cannot set **numberOfObjects** to **100**. The only way to make it **100** is to create **100** objects of the **Circle** class.

Suppose you combined `TestCircleWithPrivateDataFields` and `Circle` into one class by moving the `main` method in `TestCircleWithPrivateDataFields` into `Circle`. Could you use `myCircle.radius` in the `main` method? See Checkpoint Question 9.22 for the answer.

Design Guide

To prevent data from being tampered with and to make the class easy to maintain, declare data fields private.

Check Point

9.20 What is an accessor method? What is a mutator method? What are the naming conventions for accessor methods and mutator methods?

9.21 What are the benefits of data field encapsulation?

9.22 In the following code, `radius` is private in the `Circle` class, and `myCircle` is an object of the `Circle` class. Does the highlighted code cause any problems? If so, explain why.

```java
public class Circle {
  private double radius = 1;

  /** Find the area of this circle */
  public double getArea() {
    return radius * radius * Math.PI;
  }

  public static void main(String[] args) {
    Circle myCircle = new Circle();
    System.out.println("Radius is " + myCircle.radius);
  }
}
```

9.10 Passing Objects to Methods

Key Point

Passing an object to a method is to pass the reference of the object.

You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object. The following code passes the `myCircle` object as an argument to the `printCircle` method:

```java
public class Test {
  public static void main(String[] args) {
    // CircleWithPrivateDataFields is defined in Listing 9.8
    CircleWithPrivateDataFields myCircle = new
      CircleWithPrivateDataFields(5.0);
    printCircle(myCircle);
  }

  public static void printCircle(CircleWithPrivateDataFields c) {
    System.out.println("The area of the circle of radius "
      + c.getRadius() + " is " + c.getArea());
  }
}
```

pass an object

pass-by-value

Java uses exactly one mode of passing arguments: pass-by-value. In the preceding code, the value of `myCircle` is passed to the `printCircle` method. This value is a reference to a `Circle` object.

The program in Listing 9.10 demonstrates the difference between passing a primitive type value and passing a reference value.

LISTING 9.10 TestPassObject.java

pass object

object parameter

```
 1 public class TestPassObject {
 2   /** Main method */
 3   public static void main(String[] args) {
 4     // Create a Circle object with radius 1
 5     CircleWithPrivateDataFields myCircle =
 6       new CircleWithPrivateDataFields(1);
 7
 8     // Print areas for radius 1, 2, 3, 4, and 5.
 9     int n = 5;
10     printAreas(myCircle, n);
11
12     // See myCircle.radius and times
13     System.out.println("\n" + "Radius is " + myCircle.getRadius());
14     System.out.println("n is " + n);
15   }
16
17   /** Print a table of areas for radius */
18   public static void printAreas(
19       CircleWithPrivateDataFields c, int times) {
20     System.out.println("Radius \t\tArea");
21     while (times >= 1) {
22       System.out.println(c.getRadius() + "\t\t" + c.getArea());
23       c.setRadius(c.getRadius() + 1);
24       times--;
25     }
26   }
27 }
```

```
Radius      Area
1.0         3.141592653589793
2.0         12.566370614359172
3.0         29.274333882308138
4.0         50.26548245743669
5.0         79.53981633974483
Radius is 6.0
n is 5
```

The `CircleWithPrivateDataFields` class is defined in Listing 9.8. The program passes a `CircleWithPrivateDataFields` object `myCircle` and an integer value from `n` to invoke `printAreas(myCircle, n)` (line 10), which prints a table of areas for radii `1`, `2`, `3`, `4`, `5`, as shown in the sample output.

Figure 9.18 shows the call stack for executing the methods in the program. Note that the objects are stored in a heap (see Section 7.6).

When passing an argument of a primitive data type, the value of the argument is passed. In this case, the value of `n` (`5`) is passed to `times`. Inside the `printAreas` method, the content of `times` is changed; this does not affect the content of `n`.

pass-by-sharing

When passing an argument of a reference type, the reference of the object is passed. In this case, `c` contains a reference for the object that is also referenced via `myCircle`. Therefore, changing the properties of the object through `c` inside the `printAreas` method has the same effect as doing so outside the method through the variable `myCircle`. Pass-by-value on references can be best described semantically as *pass-by-sharing*; that is, the object referenced in the method is the same as the object being passed.

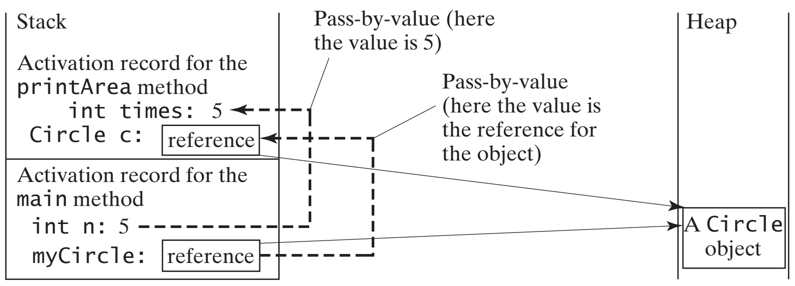

Stack
Activation record for the printArea method
int times: 5
Circle c: reference
Activation record for the main method
int n: 5
myCircle: reference
Pass-by-value (here the value is 5)
Pass-by-value (here the value is the reference for the object)
Heap
A Circle object

FIGURE 9.18 The value of `n` is passed to `times`, and the reference to `myCircle` is passed to `c` in the `printAreas` method.

9.23 Describe the difference between passing a parameter of a primitive type and passing a parameter of a reference type. Show the output of the following programs:

Check Point

```
public class Test {
  public static void main(String[] args) {
    Count myCount = new Count();
    int times = 0;

    for (int i = 0; i < 100; i++)
      increment(myCount, times);

    System.out.println("count is " + myCount.count);
    System.out.println("times is " + times);
  }

  public static void increment(Count c, int times) {
    c.count++;
    times++;
  }
}
```

```
public class Count {
  public int count;

  public Count(int c) {
    count = c;
  }

  public Count() {
    count = 1;
  }
}
```

9.24 Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    Circle circle1 = new Circle(1);
    Circle circle2 = new Circle(2);

    swap1(circle1, circle2);
    System.out.println("After swap1: circle1 = " +
      circle1.radius + " circle2 = " + circle2.radius);

    swap2(circle1, circle2);
    System.out.println("After swap2: circle1 = " +
      circle1.radius + " circle2 = " + circle2.radius);
  }

  public static void swap1(Circle x, Circle y) {
    Circle temp = x;
    x = y;
    y = temp;
  }
  public static void swap2(Circle x, Circle y) {
    double temp = x.radius;
    x.radius = y.radius;
    y.radius = temp;
  }
}

class Circle {
  double radius;

  Circle(double newRadius) {
    radius = newRadius;
  }
}
```

9.25 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    int[] a = {1, 2};
    swap(a[0], a[1]);
    System.out.println("a[0] = " + a[0]
      + " a[1] = " + a[1]);
  }

  public static void swap(int n1, int n2) {
    int temp = n1;
    n1 = n2;
    n2 = temp;
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    int[] a = {1, 2};
    swap(a);
    System.out.println("a[0] = " + a[0]
      + " a[1] = " + a[1]);
  }

  public static void swap(int[] a) {
    int temp = a[0];
    a[0] = a[1];
    a[1] = temp;
  }
}
```

(b)

```
public class Test {
  public static void main(String[] args) {
    T t = new T();
    swap(t);
    System.out.println("e1 = " + t.e1
      + " e2 = " + t.e2);
  }

  public static void swap(T t) {
    int temp = t.e1;
    t.e1 = t.e2;
    t.e2 = temp;
  }
}

class T {
  int e1 = 1;
  int e2 = 2;
}
```

(c)

```
public class Test {
  public static void main(String[] args) {
    T t1 = new T();
    T t2 = new T();
    System.out.println("t1's i = " +
      t1.i + " and j = " + t1.j);
    System.out.println("t2's i = " +
      t2.i + " and j = " + t2.j);
  }
}

class T {
  static int i = 0;
  int j = 0;

  T() {
    i++;
    j = 1;
  }
}
```

(d)

9.26 What is the output of the following programs?

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = null;
    m1(date);
    System.out.println(date);
  }

  public static void m1(Date date) {
    date = new Date();
  }
}
```

(a)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date = new Date(7654321);
  }
}
```

(b)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date.setTime(7654321);
  }
}
```

(c)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date = null;
  }
}
```

(d)

9.11 Array of Objects

An array can hold objects as well as primitive type values.

Key Point

Chapter 7, Single-Dimensional Arrays, described how to create arrays of primitive type elements. You can also create arrays of objects. For example, the following statement declares and creates an array of ten `Circle` objects:

```
Circle[] circleArray = new Circle[10];
```

To initialize `circleArray`, you can use a `for` loop like this one:

```
for (int i = 0; i < circleArray.length; i++) {
  circleArray[i] = new Circle();
}
```

An array of objects is actually an *array of reference variables*. So, invoking `circleArray[1].getArea()` involves two levels of referencing, as shown in Figure 9.19. `circleArray` references the entire array; `circleArray[1]` references a `Circle` object.

Note

When an array of objects is created using the `new` operator, each element in the array is a reference variable with a default value of `null`.

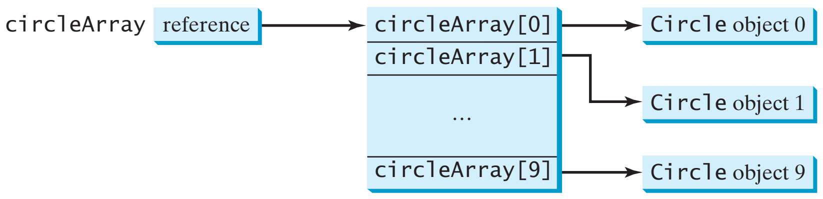


FIGURE 9.19 In an array of objects, an element of the array contains a reference to an object.

Listing 9.11 gives an example that demonstrates how to use an array of objects. The program summarizes the areas of an array of circles. The program creates `circleArray`, an array composed of five `Circle` objects; it then initializes circle radii with random values and displays the total area of the circles in the array.

LISTING 9.11 TotalArea.java

array of objects

return array of objects

pass array of objects

```java
public class TotalArea {
  /** Main method */
  public static void main(String[] args) {
    // Declare circleArray
    CircleWithPrivateDataFields[] circleArray;

    // Create circleArray
    circleArray = createCircleArray();

    // Print circleArray and total areas of the circles
    printCircleArray(circleArray);
  }

  /** Create an array of Circle objects */
  public static CircleWithPrivateDataFields[] createCircleArray() {
    CircleWithPrivateDataFields[] circleArray =
      new CircleWithPrivateDataFields[5];

    for (int i = 0; i < circleArray.length; i++) {
      circleArray[i] =
        new CircleWithPrivateDataFields(Math.random() * 100);
    }

    // Return Circle array
    return circleArray;
  }

  /** Print an array of circles and their total area */
  public static void printCircleArray(
      CircleWithPrivateDataFields[] circleArray) {
    System.out.printf("%-30s%-15s\n", "Radius", "Area");
    for (int i = 0; i < circleArray.length; i++) {
      System.out.printf("%-30f%-15f\n", circleArray[i].getRadius(),
        circleArray[i].getArea());
    }

    System.out.println("-------------------------------------------");

    // Compute and display the result
    System.out.printf("%-30s%-15f\n", "The total area of circles is",
     sum(circleArray) );
  }
43
44   /** Add circle areas */
45   public static double sum(CircleWithPrivateDataFields[] circleArray) {
46     // Initialize sum
47     double sum = 0;
48
49     // Add areas to sum
50     for (int i = 0; i < circleArray.length; i++)
51       sum += circleArray[i].getArea();
52
53     return sum;
54   }
55 }
```

pass array of objects

```
Radius                      Area
70.577708                   15649.941866
44.152266                   6124.291736
24.867853                   1942.792644
 5.680718                   101.380949
36.734246                   4239.280350
----------------------------------------------
The total area of circles is 28056.687544
```

The program invokes **`createCircleArray()`** (line 8) to create an array of five circle objects. Several circle classes were introduced in this chapter. This example uses the **`CircleWithPrivateDataFields`** class introduced in Section 9.9, Data Field Encapsulation.

The circle radii are randomly generated using the **`Math.random()`** method (line 21). The **`createCircleArray`** method returns an array of **`CircleWithPrivateDataFields`** objects (line 25). The array is passed to the **`printCircleArray`** method, which displays the radius and area of each circle and the total area of the circles.

The sum of the circle areas is computed by invoking the **`sum`** method (line 41), which takes the array of **`CircleWithPrivateDataFields`** objects as the argument and returns a **`double`** value for the total area.

Check Point

9.27 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    java.util.Date[] dates = new java.util.Date[10];
    System.out.println(dates[0]);
    System.out.println(dates[0].toString());
  }
}
```

9.12 Immutable Objects and Classes

Key Point

You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed.

Normally, you create an object and allow its contents to be changed later. However, occasionally it is desirable to create an object whose contents cannot be changed once the object has been created. We call such an object as *immutable object* and its class as *immutable class*. The **`String`** class, for example, is immutable. If you deleted the setter method in the **`CircleWithPrivateDataFields`** class in Listing 9.9, the class would be immutable, because radius is private and cannot be changed without a setter method.

immutable object
immutable class

If a class is immutable, then all its data fields must be private and it cannot contain public setter methods for any data fields. A class with all private data fields and no mutators is not necessarily immutable. For example, the following `Student` class has all private data fields and no setter methods, but it is not an immutable class.

Student class

```
public class Student {
  private int id;
  private String name;
  private java.util.Date dateCreated;

  public Student(int ssn, String newName) {
    id = ssn;
    name = newName;
    dateCreated = new java.util.Date();
  }

  public int getId() {
    return id;
  }

  public String getName() {
    return name;
  }

  public java.util.Date getDateCreated() {
    return dateCreated;
  }
}
```

As shown in the following code, the data field `dateCreated` is returned using the `getDateCreated()` method. This is a reference to a `Date` object. Through this reference, the content for `dateCreated` can be changed.

```
public class Test {
  public static void main(String[] args) {
    Student student = new Student(111223333, "John");
    java.util.Date dateCreated = student.getDateCreated();
    dateCreated.setTime(200000); // Now dateCreated field is changed!
  }
}
```

For a class to be immutable, it must meet the following requirements:

- All data fields must be private.
- There can't be any mutator methods for data fields.
- No accessor methods can return a reference to a data field that is mutable.

Interested readers may refer to Supplement III.U for an extended discussion on immutable objects.

Check Point

9.28 If a class contains only private data fields and no setter methods, is the class immutable?

9.29 If all the data fields in a class are private and of primitive types, and the class doesn't contain any setter methods, is the class immutable?

9.30 Is the following class immutable?

```
public class A {
  private int[] values;

  public int[] getValues() {
      return values;
    }
  }
```

9.13 The Scope of Variables

Key Point

The scope of instance and static variables is the entire class, regardless of where the variables are declared.

Section 6.9 discussed local variables and their scope rules. Local variables are declared and used inside a method locally. This section discusses the scope rules of all the variables in the context of a class.

class's variables

Instance and static variables in a class are referred to as the *class's variables* or *data fields*. A variable defined inside a method is referred to as a *local variable*. The scope of a class's variables is the entire class, regardless of where the variables are declared. A class's variables and methods can appear in any order in the class, as shown in Figure 9.20a. The exception is when a data field is initialized based on a reference to another data field. In such cases, the other data field must be declared first, as shown in Figure 9.20b. For consistency, this book declares data fields at the beginning of the class.

```
public class Circle {
  public double findArea() {
    return radius * radius * Math.PI;
  }

  private double radius = 1;
}
```

(a) The variable `radius` and method `findArea()` can be declared in any order.

```
public class F {
  private int i ;
  private int j = i + 1;
}
```

(b) `i` has to be declared before `j` because `j`'s initial value is dependent on `i`.

FIGURE 9.20 Members of a class can be declared in any order, with one exception.

You can declare a class's variable only once, but you can declare the same variable name in a method many times in different nonnesting blocks.

hidden variables

If a local variable has the same name as a class's variable, the local variable takes precedence and the class's variable with the same name is *hidden*. For example, in the following program, `x` is defined both as an instance variable and as a local variable in the method.

```
public class F {
  private int x = 0; // Instance variable
  private int y = 0;

  public F() {
  }

  public void p() {
    int x = 1; // Local variable
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

What is the output for `f.p()`, where `f` is an instance of `F`? The output for `f.p()` is `1` for `x` and `0` for `y`. Here is why:

- `x` is declared as a data field with the initial value of `0` in the class, but it is also declared in the method `p()` with an initial value of `1`. The latter `x` is referenced in the `System.out.println` statement.
- `y` is declared outside the method `p()`, but `y` is accessible inside the method.

Tip
To avoid confusion and mistakes, do not use the names of instance or static variables as local variable names, except for method parameters.

Check Point

9.31 What is the output of the following program?

```
public class Test {
  private static int i = 0;
  private static int j = 0;

  public static void main(String[] args) {
    int i = 2;
    int k = 3;

    {
      int j = 3;
      System.out.println("i + j is " + i + j);
    }

    k = i + j;
    System.out.println("k is " + k);
    System.out.println("j is " + j);
  }
}
```

9.14 The **this** Reference

Key Point

The keyword **this** *refers to the object itself. It can also be used inside a constructor to invoke another constructor of the same class.*

this keyword

The **this** *keyword* is the name of a reference that an object can use to refer to itself. You can use the **this** keyword to reference the object's instance members. For example, the following code in (a) uses **this** to reference the object's **radius** and invokes its **getArea()** method explicitly. The **this** reference is normally omitted, as shown in (b). However, the **this** reference is needed to reference hidden data fields or invoke an overloaded constructor.

```
public class Circle {
  private double radius;

  ...

  public double getArea() {
  return this.radius * this.radius * Math.PI;
  }

  public String toString() {
    return "radius: " + this.radius
       + "area: " + this.getArea() ;
  }
}
```

(a)

Equivalent

```
public class Circle {
  private double radius;

  ...

  public double getArea() {
    return radius * radius * Math.PI;
  }

  public String toString() {
    return "radius: " + radius
       + "area: " + getArea() ;
  }
}
```

(b)

9.14.1 Using **this** to Reference Hidden Data Fields

hidden data fields

The **this** keyword can be used to reference a class's *hidden data fields*. For example, a data-field name is often used as the parameter name in a setter method for the data field. In this case, the data field is hidden in the setter method. You need to reference the hidden data-field name in the method in order to set a new value to it. A hidden static variable can be accessed simply by using the `ClassName.staticVariable` reference. A hidden instance variable can be accessed by using the keyword `this`, as shown in Figure 9.21a.

```
public class F {
  private int i = 5;
  private static double k = 0;

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

  public static void setK(double k) {
    F.k = k;
  }

  // Other methods omitted
}
```

(a)

```
Suppose that f1 and f2 are two objects of F.

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2

Invoking F.setK(33) is to execute
   F.k = 33. setK is a static method
```

(b)

FIGURE 9.21 The keyword `this` refers to the calling object that invokes the method.

The `this` keyword gives us a way to reference the object that invokes an instance method. To invoke `f1.setI(10)`, `this.i = i` is executed, which assigns the value of parameter `i` to the data field `i` of this calling object `f1`. The keyword `this` refers to the object that invokes the instance method `setI`, as shown in Figure 9.21b. The line `F.k = k` means that the value in parameter `k` is assigned to the static data field `k` of the class, which is shared by all the objects of the class.

9.14.2 Using **this** to Invoke a Constructor

The `this` keyword can be used to invoke another constructor of the same class. For example, you can rewrite the `Circle` class as follows:

```
public class Circle {
  private double radius;

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

  public Circle() {
    this(1.0);
  }

  ...
}
```

The `this` keyword is used to reference the hidden data field radius of the object being constructed.

The `this` keyword is used to invoke another constructor.

The line `this(1.0)` in the second constructor invokes the first constructor with a `double` value argument.

Note

Java requires that the `this(arg-list)` statement appear first in the constructor before any other executable statements.

Tip

If a class has multiple constructors, it is better to implement them using `this(arg-list)` as much as possible. In general, a constructor with no or fewer arguments can invoke a constructor with more arguments using `this(arg-list)`. This syntax often simplifies coding and makes the class easier to read and to maintain.

Check Point

9.32 Describe the role of the `this` keyword.

9.33 What is wrong in the following code?

```java
public class C {
  private int p;

  public C() {
    System.out.println("C's no-arg constructor invoked");
    this(0);
  }

  public C(int p) {
    p = p;
  }

  public void setP(int p) {
    p = p;
  }
}
```

9.34 What is wrong in the following code?

```java
public class Test {
  private int id;

  public void m1() {
    this.id = 45;
  }

  public void m2() {
    Test.id = 45;
  }
}
```

Key Terms

action	322
anonymous object	331
attribute	322
behavior	322
class	322
class's variable	355
client	325
constructor	322
data field	322
data field encapsulation	344
default constructor	329
dot operator (`.`)	330
getter (or accessor)	345
instance	322
instance method	331
instance variable	331
instantiation	322
immutable class	353
immutable object	353
no-arg constructor	329
`null` value	331
object	322
object-oriented programming (OOP)	322
package-private (or package-access)	342
private constructor	344
property	322
public class	325
reference type	330
reference variable	330
setter (or mutator)	345
state	322
static method	337
static variable	337
`this` keyword	356
Unified Modeling Language (UML)	323

Chapter Summary

1. A *class* is a template for *objects*. It defines the *properties* of objects and provides *constructors* for creating objects and methods for manipulating them.

2. A class is also a data type. You can use it to declare object *reference variables*. An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored.

3. An object is an *instance* of a class. You use the `new` operator to create an object, and the *dot operator* (`.`) to access members of that object through its reference variable.

4. An *instance variable* or *method* belongs to an instance of a class. Its use is associated with individual instances. A *static variable* is a variable shared by all instances of the same class. A *static method* is a method that can be invoked without using instances.

5. Every instance of a class can access the class's static variables and methods. For clarity, however, it is better to invoke static variables and methods using `ClassName.variable` and `ClassName.method`.

6. Visibility modifiers specify how the class, method, and data are accessed. A `public` class, method, or data is accessible to all clients. A `private` method or data is accessible only inside the class.

7. You can provide a getter (accessor) method or a setter (mutator) method to enable clients to see or modify the data.

8. A getter method has the signature `public returnType getPropertyName()`. If the `returnType` is `boolean`, the `get` method should be defined as `public boolean isPropertyName()`. A setter method has the signature `public void setPropertyName(dataType propertyValue)`.

9. All parameters are passed to methods using pass-by-value. For a parameter of a primitive type, the actual value is passed; for a parameter of a *reference type*, the reference for the object is passed.

10. A Java array is an object that can contain primitive type values or object type values. When an array of objects is created, its elements are assigned the default value of `null`.

11. Once it is created, an *immutable object* cannot be modified. To prevent users from modifying an object, you can define *immutable classes*.

12. The scope of instance and static variables is the entire class, regardless of where the variables are declared. Instance and static variables can be declared anywhere in the class. For consistency, they are declared at the beginning of the class in this book.

13. The keyword `this` can be used to refer to the calling object. It can also be used inside a constructor to invoke another constructor of the same class.

Quiz

Answer the quiz for this chapter online at www.cs.armstrong.edu/liang/intro10e/quiz.html.

MyProgrammingLab™

Programming Exercises

three objectives

Pedagogical Note

The exercises in Chapters 9–13 help you achieve three objectives:

- Design classes and draw UML class diagrams.
- Implement classes from the UML.
- Use classes to develop applications.

Students can download solutions for the UML diagrams for the even-numbered exercises from the Companion Website, and instructors can download all solutions from the same site.

Sections 9.2–9.5

9.1 (*The `Rectangle` class*) Following the example of the `Circle` class in Section 9.2, design a class named `Rectangle` to represent a rectangle. The class contains:

- Two `double` data fields named `width` and `height` that specify the width and height of the rectangle. The default values are `1` for both `width` and `height`.
- A no-arg constructor that creates a default rectangle.
- A constructor that creates a rectangle with the specified `width` and `height`.
- A method named `getArea()` that returns the area of this rectangle.
- A method named `getPerimeter()` that returns the perimeter.

Draw the UML diagram for the class and then implement the class. Write a test program that creates two `Rectangle` objects—one with width `4` and height `40` and the other with width `3.5` and height `35.9`. Display the width, height, area, and perimeter of each rectangle in this order.

9.2 (*The `Stock` class*) Following the example of the `Circle` class in Section 9.2, design a class named `Stock` that contains:

- A string data field named `symbol` for the stock's symbol.
- A string data field named `name` for the stock's name.
- A `double` data field named `previousClosingPrice` that stores the stock price for the previous day.
- A `double` data field named `currentPrice` that stores the stock price for the current time.
- A constructor that creates a stock with the specified symbol and name.
- A method named `getChangePercent()` that returns the percentage changed from `previousClosingPrice` to `currentPrice`.

Draw the UML diagram for the class and then implement the class. Write a test program that creates a `Stock` object with the stock symbol `ORCL`, the name `Oracle Corporation`, and the previous closing price of `34.5`. Set a new current price to `34.35` and display the price-change percentage.

Section 9.6

***9.3** (*Use the `Date` class*) Write a program that creates a `Date` object, sets its elapsed time to `10000`, `100000`, `1000000`, `10000000`, `100000000`, `1000000000`, `10000000000`, and `100000000000`, and displays the date and time using the `toString()` method, respectively.

***9.4** (*Use the `Random` class*) Write a program that creates a `Random` object with seed `1000` and displays the first 50 random integers between `0` and `100` using the `nextInt(100)` method.

***9.5** (*Use the `GregorianCalendar` class*) Java API has the `GregorianCalendar` class in the `java.util` package, which you can use to obtain the year, month, and day of a date. The no-arg constructor constructs an instance for the current date, and the methods `get(GregorianCalendar.YEAR)`, `get(GregorianCalendar.MONTH)`, and `get(GregorianCalendar.DAY_OF_MONTH)` return the year, month, and day. Write a program to perform two tasks:

- Display the current year, month, and day.
- The `GregorianCalendar` class has the `setTimeInMillis(long)`, which can be used to set a specified elapsed time since January 1, 1970. Set the value to `1234567898765L` and display the year, month, and day.

Sections 9.7–9.9

***9.6** (*Stopwatch*) Design a class named `StopWatch`. The class contains:

- Private data fields `startTime` and `endTime` with getter methods.
- A no-arg constructor that initializes `startTime` with the current time.
- A method named `start()` that resets the `startTime` to the current time.
- A method named `stop()` that sets the `endTime` to the current time.
- A method named `getElapsedTime()` that returns the elapsed time for the stopwatch in milliseconds.

Draw the UML diagram for the class and then implement the class. Write a test program that measures the execution time of sorting 100,000 numbers using selection sort.

9.7 (*The `Account` class*) Design a class named `Account` that contains:

- A private `int` data field named `id` for the account (default `0`).
- A private `double` data field named `balance` for the account (default `0`).
- A private `double` data field named `annualInterestRate` that stores the current interest rate (default `0`). Assume all accounts have the same interest rate.
- A private `Date` data field named `dateCreated` that stores the date when the account was created.
- A no-arg constructor that creates a default account.
- A constructor that creates an account with the specified id and initial balance.
- The accessor and mutator methods for `id`, `balance`, and `annualInterestRate`.
- The accessor method for `dateCreated`.
- A method named `getMonthlyInterestRate()` that returns the monthly interest rate.
- A method named `getMonthlyInterest()` that returns the monthly interest.
- A method named `withdraw` that withdraws a specified amount from the account.
- A method named `deposit` that deposits a specified amount to the account.

Draw the UML diagram for the class and then implement the class. (*Hint*: The method `getMonthlyInterest()` is to return monthly interest, not the interest rate. Monthly interest is `balance * monthlyInterestRate`. `monthlyInterestRate` is `annualInterestRate / 12`. Note that `annualInterestRate` is a percentage, e.g., like 4.5%. You need to divide it by 100.)

Write a test program that creates an `Account` object with an account ID of 1122, a balance of $20,000, and an annual interest rate of 4.5%. Use the `withdraw` method to withdraw $2,500, use the `deposit` method to deposit $3,000, and print the balance, the monthly interest, and the date when this account was created.

VideoNote
The Fan class

9.8 (*The* **Fan** *class*) Design a class named **Fan** to represent a fan. The class contains:

- Three constants named **SLOW**, **MEDIUM**, and **FAST** with the values **1**, **2**, and **3** to denote the fan speed.
- A private **int** data field named **speed** that specifies the speed of the fan (the default is **SLOW**).
- A private **boolean** data field named **on** that specifies whether the fan is on (the default is **false**).
- A private **double** data field named **radius** that specifies the radius of the fan (the default is **5**).
- A string data field named **color** that specifies the color of the fan (the default is **blue**).
- The accessor and mutator methods for all four data fields.
- A no-arg constructor that creates a default fan.
- A method named **toString()** that returns a string description for the fan. If the fan is on, the method returns the fan speed, color, and radius in one combined string. If the fan is not on, the method returns the fan color and radius along with the string "fan is off" in one combined string.

Draw the UML diagram for the class and then implement the class. Write a test program that creates two **Fan** objects. Assign maximum speed, radius **10**, color **yellow**, and turn it on to the first object. Assign medium speed, radius **5**, color **blue**, and turn it off to the second object. Display the objects by invoking their **toString** method.

****9.9** (*Geometry: n-sided regular polygon*) In an *n*-sided regular polygon, all sides have the same length and all angles have the same degree (i.e., the polygon is both equilateral and equiangular). Design a class named **RegularPolygon** that contains:

- A private **int** data field named **n** that defines the number of sides in the polygon with default value **3**.
- A private **double** data field named **side** that stores the length of the side with default value **1**.
- A private **double** data field named **x** that defines the *x*-coordinate of the polygon's center with default value **0**.
- A private **double** data field named **y** that defines the *y*-coordinate of the polygon's center with default value **0**.
- A no-arg constructor that creates a regular polygon with default values.
- A constructor that creates a regular polygon with the specified number of sides and length of side, centered at (**0**, **0**).
- A constructor that creates a regular polygon with the specified number of sides, length of side, and *x*- and *y*-coordinates.
- The accessor and mutator methods for all data fields.
- The method **getPerimeter()** that returns the perimeter of the polygon.
- The method **getArea()** that returns the area of the polygon. The formula for computing the area of a regular polygon is $Area = \dfrac{n \times s^2}{4 \times \tan\left(\dfrac{\pi}{n}\right)}$.

Draw the UML diagram for the class and then implement the class. Write a test program that creates three **RegularPolygon** objects, created using the no-arg constructor, using **RegularPolygon(6, 4)**, and using **RegularPolygon(10, 4, 5.6, 7.8)**. For each object, display its perimeter and area.

***9.10** (*Algebra: quadratic equations*) Design a class named `QuadraticEquation` for a quadratic equation $ax^2 + bx + x = 0$. The class contains:

- Private data fields `a`, `b`, and `c` that represent three coefficients.
- A constructor for the arguments for `a`, `b`, and `c`.
- Three getter methods for `a`, `b`, and `c`.
- A method named `getDiscriminant()` that returns the discriminant, which is $b^2 - 4ac$.
- The methods named `getRoot1()` and `getRoot2()` for returning two roots of the equation

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

These methods are useful only if the discriminant is nonnegative. Let these methods return `0` if the discriminant is negative.

Draw the UML diagram for the class and then implement the class. Write a test program that prompts the user to enter values for *a*, *b*, and *c* and displays the result based on the discriminant. If the discriminant is positive, display the two roots. If the discriminant is 0, display the one root. Otherwise, display "The equation has no roots." See Programming Exercise 3.1 for sample runs.

***9.11** (*Algebra:* 2 × 2 *linear equations*) Design a class named `LinearEquation` for a 2 × 2 system of linear equations:

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \qquad x = \frac{ed - bf}{ad - bc} \qquad y = \frac{af - ec}{ad - bc}$$

The class contains:

- Private data fields `a`, `b`, `c`, `d`, `e`, and `f`.
- A constructor with the arguments for `a`, `b`, `c`, `d`, `e`, and `f`.
- Six getter methods for `a`, `b`, `c`, `d`, `e`, and `f`.
- A method named `isSolvable()` that returns true if $ad - bc$ is not 0.
- Methods `getX()` and `getY()` that return the solution for the equation.

Draw the UML diagram for the class and then implement the class. Write a test program that prompts the user to enter `a`, `b`, `c`, `d`, `e`, and `f` and displays the result. If $ad - bc$ is 0, report that "The equation has no solution." See Programming Exercise 3.3 for sample runs.

****9.12** (*Geometry: intersecting point*) Suppose two line segments intersect. The two endpoints for the first line segment are (`x1`, `y1`) and (`x2`, `y2`) and for the second line segment are (`x3`, `y3`) and (`x4`, `y4`). Write a program that prompts the user to enter these four endpoints and displays the intersecting point. As discussed in Programming Exercise 3.25, the intersecting point can be found by solving a linear equation. Use the `LinearEquation` class in Programming Exercise 9.11 to solve this equation. See Programming Exercise 3.25 for sample runs.

****9.13** (*The `Location` class*) Design a class named `Location` for locating a maximal value and its location in a two-dimensional array. The class contains public data fields `row`, `column`, and `maxValue` that store the maximal value and its indices in a two-dimensional array with `row` and `column` as `int` types and `maxValue` as a `double` type.

Write the following method that returns the location of the largest element in a two-dimensional array:

```java
public static Location locateLargest(double[][] a)
```

The return value is an instance of `Location`. Write a test program that prompts the user to enter a two-dimensional array and displays the location of the largest element in the array. Here is a sample run:

```
Enter the number of rows and columns in the array:  3 4 ↵Enter
Enter the array:
23.5 35 2 10 ↵Enter
4.5 3 45 3.5 ↵Enter
35 44 5.5 9.6 ↵Enter
The location of the largest element is 45 at (1, 2)
```

CHAPTER

10

OBJECT-ORIENTED THINKING

Objectives

- To apply class abstraction to develop software (§10.2).
- To explore the differences between the procedural paradigm and object-oriented paradigm (§10.3).
- To discover the relationships between classes (§10.4).
- To design programs using the object-oriented paradigm (§§10.5–10.6).
- To create objects for primitive values using the wrapper classes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, and `Boolean`) (§10.7).
- To simplify programming using automatic conversion between primitive types and wrapper class types (§10.8).
- To use the `BigInteger` and `BigDecimal` classes for computing very large numbers with arbitrary precisions (§10.9).
- To use the `String` class to process immutable strings (§10.10).
- To use the `StringBuilder` and `StringBuffer` classes to process mutable strings (§10.11).

10.1 Introduction

Key Point

The focus of this chapter is on class design and explores the differences between procedural programming and object-oriented programming.

The preceding chapter introduced objects and classes. You learned how to define classes, create objects, and use objects from several classes in the Java API (e.g., `Circle`, `Date`, `Random`, and `Point2D`). This book's approach is to teach problem solving and fundamental programming techniques before object-oriented programming. This chapter shows how procedural and object-oriented programming differ. You will see the benefits of object-oriented programming and learn to use it effectively.

Our focus here is on class design. We will use several examples to illustrate the advantages of the object-oriented approach. The examples involve designing new classes and using them in applications and introducing new classes in the Java API.

10.2 Class Abstraction and Encapsulation

Key Point

Class abstraction is the separation of class implementation from the use of a class. The details of implementation are encapsulated and hidden from the user. This is known as class encapsulation.

In Chapter 6, you learned about method abstraction and used it in stepwise refinement. Java provides many levels of abstraction, and *class abstraction* separates class implementation from how the class is used. The creator of a class describes the functions of the class and lets the user know how the class can be used. The collection of methods and fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the *class's contract*. As shown in Figure 10.1, the user of the class does not need to know how the class is implemented. The details of implementation are encapsulated and hidden from the user. This is called *class encapsulation*. For example, you can create a `Circle` object and find the area of the circle without knowing how the area is computed. For this reason, a class is also known as an *abstract data type* (ADT).

class abstraction

class's contract

class encapsulation

abstract data type

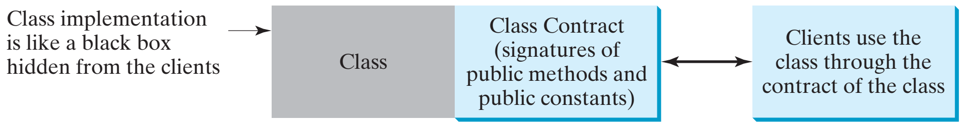

Class implementation is like a black box hidden from the clients
Class
Class Contract (signatures of public methods and public constants)
Clients use the class through the contract of the class

FIGURE 10.1 Class abstraction separates class implementation from the use of the class.

Class abstraction and encapsulation are two sides of the same coin. Many real-life examples illustrate the concept of class abstraction. Consider, for instance, building a computer system. Your personal computer has many components—a CPU, memory, disk, motherboard, fan, and so on. Each component can be viewed as an object that has properties and methods. To get the components to work together, you need know only how each component is used and how it interacts with the others. You don't need to know how the components work internally. The internal implementation is encapsulated and hidden from you. You can build a computer without knowing how a component is implemented.

The computer-system analogy precisely mirrors the object-oriented approach. Each component can be viewed as an object of the class for the component. For example, you might have a class that models all kinds of fans for use in a computer, with properties such as fan size and speed and methods such as start and stop. A specific fan is an instance of this class with specific property values.

As another example, consider getting a loan. A specific loan can be viewed as an object of a `Loan` class. The interest rate, loan amount, and loan period are its data properties, and computing the monthly payment and total payment are its methods. When you buy a car, a loan object is created by instantiating the class with your loan interest rate, loan amount, and loan period. You can then use the methods to find the monthly payment and total payment of your loan. As a user of the `Loan` class, you don't need to know how these methods are implemented.

VideoNote
The Loan class

Listing 2.9, ComputeLoan.java, presented a program for computing loan payments. That program cannot be reused in other programs because the code for computing the payments is in the `main` method. One way to fix this problem is to define static methods for computing the monthly payment and total payment. However, this solution has limitations. Suppose you wish to associate a date with the loan. There is no good way to tie a date with a loan without using objects. The traditional procedural programming paradigm is action-driven, and data are separated from actions. The object-oriented programming paradigm focuses on objects, and actions are defined along with the data in objects. To tie a date with a loan, you can define a loan class with a date along with the loan's other properties as data fields. A loan object now contains data and actions for manipulating and processing data, and the loan data and actions are integrated in one object. Figure 10.2 shows the UML class diagram for the `Loan` class.

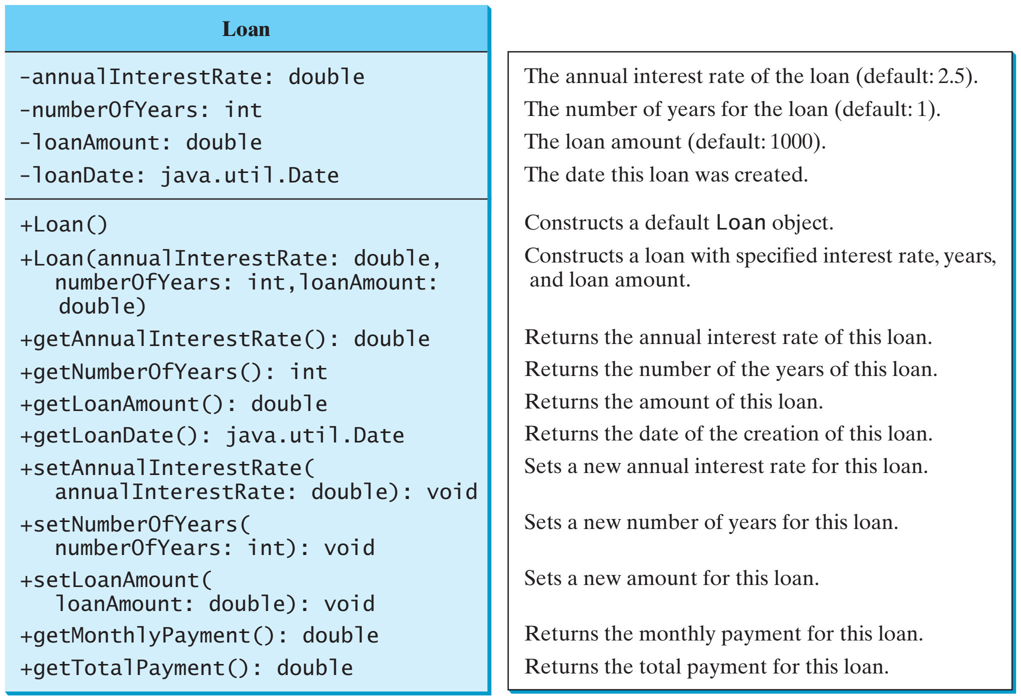

Loan
-annualInterestRate: double
-numberOfYears: int
-loanAmount: double
-loanDate: java.util.Date
+Loan()
+Loan(annualInterestRate: double, numberOfYears: int,loanAmount: double)
+getAnnualInterestRate(): double
+getNumberOfYears(): int
+getLoanAmount(): double
+getLoanDate(): java.util.Date
+setAnnualInterestRate(annualInterestRate: double): void
+setNumberOfYears(numberOfYears: int): void
+setLoanAmount(loanAmount: double): void
+getMonthlyPayment(): double
+getTotalPayment(): double
The annual interest rate of the loan (default: 2.5).
The number of years for the loan (default: 1).
The loan amount (default: 1000).
The date this loan was created.
Constructs a default Loan object.
Constructs a loan with specified interest rate, years, and loan amount.
Returns the annual interest rate of this loan.
Returns the number of the years of this loan.
Returns the amount of this loan.
Returns the date of the creation of this loan.
Sets a new annual interest rate for this loan.
Sets a new number of years for this loan.
Sets a new amount for this loan.
Returns the monthly payment for this loan.
Returns the total payment for this loan.

FIGURE 10.2 The `Loan` class models the properties and behaviors of loans.

The UML diagram in Figure 10.2 serves as the contract for the `Loan` class. Throughout this book, you will play the roles of both class user and class developer. Remember that a class user can use the class without knowing how the class is implemented.

Assume that the `Loan` class is available. The program in Listing 10.1 uses that class.

LISTING 10.1 TestLoanClass.java

```java
import java.util.Scanner;

public class TestLoanClass {
  /** Main method */
  public static void main(String[] args) {
```

create Loan object

invoke instance method

invoke instance method

```
 6      // Create a Scanner
 7      Scanner input = new Scanner(System.in);
 8
 9      // Enter annual interest rate
10      System.out.print(
11        "Enter annual interest rate, for example, 8.25: ");
12      double annualInterestRate = input.nextDouble();
13
14      // Enter number of years
15      System.out.print("Enter number of years as an integer: ");
16      int numberOfYears = input.nextInt();
17
18      // Enter loan amount
19      System.out.print("Enter loan amount, for example, 120000.95: ");
20      double loanAmount = input.nextDouble();
21
22      // Create a Loan object
23      Loan loan =
24        new Loan(annualInterestRate, numberOfYears, loanAmount);
25
26      // Display loan date, monthly payment, and total payment
27      System.out.printf("The loan was created on %s\n" +
28        "The monthly payment is %.2f\nThe total payment is %.2f\n",
29        loan.getLoanDate().toString(), loan.getMonthlyPayment(),
30        loan.getTotalPayment());
31    }
32  }
```

```
Enter annual interest rate, for example, 8.25: 2.5 ↵Enter
Enter number of years as an integer: 5 ↵Enter
Enter loan amount, for example, 120000.95: 1000 ↵Enter
The loan was created on Sat Jun 16 21:12:50 EDT 2012
The monthly payment is 17.75
The total payment is 1064.84
```

The **main** method reads the interest rate, the payment period (in years), and the loan amount; creates a **Loan** object; and then obtains the monthly payment (line 29) and the total payment (line 30) using the instance methods in the **Loan** class.

The **Loan** class can be implemented as in Listing 10.2.

LISTING 10.2 Loan.java

no-arg constructor

```
public class Loan {
  private double annualInterestRate;
  private int numberOfYears;
  private double loanAmount;
  private java.util.Date loanDate;

  /** Default constructor */
  public Loan() {
    this(2.5, 1, 1000);
  }

  /** Construct a loan with specified annual interest rate,
     number of years, and loan amount
   */
  public Loan(double annualInterestRate, int numberOfYears,
      double loanAmount) {
    this.annualInterestRate = annualInterestRate;
    this.numberOfYears = numberOfYears;
    this.loanAmount = loanAmount;
    loanDate = new java.util.Date();
  }

  /** Return annualInterestRate */
  public double getAnnualInterestRate() {
    return annualInterestRate;
  }

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

  /** Return numberOfYears */
  public int getNumberOfYears() {
    return numberOfYears;
  }

  /** Set a new numberOfYears */
  public void setNumberOfYears(int numberOfYears) {
    this.numberOfYears = numberOfYears;
  }

  /** Return loanAmount */
  public double getLoanAmount() {
    return loanAmount;
  }

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

  /** Find monthly payment */
  public double getMonthlyPayment() {
    double monthlyInterestRate = annualInterestRate / 1200;
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
      (1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));
    return monthlyPayment;
  }

  /** Find total payment */
  public double getTotalPayment() {
    double totalPayment = getMonthlyPayment() * numberOfYears * 12;
    return totalPayment;
  }

  /** Return loan date */
  public java.util.Date getLoanDate() {
    return loanDate;
  }
}
```

constructor

From a class developer's perspective, a class is designed for use by many different customers. In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through constructors, properties, and methods.

The `Loan` class contains two constructors, four getter methods, three setter methods, and the methods for finding the monthly payment and the total payment. You can construct a `Loan` object by using the no-arg constructor or the constructor with three parameters: annual interest rate, number of years, and loan amount. When a loan object is created, its date is stored in the `loanDate` field. The `getLoanDate` method returns the date. The methods—`getAnnualInterest`, `getNumberOfYears`, and `getLoanAmount`—return the annual interest rate, payment years, and loan amount, respectively. All the data properties and methods in this class are tied to a specific instance of the `Loan` class. Therefore, they are instance variables and methods.

Important Pedagogical Tip

Use the UML diagram for the `Loan` class shown in Figure 10.2 to write a test program that uses the `Loan` class even though you don't know how the `Loan` class is implemented. This has three benefits:

- It demonstrates that developing a class and using a class are two separate tasks.
- It enables you to skip the complex implementation of certain classes without interrupting the sequence of this book.
- It is easier to learn how to implement a class if you are familiar with it by using the class.

For all the class examples from now on, create an object from the class and try using its methods before turning your attention to its implementation.

Check Point

10.1 If you redefine the `Loan` class in Listing 10.2 without setter methods, is the class immutable?

10.3 Thinking in Objects

Key Point

The procedural paradigm focuses on designing methods. 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.

Chapters 1–8 introduced fundamental programming techniques for problem solving using loops, methods, and arrays. Knowing these techniques lays a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software. This section improves the solution for a problem introduced in Chapter 3 using the object-oriented approach. From these improvements, you will gain insight into the differences between procedural and object-oriented programming and see the benefits of developing reusable code using objects and classes.

Listing 3.4, ComputeAndInterpretBMI.java, presented a program for computing body mass index. The code cannot be reused in other programs, because the code is in the `main` method. To make it reusable, define a static method to compute body mass index as follows:

```
public static double getBMI(double weight, double height)
```

This method is useful for computing body mass index for a specified weight and height. However, it has limitations. Suppose you need to associate the weight and height with a person's name and birth date. You could declare separate variables to store these values, but these values would not be tightly coupled. The ideal way to couple them is to create an object that contains them all. Since these values are tied to individual objects, they should be stored in instance data fields. You can define a class named `BMI` as shown in Figure 10.3.

VideoNote
The `BMI` class

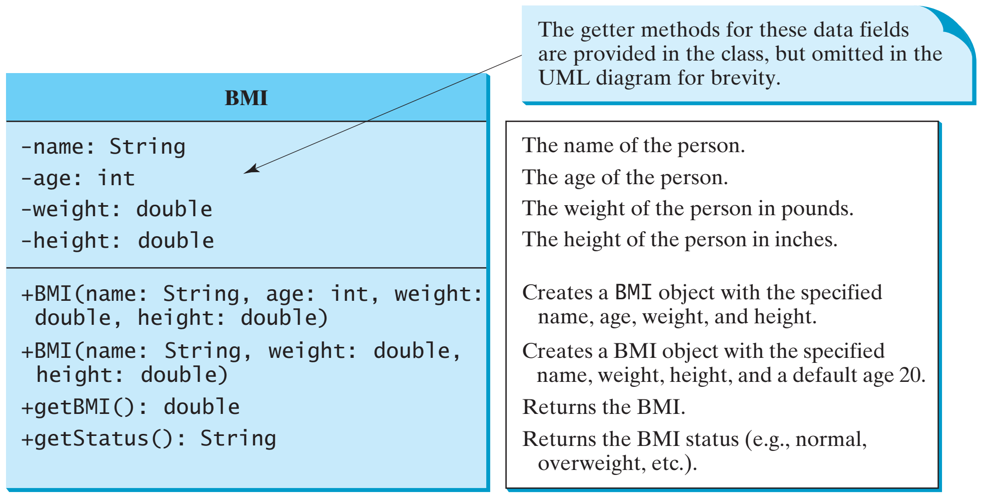

The getter methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
BMI
-name: String
-age: int
-weight: double
-height: double
+BMI(name: String, age: int, weight: double, height: double)
+BMI(name: String, weight: double, height: double)
+getBMI(): double
+getStatus(): String
The name of the person.
The age of the person.
The weight of the person in pounds.
The height of the person in inches.
Creates a BMI object with the specified name, age, weight, and height.
Creates a BMI object with the specified name, weight, height, and a default age 20.
Returns the BMI.
Returns the BMI status (e.g., normal, overweight, etc.).

FIGURE 10.3 The BMI class encapsulates BMI information.

Assume that the BMI class is available. Listing 10.3 gives a test program that uses this class.

LISTING 10.3 UseBMIClass.java

```
 1  public class UseBMIClass {
 2    public static void main(String[] args) {
 3      BMI bmi1 = new BMI("Kim Yang", 18, 145, 70);
 4      System.out.println("The BMI for " + bmi1.getName() + " is "
 5        + bmi1.getBMI() + " " + bmi1.getStatus());
 6
 7      BMI bmi2 = new BMI("Susan King", 215, 70);
 8      System.out.println("The BMI for " + bmi2.getName() + " is "
 9        + bmi2.getBMI() + " " + bmi2.getStatus());
10    }
11  }
```

create an object
invoke instance method
create an object
invoke instance method

```
The BMI for Kim Yang is 20.81 Normal
The BMI for Susan King is 30.85 Obese
```

Line 3 creates the object bmi1 for Kim Yang and line 7 creates the object bmi2 for Susan King. You can use the instance methods getName(), getBMI(), and getStatus() to return the BMI information in a BMI object.

The BMI class can be implemented as in Listing 10.4.

LISTING 10.4 BMI.java

```
 1  public class BMI {
 2    private String name;
 3    private int age;
 4    private double weight; // in pounds
 5    private double height; // in inches
 6    public static final double KILOGRAMS_PER_POUND = 0.45359237;
 7    public static final double METERS_PER_INCH = 0.0254;
 8
 9    public BMI(String name, int age, double weight, double height) {
10      this.name = name;
```

constructor

```
    this.age = age;
    this.weight = weight;
    this.height = height;
  }

```

constructor

```
  public BMI(String name, double weight, double height) {
    this(name, 20, weight, height);
  }

```

getBMI

```
  public double getBMI() {
    double bmi = weight * KILOGRAMS_PER_POUND /
      ((height * METERS_PER_INCH) * (height * METERS_PER_INCH));
    return Math.round(bmi * 100) / 100.0;
  }

```

getStatus

```
  public String getStatus() {
    double bmi = getBMI();
    if (bmi < 18.5)
      return "Underweight";
    else if (bmi < 25)
      return "Normal";
    else if (bmi < 30)
      return "Overweight";
    else
      return "Obese";
  }

  public String getName() {
    return name;
  }

  public int getAge() {
    return age;
  }

  public double getWeight() {
    return weight;
  }

  public double getHeight() {
    return height;
  }
}
```

The mathematical formula for computing the BMI using weight and height is given in Section 3.8. The instance method **getBMI()** returns the BMI. Since the weight and height are instance data fields in the object, the **getBMI()** method can use these properties to compute the BMI for the object.

The instance method **getStatus()** returns a string that interprets the BMI. The interpretation is also given in Section 3.8.

procedural vs. object-oriented paradigms

This example demonstrates the advantages of the object-oriented paradigm over the procedural paradigm. The procedural paradigm focuses on designing methods. 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.

In procedural programming, data and operations on the data are separate, and this methodology requires passing data to methods. Object-oriented programming places data and the operations that pertain to them in an object. This approach solves many of the problems inherent in procedural programming. The object-oriented programming approach organizes programs in a way that mirrors the real world, in which all objects are associated with both attributes and activities. Using objects improves software reusability and makes programs easier to develop and easier to maintain. Programming in Java involves thinking in terms of objects; a Java program can be viewed as a collection of cooperating objects.

10.2 Is the `BMI` class defined in Listing 10.4 immutable?

Check Point

10.4 Class Relationships

Key Point

To design classes, you need to explore the relationships among classes. The common relationships among classes are association, aggregation, composition, *and* inheritance.

This section explores association, aggregation, and composition. The inheritance relationship will be introduced in the next chapter.

10.4.1 Association

association

Association is a general binary relationship that describes an activity between two classes. For example, a student taking a course is an association between the `Student` class and the `Course` class, and a faculty member teaching a course is an association between the `Faculty` class and the `Course` class. These associations can be represented in UML graphical notation, as shown in Figure 10.4.

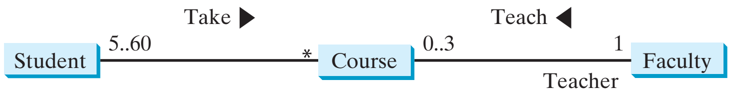


FIGURE 10.4 This UML diagram shows that a student may take any number of courses, a faculty member may teach at most three courses, a course may have from five to sixty students, and a course is taught by only one faculty member.

An association is illustrated by a solid line between two classes with an optional label that describes the relationship. In Figure 10.4, the labels are *Take* and *Teach*. Each relationship may have an optional small black triangle that indicates the direction of the relationship. In this figure, the direction indicates that a student takes a course (as opposed to a course taking a student).

Each class involved in the relationship may have a role name that describes the role it plays in the relationship. In Figure 10.4, *teacher* is the role name for `Faculty`.

multiplicity

Each class involved in an association may specify a *multiplicity*, which is placed at the side of the class to specify how many of the class's objects are involved in the relationship in UML. A multiplicity could be a number or an interval that specifies how many of the class's objects are involved in the relationship. The character `*` means an unlimited number of objects, and the interval `m..n` indicates that the number of objects is between `m` and `n`, inclusively. In Figure 10.4, each student may take any number of courses, and each course must have at least five and at most sixty students. Each course is taught by only one faculty member, and a faculty member may teach from zero to three courses per semester.

In Java code, you can implement associations by using data fields and methods. For example, the relationships in Figure 10.4 may be implemented using the classes in Figure 10.5. The relation "a student takes a course" is implemented using the **addCourse** method in the **Student** class and the **addStuent** method in the **Course** class. The relation "a faculty teaches a course" is implemented using the **addCourse** method in the **Faculty** class and the **setFaculty** method in the **Course** class. The **Student** class may use a list to store the courses that the student is taking, the **Faculty** class may use a list to store the courses that the faculty is teaching, and the **Course** class may use a list to store students enrolled in the course and a data field to store the instructor who teaches the course.

```java
public class Student {
  private Course[]
    courseList;

  public void addCourse(
    Course s) { ... }
}
```

```java
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  public void addStudent(
    Student s) { ... }

  public void setFaculty(
    Faculty faculty) { ... }
}
```

```java
public class Faculty {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```

FIGURE 10.5 The association relations are implemented using data fields and methods in classes.

many possible implementations

Note

There are many possible ways to implement relationships. For example, the student and faculty information in the **Course** class can be omitted, since they are already in the **Student** and **Faculty** class. Likewise, if you don't need to know the courses a student takes or a faculty member teaches, the data field **courseList** and the **addCourse** method in **Student** or **Faculty** can be omitted.

10.4.2 Aggregation and Composition

aggregation
aggregating object
aggregated object
aggregated class
aggregating class

Aggregation is a special form of association that represents an ownership relationship between two objects. Aggregation models *has-a* relationships. The owner object is called an *aggregating object*, and its class is called an *aggregating class*. The subject object is called an *aggregated object*, and its class is called an *aggregated class*.

composition

An object can be owned by several other aggregating objects. If an object is exclusively owned by an aggregating object, the relationship between the object and its aggregating object is referred to as a *composition*. For example, "a student has a name" is a composition relationship between the **Student** class and the **Name** class, whereas "a student has an address" is an aggregation relationship between the **Student** class and the **Address** class, since an address can be shared by several students. In UML, a filled diamond is attached to an aggregating class (in this case, **Student**) to denote the composition relationship with an aggregated class (**Name**), and an empty diamond is attached to an aggregating class (**Student**) to denote the aggregation relationship with an aggregated class (**Address**), as shown in Figure 10.6.

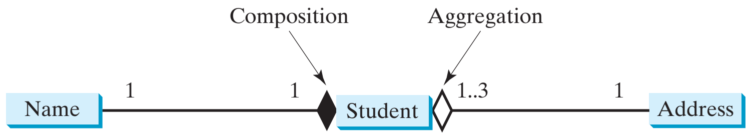


FIGURE 10.6 Each student has a name and an address.

In Figure 10.6, each student has only one multiplicity—address—and each address can be shared by up to **3** students. Each student has one name, and a name is unique for each student.

An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationships in Figure 10.6 may be implemented using the classes in Figure 10.7. The relation "a student has a name" and "a student has an address" are implemented in the data field **name** and **address** in the **Student** class.

```
public class Name {
  ...
}
```

Aggregated class

```
public class Student {
  private Name name;
  private Address address;
  ...
}
```

Aggregating class

```
public class Address {
  ...
}
```

Aggregated class

FIGURE 10.7 The composition relations are implemented using data fields in classes.

Aggregation may exist between objects of the same class. For example, a person may have a supervisor. This is illustrated in Figure 10.8.

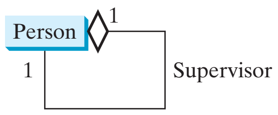
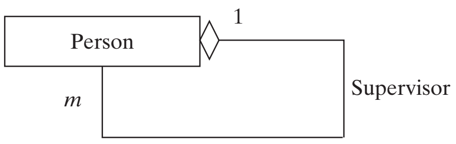


FIGURE 10.8 A person may have a supervisor.

In the relationship "a person has a supervisor," a supervisor can be represented as a data field in the **Person** class, as follows:

```
public class Person {
  // The type for the data is the class itself
  private Person supervisor;

  ...
}
```

If a person can have several supervisors, as shown in Figure 10.9a, you may use an array to store supervisors, as shown in Figure 10.9b.



(a)

```
public class Person {
  ...
  private Person[] supervisors;
}
```

(b)

FIGURE 10.9 A person can have several supervisors.

Note

Since aggregation and composition relationships are represented using classes in the same way, we will not differentiate them and call both compositions for simplicity.

aggregation or composition

Check Point

10.3 What are common relationships among classes?

10.4 What is association? What is aggregation? What is composition?

10.5 What is UML notation of aggregation and composition?

10.6 Why both aggregation and composition are together referred to as composition?

10.5 Case Study: Designing the **Course** Class

Key Point

This section designs a class for modeling courses.

This book's philosophy is *teaching by example and learning by doing*. The book provides a wide variety of examples to demonstrate object-oriented programming. This section and the next offer additional examples on designing classes.

Suppose you need to process course information. Each course has a name and has students enrolled. You should be able to add/drop a student to/from the course. You can use a class to model the courses, as shown in Figure 10.10.

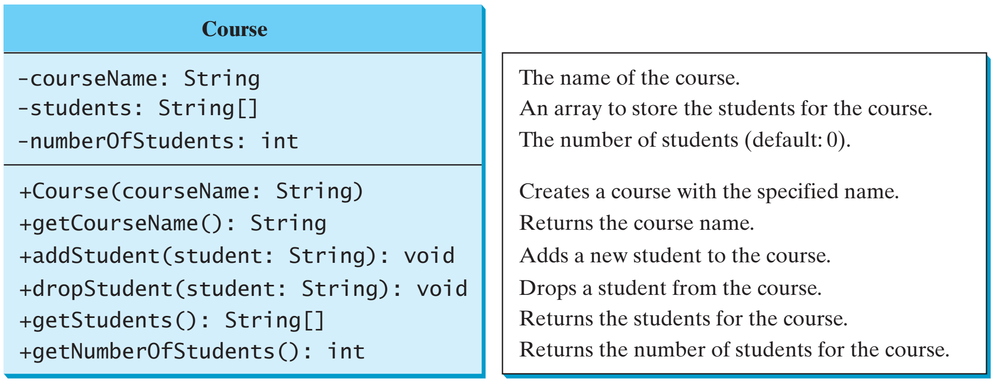

Course
-courseName: String
-students: String[]
-numberOfStudents: int
+Course(courseName: String)
+getCourseName(): String
+addStudent(student: String): void
+dropStudent(student: String): void
+getStudents(): String[]
+getNumberOfStudents(): int
The name of the course.
An array to store the students for the course.
The number of students (default: 0).
Creates a course with the specified name.
Returns the course name.
Adds a new student to the course.
Drops a student from the course.
Returns the students for the course.
Returns the number of students for the course.

Figure 10.10 The `Course` class models the courses.

A `Course` object can be created using the constructor `Course(String name)` by passing a course name. You can add students to the course using the `addStudent(String student)` method, drop a student from the course using the `dropStudent(String student)` method, and return all the students in the course using the `getStudents()` method. Suppose the `Course` class is available; Listing 10.5 gives a test class that creates two courses and adds students to them.

Listing 10.5 TestCourse.java

create a course

add a student

number of students
return students

```java
public class TestCourse {
  public static void main(String[] args) {
    Course course1 = new Course("Data Structures");
    Course course2 = new Course("Database Systems");

    course1.addStudent("Peter Jones");
    course1.addStudent("Kim Smith");
    course1.addStudent("Anne Kennedy");

    course2.addStudent("Peter Jones");
    course2.addStudent("Steve Smith");

    System.out.println("Number of students in course1: "
      + course1.getNumberOfStudents());
    String[] students = course1.getStudents();
    for (int i = 0; i < course1.getNumberOfStudents(); i++)
      System.out.print(students[i] + ", ");

    System.out.println();
    System.out.print("Number of students in course2: "
      + course2.getNumberOfStudents());
  }
}
```

```
Number of students in course1: 3
Peter Jones, Kim Smith, Anne Kennedy,
Number of students in course2: 2
```

The **Course** class is implemented in Listing 10.6. It uses an array to store the students in the course. For simplicity, assume that the maximum course enrollment is **100**. The array is created using **new String[100]** in line 3. The **addStudent** method (line 10) adds a student to the array. Whenever a new student is added to the course, **numberOfStudents** is increased (line 12). The **getStudents** method returns the array. The **dropStudent** method (line 27) is left as an exercise.

LISTING 10.6 Course.java

```
 1 public class Course {
 2   private String courseName;
 3   private String[] students = new String[100];
 4   private int numberOfStudents;
 5
 6   public Course(String courseName) {
 7     this.courseName = courseName;
 8   }
 9
10   public void addStudent(String student) {
11     students[numberOfStudents] = student;
12     numberOfStudents++;
13   }
14
15   public String[] getStudents() {
16     return students;
17   }
18
19   public int getNumberOfStudents() {
20     return numberOfStudents;
21   }
22
23   public String getCourseName() {
24     return courseName;
25   }
26
27   public void dropStudent(String student) {
28     // Left as an exercise in Programming Exercise 10.9
29   }
30 }
```

create students

add a course

return students

number of students

The array size is fixed to be **100** (line 3), so you cannot have more than 100 students in the course. You can improve the class by automatically increasing the array size in Programming Exercise 10.9.

When you create a **Course** object, an array object is created. A **Course** object contains a reference to the array. For simplicity, you can say that the **Course** object contains the array.

The user can create a **Course** object and manipulate it through the public methods **addStudent**, **dropStudent**, **getNumberOfStudents**, and **getStudents**. However, the user doesn't need to know how these methods are implemented. The **Course** class encapsulates the internal implementation. This example uses an array to store students, but you could use a different data structure to store **students**. The program that uses **Course** does not need to change as long as the contract of the public methods remains unchanged.

10.6 Case Study: Designing a Class for Stacks

Key Point

This section designs a class for modeling stacks.

stack

Recall that a *stack* is a data structure that holds data in a last-in, first-out fashion, as shown in Figure 10.11.

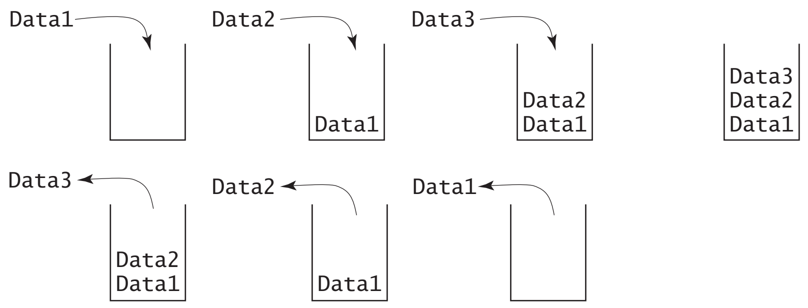
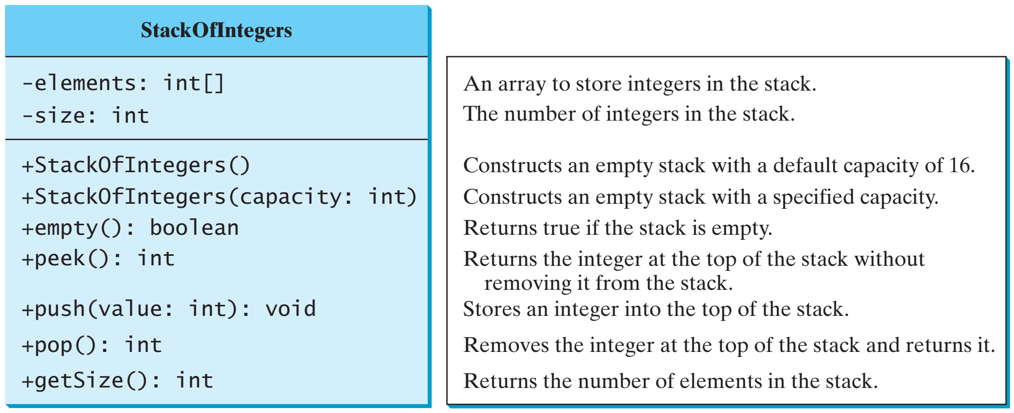


FIGURE 10.11 A stack holds data in a last-in, first-out fashion.

Stacks have many applications. For example, the compiler uses a stack to process method invocations. When a method is invoked, its parameters and local variables are pushed into a stack. When a method calls another method, the new method's parameters and local variables are pushed into the stack. When a method finishes its work and returns to its caller, its associated space is released from the stack.

VideoNote
The `StackOfIntegers` class

You can define a class to model stacks. For simplicity, assume the stack holds the `int` values. So name the stack class `StackOfIntegers`. The UML diagram for the class is shown in Figure 10.12.



StackOfIntegers	
`-elements: int[]`	An array to store integers in the stack.
`-size: int`	The number of integers in the stack.
`+StackOfIntegers()`	Constructs an empty stack with a default capacity of 16.
`+StackOfIntegers(capacity: int)`	Constructs an empty stack with a specified capacity.
`+empty(): boolean`	Returns true if the stack is empty.
`+peek(): int`	Returns the integer at the top of the stack without removing it from the stack.
`+push(value: int): void`	Stores an integer into the top of the stack.
`+pop(): int`	Removes the integer at the top of the stack and returns it.
`+getSize(): int`	Returns the number of elements in the stack.

FIGURE 10.12 The `StackOfIntegers` class encapsulates the stack storage and provides the operations for manipulating the stack.

Suppose that the class is available. The test program in Listing 10.7 uses the class to create a stack (line 3), store ten integers `0`, `1`, `2`, . . . , and `9` (line 6), and displays them in reverse order (line 9).

LISTING 10.7 `TestStackOfIntegers.java`

create a stack

```
1  public class TestStackOfIntegers {
2    public static void main(String[] args) {
3      StackOfIntegers stack = new StackOfIntegers();
```

```

    for (int i = 0; i < 10; i++)
      stack.push(i);

    while (!stack.empty())
      System.out.print(stack.pop() + " ");
  }
}
```

push to stack

pop from stack

```
9 8 7 6 5 4 3 2 1 0
```

How do you implement the **StackOfIntegers** class? The elements in the stack are stored in an array named **elements**. When you create a stack, the array is also created. The no-arg constructor creates an array with the default capacity of **16**. The variable **size** counts the number of elements in the stack, and **size – 1** is the index of the element at the top of the stack, as shown in Figure 10.13. For an empty stack, **size** is **0**.

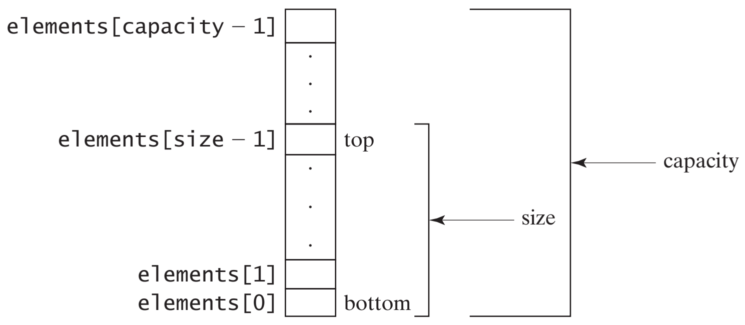


FIGURE 10.13 The **StackOfIntegers** class encapsulates the stack storage and provides the operations for manipulating the stack.

The **StackOfIntegers** class is implemented in Listing 10.8. The methods **empty()**, **peek()**, **pop()**, and **getSize()** are easy to implement. To implement **push(int value)**, assign **value** to **elements[size]** if **size < capacity** (line 24). If the stack is full (i.e., **size >= capacity**), create a new array of twice the current capacity (line 19), copy the contents of the current array to the new array (line 20), and assign the reference of the new array to the current array in the stack (line 21). Now you can add the new value to the array (line 24).

LISTING 10.8 StackOfIntegers.java

```
 1 public class StackOfIntegers {
 2   private int[] elements;
 3   private int size;
 4   public static final int DEFAULT_CAPACITY = 16;
 5
 6   /** Construct a stack with the default capacity 16 */
 7   public StackOfIntegers() {
 8     this (DEFAULT_CAPACITY);
 9   }
10
11   /** Construct a stack with the specified maximum capacity */
12   public StackOfIntegers(int capacity) {
13     elements = new int[capacity];
14   }
15
```

max capacity 16

```
  /** Push a new integer to the top of the stack */
  public void push(int value) {
    if (size >= elements.length) {
      int[] temp = new int[elements.length * 2];
      System.arraycopy(elements, 0, temp, 0, elements.length);
      elements = temp;
    }

    elements[size++] = value;
  }

  /** Return and remove the top element from the stack */
  public int pop() {
    return elements[--size];
  }

  /** Return the top element from the stack */
  public int peek() {
    return elements[size - 1];
  }

  /** Test whether the stack is empty */
  public boolean empty() {
    return size == 0;
  }

  /** Return the number of elements in the stack */
  public int getSize() {
    return size;
  }
}
```

double the capacity

add to stack

10.7 Processing Primitive Data Type Values as Objects

Key Point

A primitive type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API.

Owing to performance considerations, primitive data type values are not objects in Java. Because of the overhead of processing objects, the language's performance would be adversely affected if primitive data type values were treated as objects. However, many Java methods require the use of objects as arguments. Java offers a convenient way to incorporate, or wrap, a primitive data type into an object (e.g., wrapping **int** into the **Integer** class, wrapping **double** into the **Double** class, and wrapping **char** into the **Character** class,). By using a wrapper class, you can process primitive data type values as objects. Java provides **Boolean**, **Character**, **Double**, **Float**, **Byte**, **Short**, **Integer**, and **Long** wrapper classes in the **java.lang** package for primitive data types. The **Boolean** class wraps a Boolean value **true** or **false**. This section uses **Integer** and **Double** as examples to introduce the numeric wrapper classes.

why wrapper class?

Note

naming convention

Most wrapper class names for a primitive type are the same as the primitive data type name with the first letter capitalized. The exceptions are **Integer** and **Character**.

Numeric wrapper classes are very similar to each other. Each contains the methods **doubleValue()**, **floatValue()**, **intValue()**, **longValue()**, **shortValue()**, and **byteValue()**. These methods "convert" objects into primitive type values. The key features of **Integer** and **Double** are shown in Figure 10.14.

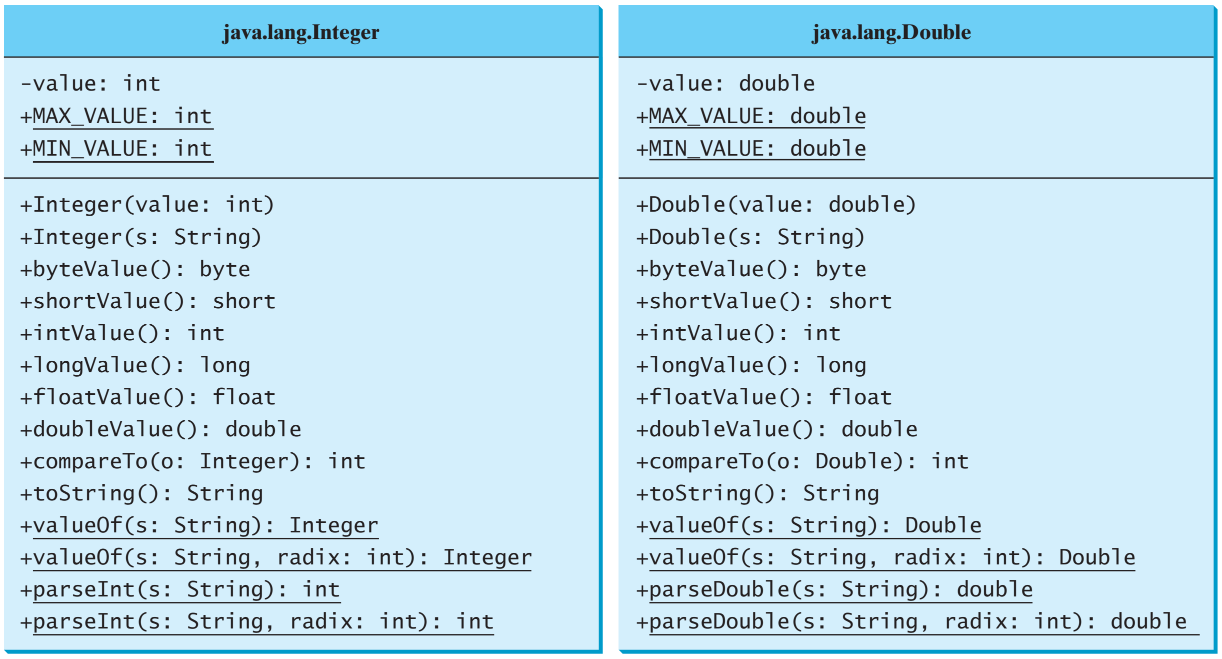


FIGURE 10.14 The wrapper classes provide constructors, constants, and conversion methods for manipulating various data types.

constructors

You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value—for example, `new Double(5.0)`, `new Double("5.0")`, `new Integer(5)`, and `new Integer("5")`.

no no-arg constructor
immutable

The wrapper classes do not have no-arg constructors. The instances of all wrapper classes are immutable; this means that, once the objects are created, their internal values cannot be changed.

constants

Each numeric wrapper class has the constants `MAX_VALUE` and `MIN_VALUE`. `MAX_VALUE` represents the maximum value of the corresponding primitive data type. For `Byte`, `Short`, `Integer`, and `Long`, `MIN_VALUE` represents the minimum `byte`, `short`, `int`, and `long` values. For `Float` and `Double`, `MIN_VALUE` represents the minimum *positive* `float` and `double` values. The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E–45), and the maximum double floating-point number (1.79769313486231570e + 308d).

```
System.out.println("The maximum integer is " + Integer.MAX_VALUE);
System.out.println("The minimum positive float is " +
  Float.MIN_VALUE);
System.out.println(
  "The maximum double-precision floating-point number is " +
  Double.MAX_VALUE);
```

conversion methods

Each numeric wrapper class contains the methods `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`, and `shortValue()` for returning a `double`, `float`, `int`, `long`, or `short` value for the wrapper object. For example,

```
new Double(12.4).intValue() returns 12;
new Integer(12).doubleValue() returns 12.0;
```

`compareTo` method

Recall that the `String` class contains the `compareTo` method for comparing two strings. The numeric wrapper classes contain the `compareTo` method for comparing two numbers and returns `1`, `0`, or `-1`, if this number is greater than, equal to, or less than the other number. For example,

```
new Double(12.4).compareTo(new Double(12.3)) returns 1;
new Double(12.3).compareTo(new Double(12.3)) returns 0;
new Double(12.3).compareTo(new Double(12.51)) returns -1;
```

static `valueOf` methods

The numeric wrapper classes have a useful static method, `valueOf (String s)`. This method creates a new object initialized to the value represented by the specified string. For example,

```
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```

static parsing methods

You have used the `parseInt` method in the `Integer` class to parse a numeric string into an `int` value and the `parseDouble` method in the `Double` class to parse a numeric string into a `double` value. Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on `10` (decimal) or any specified radix (e.g., `2` for binary, `8` for octal, and `16` for hexadecimal).

```
// These two methods are in the Byte class
public static byte parseByte(String s)
public static byte parseByte(String s, int radix)

// These two methods are in the Short class
public static short parseShort(String s)
public static short parseShort(String s, int radix)

// These two methods are in the Integer class
public static int parseInt(String s)
public static int parseInt(String s, int radix)

// These two methods are in the Long class
public static long parseLong(String s)
public static long parseLong(String s, int radix)

// These two methods are in the Float class
public static float parseFloat(String s)
public static float parseFloat(String s, int radix)

// These two methods are in the Double class
public static double parseDouble(String s)
public static double parseDouble(String s, int radix)
```

For example,

```
Integer.parseInt("11", 2) returns 3;
Integer.parseInt("12", 8) returns 10;
Integer.parseInt("13", 10) returns 13;
Integer.parseInt("1A", 16) returns 26;
```

`Integer.parseInt("12", 2)` would raise a runtime exception because `12` is not a binary number.

converting decimal to hex

Note that you can convert a decimal number into a hex number using the `format` method. For example,

```
String.format("%x", 26) returns 1A;
```

Check Point

10.7 Describe primitive-type wrapper classes.

10.8 Can each of the following statements be compiled?

a. `Integer i = new Integer("23");`

b. `Integer i = new Integer(23);`

c. `Integer i = Integer.valueOf("23");`

d. `Integer i = Integer.parseInt("23", 8);`

e. `Double d = new Double();`

f. `Double d = Double.valueOf("23.45");`

g. `int i = (Integer.valueOf("23")).intValue();`

h. `double d = (Double.valueOf("23.4")).doubleValue();`

i. `int i = (Double.valueOf("23.4")).intValue();`

j. `String s = (Double.valueOf("23.4")).toString();`

10.9 How do you convert an integer into a string? How do you convert a numeric string into an integer? How do you convert a double number into a string? How do you convert a numeric string into a double value?

10.10 Show the output of the following code.

```
public class Test {
  public static void main(String[] args) {
    Integer x = new Integer(3);
    System.out.println(x.intValue());
    System.out.println(x.compareTo(new Integer(4)));
  }
}
```

10.11 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    System.out.println(Integer.parseInt("10"));
    System.out.println(Integer.parseInt("10", 10));
    System.out.println(Integer.parseInt("10", 16));
    System.out.println(Integer.parseInt("11"));
    System.out.println(Integer.parseInt("11", 10));
    System.out.println(Integer.parseInt("11", 16));
  }
}
```

10.8 Automatic Conversion between Primitive Types and Wrapper Class Types

Key Point

A primitive type value can be automatically converted to an object using a wrapper class, and vice versa, depending on the context.

Converting a primitive value to a wrapper object is called *boxing*. The reverse conversion is called *unboxing*. Java allows primitive types and wrapper classes to be converted automatically. The compiler will automatically box a primitive value that appears in a context requiring an object, and will unbox an object that appears in a context requiring a primitive value. This is called *autoboxing* and *autounboxing*.

boxing
unboxing

autoboxing
autounboxing

For instance, the following statement in (a) can be simplified as in (b) due to autoboxing.

```
Integer intObject = new Integer (2);
```
(a)

Equivalent

autoboxing

```
Integer intObject = 2;
```
(b)

Consider the following example:

```
1  Integer[] intArray = {1, 2, 3};
2  System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

In line 1, the primitive values `1`, `2`, and `3` are automatically boxed into objects `new Integer(1)`, `new Integer(2)`, and `new Integer(3)`. In line 2, the objects `intArray[0]`, `intArray[1]`, and `intArray[2]` are automatically unboxed into `int` values that are added together.

Check Point

10.12 What are autoboxing and autounboxing? Are the following statements correct?

a. `Integer x = 3 + new Integer(5);`
b. `Integer x = 3;`
c. `Double x = 3;`
d. `Double x = 3.0;`
e. `int x = new Integer(3);`
f. `int x = new Integer(3) + new Integer(4);`

10.13 Show the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    Double x = 3.5;
    System.out.println(x.intValue());
    System.out.println(x.compareTo(4.5));
  }
}
```

10.9 The BigInteger and BigDecimal Classes

Key Point

The `BigInteger` and `BigDecimal` classes can be used to represent integers or decimal numbers of any size and precision.

immutable

VideoNote
Process large numbers

If you need to compute with very large integers or high-precision floating-point values, you can use the `BigInteger` and `BigDecimal` classes in the `java.math` package. Both are *immutable*. The largest integer of the `long` type is `Long.MAX_VALUE` (i.e., `9223372036854775807`). An instance of `BigInteger` can represent an integer of any size. You can use `new BigInteger(String)` and `new BigDecimal(String)` to create an instance of `BigInteger` and `BigDecimal`, use the `add`, `subtract`, `multiply`, `divide`, and `remainder` methods to perform arithmetic operations, and use the `compareTo` method to compare two big numbers. For example, the following code creates two `BigInteger` objects and multiplies them.

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

The output is `18446744073709551614`.

There is no limit to the precision of a `BigDecimal` object. The `divide` method may throw an `ArithmeticException` if the result cannot be terminated. However, you can use the overloaded `divide(BigDecimal d, int scale, int roundingMode)` method to specify a scale and a rounding mode to avoid this exception, where `scale` is the maximum number of digits after the decimal point. For example, the following code creates two `BigDecimal` objects and performs division with scale `20` and rounding mode `BigDecimal.ROUND_UP`.

```
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

The output is `0.33333333333333333334`.

Note that the factorial of an integer can be very large. Listing 10.9 gives a method that can return the factorial of any integer.

LISTING 10.9 LargeFactorial.java

```
 1  import java.math.*;
 2
 3  public class LargeFactorial {
 4    public static void main(String[] args) {
 5      System.out.println("50! is \n" + factorial(50));
 6    }
 7
 8    public static BigInteger factorial(long n) {
 9      BigInteger result = BigInteger.ONE;
10      for (int i = 1; i <= n; i++)
11        result = result.multiply(new BigInteger(i + ""));
12
13      return result;
14    }
15  }
```

constant

multiply

```
50! is
30414093201713378043612608166064768844377641568960512000000000000
```

`BigInteger.ONE` (line 9) is a constant defined in the `BigInteger` class. `BigInteger.ONE` is the same as `new BigInteger("1")`.

A new result is obtained by invoking the `multiply` method (line 11).

Check Point

10.14 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    java.math.BigInteger x = new java.math.BigInteger("3");
    java.math.BigInteger y = new java.math.BigInteger("7");
    java.math.BigInteger z = x.add(y);
    System.out.println("x is " + x);
    System.out.println("y is " + y);
    System.out.println("z is " + z);
  }
}
```

10.10 The **String** Class

Key Point

*A **String** object is immutable: Its content cannot be changed once the string is created.*

VideoNote
The String class

Strings were introduced in Section 4.4. You know strings are objects. You can invoke the **charAt(index)** method to obtain a character at the specified index from a string, the **length()** method to return the size of a string, the **substring** method to return a substring in a string, and the **indexOf** and **lastIndexOf** methods to return the first or last index of a matching character or a substring. We will take a closer look at strings in this section.

The **String** class has 13 constructors and more than 40 methods for manipulating strings. Not only is it very useful in programming, but it is also a good example for learning classes and objects.

10.10.1 Constructing a String

You can create a string object from a string literal or from an array of characters. To create a string from a string literal, use the syntax:

```
String newString = new String(stringLiteral);
```

The argument **stringLiteral** is a sequence of characters enclosed inside double quotes. The following statement creates a **String** object **message** for the string literal **"Welcome to Java"**:

```
String message = new String("Welcome to Java");
```

string literal object

Java treats a string literal as a **String** object. Thus, the following statement is valid:

```
String message = "Welcome to Java";
```

You can also create a string from an array of characters. For example, the following statements create the string **"Good Day"**:

```
char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};
String message = new String(charArray);
```

String variable, String object, string value

Note

A **String** variable holds a reference to a **String** object that stores a string value. Strictly speaking, the terms ***String*** *variable*, ***String*** *object*, and *string value* are different, but most of the time the distinctions between them can be ignored. For simplicity, the term *string* will often be used to refer to **String** variable, **String** object, and string value.

10.10.2 Immutable Strings and Interned Strings

immutable

A **String** object is immutable; its contents cannot be changed. Does the following code change the contents of the string?

```
String s = "Java";
s = "HTML";
```

The answer is no. The first statement creates a **String** object with the content **"Java"** and assigns its reference to **s**. The second statement creates a new **String** object with the content **"HTML"** and assigns its reference to **s**. The first **String** object still exists after the assignment, but it can no longer be accessed, because variable **s** now points to the new object, as shown in Figure 10.15.

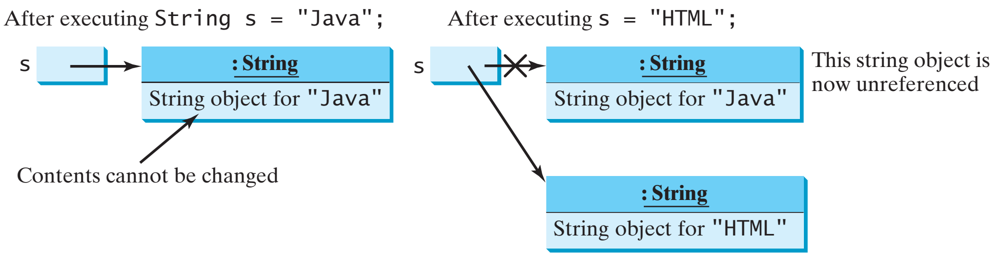
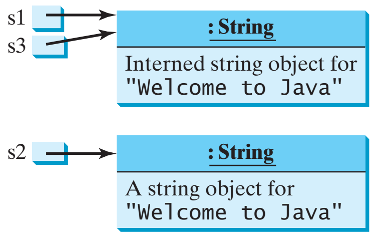

After executing String s = "Java";
s
: String
String object for "Java"
Contents cannot be changed
After executing s = "HTML";
s
: String
String object for "Java"
This string object is now unreferenced
: String
String object for "HTML"

FIGURE 10.15 Strings are immutable; once created, their contents cannot be changed.

Because strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory. Such an instance is called an *interned string*. For example, the following statements:

interned string

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```


s1
s3
: String
Interned string object for "Welcome to Java"
s2
: String
A string object for "Welcome to Java"

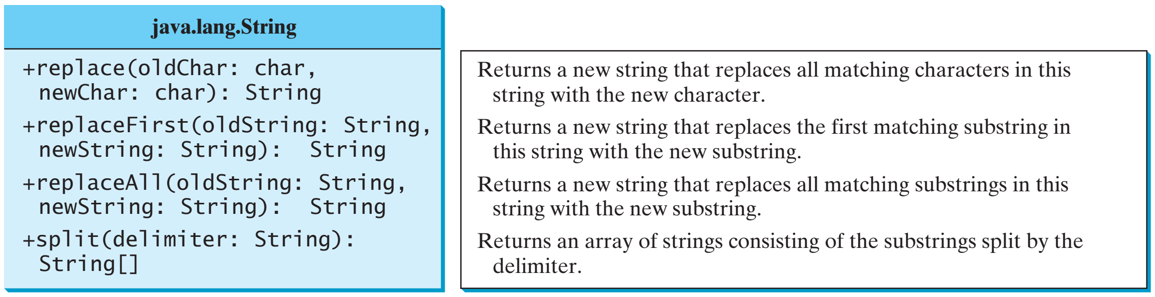
display

```
s1 == s2 is false
s1 == s3 is true
```

In the preceding statements, `s1` and `s3` refer to the same interned string—`"Welcome to Java"`—so `s1 == s3` is `true`. However, `s1 == s2` is `false`, because `s1` and `s2` are two different string objects, even though they have the same contents.

10.10.3 Replacing and Splitting Strings

The `String` class provides the methods for replacing and splitting strings, as shown in Figure 10.16.



java.lang.String	
+replace(oldChar: char, newChar: char): String	Returns a new string that replaces all matching characters in this string with the new character.
+replaceFirst(oldString: String, newString: String): String	Returns a new string that replaces the first matching substring in this string with the new substring.
+replaceAll(oldString: String, newString: String): String	Returns a new string that replaces all matching substrings in this string with the new substring.
+split(delimiter: String): String[]	Returns an array of strings consisting of the substrings split by the delimiter.

FIGURE 10.16 The `String` class contains the methods for replacing and splitting strings.

Once a string is created, its contents cannot be changed. The methods **replace**, **replaceFirst**, and **replaceAll** return a new string derived from the original string (without changing the original string!). Several versions of the **replace** methods are provided to replace a character or a substring in the string with a new character or a new substring.

For example,

replace
replaceFirst
replace
replace

```
"Welcome".replace('e', 'A') returns a new string, WAlcomA.
"Welcome".replaceFirst("e", "AB") returns a new string, WABlcome.
"Welcome".replace("e", "AB") returns a new string, WABlcomAB.
"Welcome".replace("el", "AB") returns a new string, WABcome.
```

split

The **split** method can be used to extract tokens from a string with the specified delimiters. For example, the following code

```
String[] tokens = "Java#HTML#Perl".split("#");
for (int i = 0; i < tokens.length; i++)
  System.out.print(tokens[i] + " ");
```

displays

```
Java HTML Perl
```

10.10.4 Matching, Replacing and Splitting by Patterns

why regular expression?

Often you will need to write code that validates user input, such as to check whether the input is a number, a string with all lowercase letters, or a Social Security number. How do you write this type of code? A simple and effective way to accomplish this task is to use the regular expression.

regular expression
regex

A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. You can match, replace, or split a string by specifying a pattern. This is an extremely useful and powerful feature.

matches(regex)

Let us begin with the **matches** method in the **String** class. At first glance, the **matches** method is very similar to the **equals** method. For example, the following two statements both evaluate to **true**.

```
"Java".matches("Java");
"Java".equals("Java");
```

However, the **matches** method is more powerful. It can match not only a fixed string, but also a set of strings that follow a pattern. For example, the following statements all evaluate to **true**:

```
"Java is fun".matches("Java.*")
"Java is cool".matches("Java.*")
"Java is powerful".matches("Java.*")
```

Java.* in the preceding statements is a regular expression. It describes a string pattern that begins with Java followed by *any* zero or more characters. Here, the substring matches any zero or more characters.

The following statement evaluates to **true**.

```
"440-02-4534".matches("\\d{3}-\\d{2}-\\d{4}")
```

Here **\\d** represents a single digit, and **\\d{3}** represents three digits.

The `replaceAll`, `replaceFirst`, and `split` methods can be used with a regular expression. For example, the following statement returns a new string that replaces `$`, `+`, or `#` in `a+b$#c` with the string `NNN`.

replaceAll(regex)

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

Here the regular expression `[$+#]` specifies a pattern that matches `$`, `+`, or `#`. So, the output is `aNNNbNNNNNNc`.

The following statement splits the string into an array of strings delimited by punctuation marks.

split(regex)

```
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");

for (int i = 0; i < tokens.length; i++)
  System.out.println(tokens[i]);
```

further studies

In this example, the regular expression `[.,:;?]` specifies a pattern that matches `.`, `,`, `:`, `;`, or `?`. Each of these characters is a delimiter for splitting the string. Thus, the string is split into `Java`, `C`, `C#`, and `C++`, which are stored in array `tokens`.

Regular expression patterns are complex for beginning students to understand. For this reason, simple patterns are introduced in this section. Please refer to Appendix H, Regular Expressions, to learn more about these patterns.

10.10.5 Conversion between Strings and Arrays

Strings are not arrays, but a string can be converted into an array, and vice versa. To convert a string into an array of characters, use the `toCharArray` method. For example, the following statement converts the string `Java` to an array.

toCharArray

```
char[] chars = "Java".toCharArray();
```

Thus, `chars[0]` is `J`, `chars[1]` is `a`, `chars[2]` is `v`, and `chars[3]` is `a`.

You can also use the `getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)` method to copy a substring of the string from index `srcBegin` to index `srcEnd-1` into a character array `dst` starting from index `dstBegin`. For example, the following code copies a substring `"3720"` in `"CS3720"` from index `2` to index `6-1` into the character array `dst` starting from index `4`.

getChars

```
char[] dst = {'J', 'A', 'V', 'A', '1', '3', '0', '1'};
"CS3720".getChars(2, 6, dst, 4);
```

Thus, `dst` becomes `{'J', 'A', 'V', 'A', '3', '7', '2', '0'}`.

To convert an array of characters into a string, use the `String(char[])` constructor or the `valueOf(char[])` method. For example, the following statement constructs a string from an array using the `String` constructor.

```
String str = new String(new char[]{'J', 'a', 'v', 'a'});
```

The next statement constructs a string from an array using the `valueOf` method.

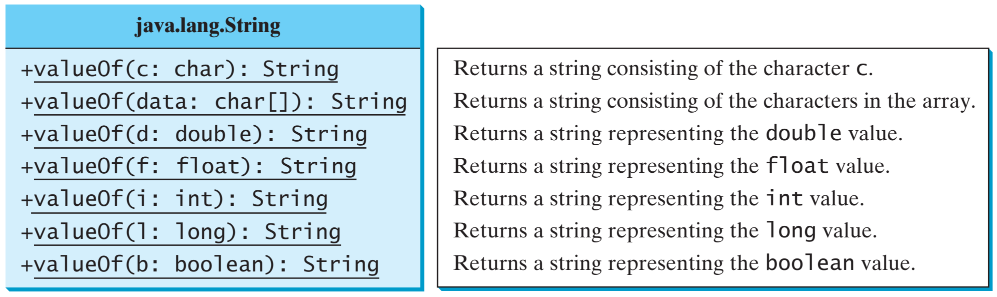
valueOf

```
String str = String.valueOf(new char[]{'J', 'a', 'v', 'a'});
```

10.10.6 Converting Characters and Numeric Values to Strings

Recall that you can use `Double.parseDouble(str)` or `Integer.parseInt(str)` to convert a string to a `double` value or an `int` value and you can convert a character or a number into a string by using the string concatenating operator. Another way of converting a overloaded `valueOf`

number into a string is to use the overloaded static **`valueOf`** method. This method can also be used to convert a character or an array of characters into a string, as shown in Figure 10.17.



java.lang.String	
+valueOf(c: char): String	Returns a string consisting of the character `c`.
+valueOf(data: char[]): String	Returns a string consisting of the characters in the array.
+valueOf(d: double): String	Returns a string representing the `double` value.
+valueOf(f: float): String	Returns a string representing the `float` value.
+valueOf(i: int): String	Returns a string representing the `int` value.
+valueOf(l: long): String	Returns a string representing the `long` value.
+valueOf(b: boolean): String	Returns a string representing the `boolean` value.

FIGURE 10.17 The **`String`** class contains the static methods for creating strings from primitive type values.

For example, to convert a **`double`** value **`5.44`** to a string, use **`String.valueOf(5.44)`**. The return value is a string consisting of the characters **`'5'`**, **`'.'`**, **`'4'`**, and **`'4'`**.

10.10.7 Formatting Strings

The **`String`** class contains the static **`format`** method to return a formatted string. The syntax to invoke this method is:

```
String.format(format, item1, item2, ..., itemk)
```

This method is similar to the **`printf`** method except that the **`format`** method returns a formatted string, whereas the **`printf`** method displays a formatted string. For example,

```
String s = String.format("%7.2f%6d%-4s", 45.556, 14, "AB");
System.out.println(s);
```

displays

```
□□45.56□□□□14AB□□
```

Note that

```
System.out.printf(format, item1, item2, ..., itemk);
```

is equivalent to

```
System.out.print(
  String.format(format, item1, item2, ..., itemk));
```

where the square box (□) denotes a blank space.

Check Point

10.15 Suppose that **`s1`**, **`s2`**, **`s3`**, and **`s4`** are four strings, given as follows:

```
String s1 = "Welcome to Java";
String s2 = s1;
String s3 = new String("Welcome to Java");
String s4 = "Welcome to Java";
```

What are the results of the following expressions?

a. `s1 == s2`

b. `s1 == s3`

c. `s1 == s4`
d. `s1.equals(s3)`
e. `s1.equals(s4)`
f. `"Welcome to Java".replace("Java", "HTML")`
g. `s1.replace('o', 'T')`
h. `s1.replaceAll("o", "T")`
i. `s1.replaceFirst("o", "T")`
j. `s1.toCharArray()`

10.16 To create the string `Welcome to Java`, you may use a statement like this:

```
String s = "Welcome to Java";
```

or:

```
String s = new String("Welcome to Java");
```

Which one is better? Why?

10.17 What is the output of the following code?

```
String s1 = "Welcome to Java";
String s2 = s1.replace("o", "abc");
System.out.println(s1);
System.out.println(s2);
```

10.18 Let `s1` be `"Welcome"` and `s2` be `"welcome"`. Write the code for the following statements:

a. Replace all occurrences of the character `e` with `E` in `s1` and assign the new string to `s2`.

b. Split `Welcome to Java and HTML` into an array `tokens` delimited by a space and assign the first two tokens into `s1` and `s2`.

10.19 Does any method in the `String` class change the contents of the string?

10.20 Suppose string `s` is created using `new String()`; what is `s.length()`?

10.21 How do you convert a `char`, an array of characters, or a number to a string?

10.22 Why does the following code cause a `NullPointerException`?

```
public class Test {
  private String text;

  public Test(String s) {
    String text  = s;
  }

  public static void main(String[] args) {
    Test test = new Test("ABC");
    System.out.println(test.text.toLowerCase());
  }
}
```

10.23 What is wrong in the following program?

```
public class Test {
  String text;

  public void Test(String s) {
    text = s;
  }

  public static void main(String[] args) {
    Test test = new Test("ABC");
    System.out.println(test);
  }
}
```

10.24 Show the output of the following code.

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Hi, ABC, good".matches("ABC "));
    System.out.println("Hi, ABC, good".matches(".*ABC.*"));
    System.out.println("A,B;C".replaceAll(",;", "#"));
    System.out.println("A,B;C".replaceAll("[,;]", "#"));

    String[] tokens = "A,B;C".split("[,;]");
    for (int i = 0; i < tokens.length; i++)
      System.out.print(tokens[i] +  " ");
  }
}
```

10.25 Show the output of the following code.

```
public class Test {
  public static void main(String[] args) {
    String s = "Hi, Good Morning";
    System.out.println(m(s));
  }

  public static int m(String s) {
    int count = 0;
    for (int i = 0; i < s.length(); i++)
      if (Character.isUpperCase(s.charAt(i)))
        count++;

    return count;
  }
}
```

10.11 The `StringBuilder` and `StringBuffer` Classes

Key Point

The `StringBuilder` and `StringBuffer` classes are similar to the `String` class except that the `String` class is immutable.

In general, the `StringBuilder` and `StringBuffer` classes can be used wherever a string is used. `StringBuilder` and `StringBuffer` are more flexible than `String`. You can add, insert, or append new contents into `StringBuilder` and `StringBuffer` objects, whereas the value of a `String` object is fixed once the string is created.

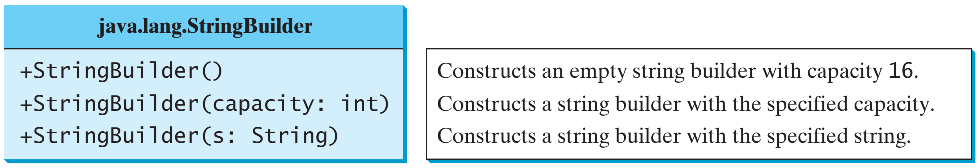
`StringBuilder`

The `StringBuilder` class is similar to `StringBuffer` except that the methods for modifying the buffer in `StringBuffer` are *synchronized*, which means that only one task is allowed to execute the methods. Use `StringBuffer` if the class might be accessed by multiple tasks concurrently, because synchronization is needed in this case to prevent corruptions to

StringBuffer. Concurrent programming will be introduced in Chapter 30. Using **StringBuilder** is more efficient if it is accessed by just a single task, because no synchronization is needed in this case. The constructors and methods in **StringBuffer** and **StringBuilder** are almost the same. This section covers **StringBuilder**. You can replace **StringBuilder** in all occurrences in this section by **StringBuffer**. The program can compile and run without any other changes.

StringBuilder constructors

The **StringBuilder** class has three constructors and more than 30 methods for managing the builder and modifying strings in the builder. You can create an empty string builder or a string builder from a string using the constructors, as shown in Figure 10.18.

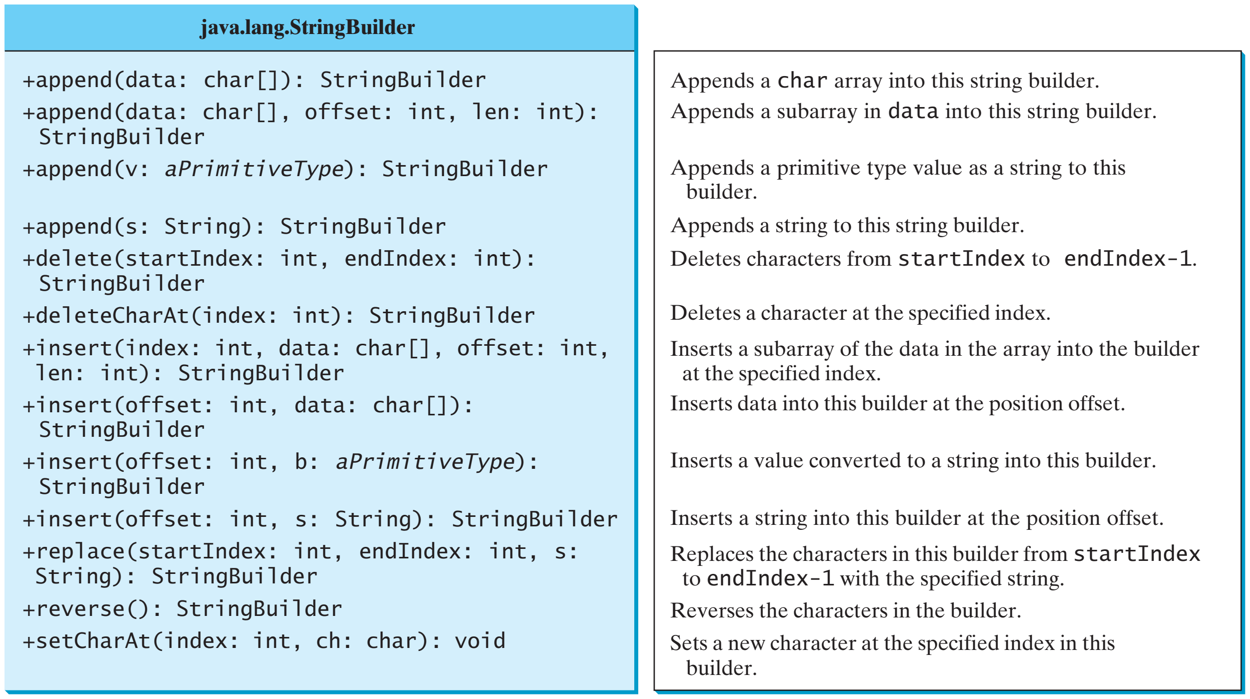


java.lang.StringBuilder	
`+StringBuilder()`	Constructs an empty string builder with capacity `16`.
`+StringBuilder(capacity: int)`	Constructs a string builder with the specified capacity.
`+StringBuilder(s: String)`	Constructs a string builder with the specified string.

FIGURE 10.18 The **StringBuilder** class contains the constructors for creating instances of **StringBuilder**.

10.11.1 Modifying Strings in the **StringBuilder**

You can append new contents at the end of a string builder, insert new contents at a specified position in a string builder, and delete or replace characters in a string builder, using the methods listed in Figure 10.19.



java.lang.StringBuilder	
`+append(data: char[]): StringBuilder`	Appends a `char` array into this string builder.
`+append(data: char[], offset: int, len: int): StringBuilder`	Appends a subarray in `data` into this string builder.
`+append(v: `*`aPrimitiveType`*`): StringBuilder`	Appends a primitive type value as a string to this builder.
`+append(s: String): StringBuilder`	Appends a string to this string builder.
`+delete(startIndex: int, endIndex: int): StringBuilder`	Deletes characters from `startIndex` to `endIndex-1`.
`+deleteCharAt(index: int): StringBuilder`	Deletes a character at the specified index.
`+insert(index: int, data: char[], offset: int, len: int): StringBuilder`	Inserts a subarray of the data in the array into the builder at the specified index.
`+insert(offset: int, data: char[]): StringBuilder`	Inserts data into this builder at the position offset.
`+insert(offset: int, b: `*`aPrimitiveType`*`): StringBuilder`	Inserts a value converted to a string into this builder.
`+insert(offset: int, s: String): StringBuilder`	Inserts a string into this builder at the position offset.
`+replace(startIndex: int, endIndex: int, s: String): StringBuilder`	Replaces the characters in this builder from `startIndex` to `endIndex-1` with the specified string.
`+reverse(): StringBuilder`	Reverses the characters in the builder.
`+setCharAt(index: int, ch: char): void`	Sets a new character at the specified index in this builder.

FIGURE 10.19 The **StringBuilder** class contains the methods for modifying string builders.

The `StringBuilder` class provides several overloaded methods to append `boolean`, `char`, `char[]`, `double`, `float`, `int`, `long`, and `String` into a string builder. For example, the following code appends strings and characters into `stringBuilder` to form a new string, `Welcome to Java`.

append

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("Welcome");
stringBuilder.append(' ');
stringBuilder.append("to");
stringBuilder.append(' ');
stringBuilder.append("Java");
```

The `StringBuilder` class also contains overloaded methods to insert `boolean`, `char`, `char array`, `double`, `float`, `int`, `long`, and `String` into a string builder. Consider the following code:

insert

```
stringBuilder.insert(11, "HTML and ");
```

Suppose `stringBuilder` contains `Welcome to Java` before the `insert` method is applied. This code inserts `"HTML and "` at position 11 in `stringBuilder` (just before the `J`). The new `stringBuilder` is `Welcome to HTML and Java`.

You can also delete characters from a string in the builder using the two `delete` methods, reverse the string using the `reverse` method, replace characters using the `replace` method, or set a new character in a string using the `setCharAt` method.

For example, suppose `stringBuilder` contains `Welcome to Java` before each of the following methods is applied:

delete
deleteCharAt
reverse
replace
setCharAt

`stringBuilder.delete(8, 11)` changes the builder to `Welcome Java`.
`stringBuilder.deleteCharAt(8)` changes the builder to `Welcome o Java`.
`stringBuilder.reverse()` changes the builder to `avaJ ot emocleW`.
`stringBuilder.replace(11, 15, "HTML")` changes the builder to `Welcome to HTML`.
`stringBuilder.setCharAt(0, 'w')` sets the builder to `welcome to Java`.

All these modification methods except `setCharAt` do two things:

- Change the contents of the string builder

ignore return value

- Return the reference of the string builder

For example, the following statement

```
StringBuilder stringBuilder1 = stringBuilder.reverse();
```

reverses the string in the builder and assigns the builder's reference to `stringBuilder1`. Thus, `stringBuilder` and `stringBuilder1` both point to the same `StringBuilder` object. Recall that a value-returning method can be invoked as a statement, if you are not interested in the return value of the method. In this case, the return value is simply ignored. For example, in the following statement

```
stringBuilder.reverse();
```

the return value is ignored.

String or StringBuilder?

Tip

If a string does not require any change, use `String` rather than `StringBuilder`. Java can perform some optimizations for `String`, such as sharing interned strings.

10.11.2 The **toString**, **capacity**, **length**, **setLength**, and **charAt** Methods

The `StringBuilder` class provides the additional methods for manipulating a string builder and obtaining its properties, as shown in Figure 10.20.

java.lang.StringBuilder	
`+toString(): String`	Returns a string object from the string builder.
`+capacity(): int`	Returns the capacity of this string builder.
`+charAt(index: int): char`	Returns the character at the specified index.
`+length(): int`	Returns the number of characters in this builder.
`+setLength(newLength: int): void`	Sets a new length in this builder.
`+substring(startIndex: int): String`	Returns a substring starting at `startIndex`.
`+substring(startIndex: int, endIndex: int): String`	Returns a substring from `startIndex` to `endIndex-1`.
`+trimToSize(): void`	Reduces the storage size used for the string builder.

FIGURE 10.20 The `StringBuilder` class contains the methods for modifying string builders.

`capacity()`

The `capacity()` method returns the current capacity of the string builder. The capacity is the number of characters the string builder is able to store without having to increase its size.

`length()`
`setLength(int)`

The `length()` method returns the number of characters actually stored in the string builder. The `setLength(newLength)` method sets the length of the string builder. If the `newLength` argument is less than the current length of the string builder, the string builder is truncated to contain exactly the number of characters given by the `newLength` argument. If the `newLength` argument is greater than or equal to the current length, sufficient null characters (`\u0000`) are appended to the string builder so that `length` becomes the `newLength` argument. The `newLength` argument must be greater than or equal to `0`.

`charAt(int)`

The `charAt(index)` method returns the character at a specific `index` in the string builder. The index is `0` based. The first character of a string builder is at index `0`, the next at index `1`, and so on. The `index` argument must be greater than or equal to `0`, and less than the length of the string builder.

Note

length and capacity

The length of the string is always less than or equal to the capacity of the builder. The length is the actual size of the string stored in the builder, and the capacity is the current size of the builder. The builder's capacity is automatically increased if more characters are added to exceed its capacity. Internally, a string builder is an array of characters, so the builder's capacity is the size of the array. If the builder's capacity is exceeded, the array is replaced by a new array. The new array size is `2 * (the previous array size + 1)`.

Tip

initial capacity

You can use `new StringBuilder(initialCapacity)` to create a `StringBuilder` with a specified initial capacity. By carefully choosing the initial capacity, you can make your program more efficient. If the capacity is always larger than the actual length of the builder, the JVM will never need to reallocate memory for the builder. On the other hand, if the capacity is too large, you will waste memory space. You can use the `trimToSize()` method to reduce the capacity to the actual size.

`trimToSize()`

10.11.3 Case Study: Ignoring Nonalphanumeric Characters When Checking Palindromes

Listing 5.14, Palindrome.java, considered all the characters in a string to check whether it is a palindrome. Write a new program that ignores nonalphanumeric characters in checking whether a string is a palindrome.

Here are the steps to solve the problem:

1. Filter the string by removing the nonalphanumeric characters. This can be done by creating an empty string builder, adding each alphanumeric character in the string to a string builder, and returning the string from the string builder. You can use the `isLetterOrDigit(ch)` method in the `Character` class to check whether character `ch` is a letter or a digit.

2. Obtain a new string that is the reversal of the filtered string. Compare the reversed string with the filtered string using the `equals` method.

The complete program is shown in Listing 10.10.

LISTING 10.10 PalindromeIgnoreNonAlphanumeric.java

```java
import java.util.Scanner;

public class PalindromeIgnoreNonAlphanumeric {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter a string
    System.out.print("Enter a string: ");
    String s = input.nextLine();

    // Display result
    System.out.println("Ignoring nonalphanumeric characters, \nis "
      + s + " a palindrome? " + isPalindrome(s));
  }

  /** Return true if a string is a palindrome */
  public static boolean isPalindrome(String s) {
    // Create a new string by eliminating nonalphanumeric chars
    String s1 = filter(s);

    // Create a new string that is the reversal of s1
    String s2 = reverse(s1);

    // Check if the reversal is the same as the original string
    return s2.equals(s1);
  }

  /** Create a new string by eliminating nonalphanumeric chars */
  public static String filter(String s) {
    // Create a string builder
    StringBuilder stringBuilder = new StringBuilder();

    // Examine each char in the string to skip alphanumeric char
    for (int i = 0; i < s.length(); i++) {
      if (Character.isLetterOrDigit(s.charAt(i))) {
         stringBuilder.append(s.charAt(i));
      }
```

check palindrome

add letter or digit

```
40      }
41
42      // Return a new filtered string
43      return stringBuilder.toString();
44    }
45
46    /** Create a new string by reversing a specified string */
47    public static String reverse(String s) {
48      StringBuilder stringBuilder = new StringBuilder(s);
49      stringBuilder.reverse(); // Invoke reverse in StringBuilder
50      return stringBuilder.toString();
51    }
52  }
```

```
Enter a string: ab<c>cb?a ↵ Enter
Ignoring nonalphanumeric characters,
is ab<c>cb?a a palindrome? true
```

```
Enter a string: abcc><?cab ↵ Enter
Ignoring nonalphanumeric characters,
is abcc><?cab a palindrome? false
```

The `filter(String s)` method (lines 31–44) examines each character in string `s` and copies it to a string builder if the character is a letter or a numeric character. The `filter` method returns the string in the builder. The `reverse(String s)` method (lines 47–51) creates a new string that reverses the specified string `s`. The `filter` and `reverse` methods both return a new string. The original string is not changed.

The program in Listing 5.14 checks whether a string is a palindrome by comparing pairs of characters from both ends of the string. Listing 10.10 uses the `reverse` method in the `StringBuilder` class to reverse the string, then compares whether the two strings are equal to determine whether the original string is a palindrome.

Check Point

10.26 What is the difference between `StringBuilder` and `StringBuffer`?

10.27 How do you create a string builder from a string? How do you return a string from a string builder?

10.28 Write three statements to reverse a string `s` using the `reverse` method in the `StringBuilder` class.

10.29 Write three statements to delete a substring from a string `s` of `20` characters, starting at index `4` and ending with index `10`. Use the `delete` method in the `StringBuilder` class.

10.30 What is the internal storage for characters in a string and a string builder?

10.31 Suppose that `s1` and `s2` are given as follows:

```
StringBuilder s1 = new StringBuilder("Java");
StringBuilder s2 = new StringBuilder("HTML");
```

Show the value of `s1` after each of the following statements. Assume that the statements are independent.

a. `s1.append(" is fun");`
b. `s1.append(s2);`

c. `s1.insert(2, "is fun");`
d. `s1.insert(1, s2);`
e. `s1.charAt(2);`
f. `s1.length();`
g. `s1.deleteCharAt(3);`
h. `s1.delete(1, 3);`
i. `s1.reverse();`
j. `s1.replace(1, 3, "Computer");`
k. `s1.substring(1, 3);`
l. `s1.substring(2);`

10.32 Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    String s = "Java";
    StringBuilder builder = new StringBuilder(s);
    change(s, builder);

    System.out.println(s);
    System.out.println(builder);
  }

  private static void change(String s, StringBuilder builder) {
    s = s + " and HTML";
    builder.append(" and HTML");
  }
}
```

Key Terms

abstract data type (ADT) 366	composition 374
aggregation 374	has-a relationship 374
boxing 383	multiplicity 373
class abstraction 366	stack 378
class encapsulation 366	unboxing 383
class's contract 366	

Chapter Summary

1. The procedural paradigm focuses on designing methods. 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.

2. Many Java methods require the use of objects as arguments. Java offers a convenient way to incorporate, or wrap, a primitive data type into an object (e.g., wrapping `int` into the `Integer` class, and wrapping `double` into the `Double` class).

3. Java can automatically convert a primitive type value to its corresponding wrapper object in the context and vice versa.

4. The `BigInteger` class is useful for computing and processing integers of any size. The `BigDecimal` class can be used to compute and process floating-point numbers with any arbitrary precision.

5. A `String` object is immutable; its contents cannot be changed. To improve efficiency and save memory, the JVM stores two literal strings that have the same character sequence in a unique object. This unique object is called an *interned string object*.

6. A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. You can match, replace, or split a string by specifying a pattern.

7. The `StringBuilder` and `StringBuffer` classes can be used to replace the `String` class. The `String` object is immutable, but you can add, insert, or append new contents into `StringBuilder` and `StringBuffer` objects. Use `String` if the string contents do not require any change, and use `StringBuilder` or `StringBuffer` if they might change.

Quiz

Answer the quiz for this chapter online at www.cs.armstrong.edu/liang/intro10e/quiz.html.

Programming Exercises

MyProgrammingLab™

Sections 10.2–10.3

***10.1** (*The* `Time` *class*) Design a class named `Time`. The class contains:

- The data fields `hour`, `minute`, and `second` that represent a time.
- A no-arg constructor that creates a `Time` object for the current time. (The values of the data fields will represent the current time.)
- A constructor that constructs a `Time` object with a specified elapsed time since midnight, January 1, 1970, in milliseconds. (The values of the data fields will represent this time.)
- A constructor that constructs a `Time` object with the specified hour, minute, and second.
- Three getter methods for the data fields `hour`, `minute`, and `second`, respectively.
- A method named `setTime(long elapseTime)` that sets a new time for the object using the elapsed time. For example, if the elapsed time is `555550000` milliseconds, the hour is `10`, the minute is `19`, and the second is `10`.

Draw the UML diagram for the class and then implement the class. Write a test program that creates two `Time` objects (using `new Time()` and `new Time(555550000)`) and displays their hour, minute, and second in the format hour:minute:second.

(*Hint*: The first two constructors will extract the hour, minute, and second from the elapsed time. For the no-arg constructor, the current time can be obtained using `System.currentTimeMillis()`, as shown in Listing 2.7, ShowCurrentTime.java.)

10.2 (*The* BMI *class*) Add the following new constructor in the BMI class:

```
/** Construct a BMI with the specified name, age, weight,
 * feet, and inches
 */
public BMI(String name, int age, double weight, double feet,
  double inches)
```

10.3 (*The* MyInteger *class*) Design a class named MyInteger. The class contains:

- An int data field named value that stores the int value represented by this object.
- A constructor that creates a MyInteger object for the specified int value.
- A getter method that returns the int value.
- The methods isEven(), isOdd(), and isPrime() that return true if the value in this object is even, odd, or prime, respectively.
- The static methods isEven(int), isOdd(int), and isPrime(int) that return true if the specified value is even, odd, or prime, respectively.
- The static methods isEven(MyInteger), isOdd(MyInteger), and isPrime(MyInteger) that return true if the specified value is even, odd, or prime, respectively.
- The methods equals(int) and equals(MyInteger) that return true if the value in this object is equal to the specified value.
- A static method parseInt(char[]) that converts an array of numeric characters to an int value.
- A static method parseInt(String) that converts a string into an int value.

Draw the UML diagram for the class and then implement the class. Write a client program that tests all methods in the class.

10.4 (*The* MyPoint *class*) Design a class named MyPoint to represent a point with x- and y-coordinates. The class contains:

- The data fields x and y that represent the coordinates with getter methods.
- A no-arg constructor that creates a point (0, 0).
- A constructor that constructs a point with specified coordinates.
- A method named distance that returns the distance from this point to a specified point of the MyPoint type.
- A method named distance that returns the distance from this point to another point with specified x- and y-coordinates.

VideoNote
The MyPoint class

Draw the UML diagram for the class and then implement the class. Write a test program that creates the two points (0, 0) and (10, 30.5) and displays the distance between them.

Sections 10.4–10.8

***10.5** (*Displaying the prime factors*) Write a program that prompts the user to enter a positive integer and displays all its smallest factors in decreasing order. For example, if the integer is 120, the smallest factors are displayed as 5, 3, 2, 2, 2. Use the StackOfIntegers class to store the factors (e.g., 2, 2, 2, 3, 5) and retrieve and display them in reverse order.

***10.6** (*Displaying the prime numbers*) Write a program that displays all the prime numbers less than 120 in decreasing order. Use the StackOfIntegers class to store the prime numbers (e.g., 2, 3, 5, ...) and retrieve and display them in reverse order.

****10.7** (*Game: ATM machine*) Use the **Account** class created in Programming Exercise 9.7 to simulate an ATM machine. Create ten accounts in an array with id **0**, **1**, . . . , **9**, and initial balance $100. The system prompts the user to enter an id. If the id is entered incorrectly, ask the user to enter a correct id. Once an id is accepted, the main menu is displayed as shown in the sample run. You can enter a choice **1** for viewing the current balance, **2** for withdrawing money, **3** for depositing money, and **4** for exiting the main menu. Once you exit, the system will prompt for an id again. Thus, once the system starts, it will not stop.

```
Enter an id: 4 ↵Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 ↵Enter
The balance is 100.0

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 2 ↵Enter
Enter an amount to withdraw: 3 ↵Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 ↵Enter
The balance is 97.0

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 3 ↵Enter
Enter an amount to deposit: 10 ↵Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 ↵Enter
The balance is 107.0

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 4 ↵Enter

Enter an id:
```

*****10.8** (*Financial: the* **Tax** *class*) Programming Exercise 8.12 writes a program for computing taxes using arrays. Design a class named **Tax** to contain the following instance data fields:

- **int filingStatus**: One of the four tax-filing statuses: **0**—single filer, **1**—married filing jointly or qualifying widow(er), **2**—married filing separately, and **3**—head of household. Use the public static constants **SINGLE_FILER** (**0**), **MARRIED_JOINTLY_OR_QUALIFYING_WIDOW(ER)** (**1**), **MARRIED_SEPARATELY** (**2**), **HEAD_OF_HOUSEHOLD** (**3**) to represent the statuses.
- **int[][] brackets**: Stores the tax brackets for each filing status.
- **double[] rates**: Stores the tax rates for each bracket.
- **double taxableIncome**: Stores the taxable income.

Provide the getter and setter methods for each data field and the **getTax()** method that returns the tax. Also provide a no-arg constructor and the constructor **Tax(filingStatus, brackets, rates, taxableIncome)**.

Draw the UML diagram for the class and then implement the class. Write a test program that uses the **Tax** class to print the 2001 and 2009 tax tables for taxable income from $50,000 to $60,000 with intervals of $1,000 for all four statuses. The tax rates for the year 2009 were given in Table 3.2. The tax rates for 2001 are shown in Table 10.1.

TABLE 10.1 2001 United States Federal Personal Tax Rates

Tax rate	*Single filers*	*Married filing jointly or qualifying widow(er)*	*Married filing separately*	*Head of household*
15%	Up to $27,050	Up to $45,200	Up to $22,600	Up to $36,250
27.5%	$27,051–$65,550	$45,201–$109,250	$22,601–$54,625	$36,251–$93,650
30.5%	$65,551–$136,750	$109,251–$166,500	$54,626–$83,250	$93,651–$151,650
35.5%	$136,751–$297,350	$166,501–$297,350	$83,251–$148,675	$151,651–$297,350
39.1%	$297,351 or more	$297,351 or more	$ 148,676 or more	$297,351 or more

****10.9** (*The* **Course** *class*) Revise the **Course** class as follows:

- The array size is fixed in Listing 10.6. Improve it to automatically increase the array size by creating a new larger array and copying the contents of the current array to it.
- Implement the **dropStudent** method.
- Add a new method named **clear()** that removes all students from the course.

Write a test program that creates a course, adds three students, removes one, and displays the students in the course.

***10.10** (*The* **Queue** *class*) Section 10.6 gives a class for **Stack**. Design a class named **Queue** for storing integers. Like a stack, a queue holds elements. In a stack, the elements are retrieved in a last-in first-out fashion. In a queue, the elements are retrieved in a first-in first-out fashion. The class contains:

- An **int[]** data field named **elements** that stores the **int** values in the queue.
- A data field named **size** that stores the number of elements in the queue.
- A constructor that creates a **Queue** object with default capacity **8**.
- The method **enqueue(int v)** that adds **v** into the queue.

- The method `dequeue()` that removes and returns the element from the queue.
- The method `empty()` that returns true if the queue is empty.
- The method `getSize()` that returns the size of the queue.

Draw an UML diagram for the class. Implement the class with the initial array size set to 8. The array size will be doubled once the number of the elements exceeds the size. After an element is removed from the beginning of the array, you need to shift all elements in the array one position the left. Write a test program that adds 20 numbers from 1 to 20 into the queue and removes these numbers and displays them.

***10.11** (*Geometry: the* `Circle2D` *class*) Define the `Circle2D` class that contains:

- Two `double` data fields named `x` and `y` that specify the center of the circle with getter methods.
- A data field `radius` with a getter method.
- A no-arg constructor that creates a default circle with (`0`, `0`) for (`x`, `y`) and `1` for `radius`.
- A constructor that creates a circle with the specified `x`, `y`, and `radius`.
- A method `getArea()` that returns the area of the circle.
- A method `getPerimeter()` that returns the perimeter of the circle.
- A method `contains(double x, double y)` that returns `true` if the specified point (`x`, `y`) is inside this circle (see Figure 10.21a).
- A method `contains(Circle2D circle)` that returns `true` if the specified circle is inside this circle (see Figure 10.21b).
- A method `overlaps(Circle2D circle)` that returns `true` if the specified circle overlaps with this circle (see Figure 10.21c).

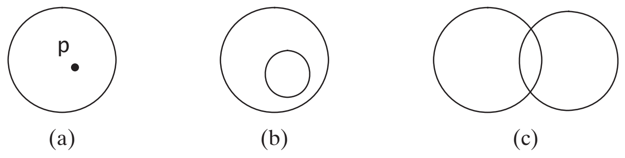


FIGURE 10.21 (a) A point is inside the circle. (b) A circle is inside another circle. (c) A circle overlaps another circle.

Draw the UML diagram for the class and then implement the class. Write a test program that creates a `Circle2D` object `c1` (`new Circle2D(2, 2, 5.5)`), displays its area and perimeter, and displays the result of `c1.contains(3, 3)`, `c1.contains(new Circle2D(4, 5, 10.5))`, and `c1.overlaps(new Circle2D(3, 5, 2.3))`.

*****10.12** (*Geometry: the* `Triangle2D` *class*) Define the `Triangle2D` class that contains:

- Three points named `p1`, `p2`, and `p3` of the type `MyPoint` with getter and setter methods. `MyPoint` is defined in Programming Exercise 10.4.
- A no-arg constructor that creates a default triangle with the points (`0`, `0`), (`1`, `1`), and (`2`, `5`).
- A constructor that creates a triangle with the specified points.
- A method `getArea()` that returns the area of the triangle.
- A method `getPerimeter()` that returns the perimeter of the triangle.
- A method `contains(MyPoint p)` that returns `true` if the specified point `p` is inside this triangle (see Figure 10.22a).

- A method `contains(Triangle2D t)` that returns `true` if the specified triangle is inside this triangle (see Figure 10.22b).
- A method `overlaps(Triangle2D t)` that returns `true` if the specified triangle overlaps with this triangle (see Figure 10.22c).

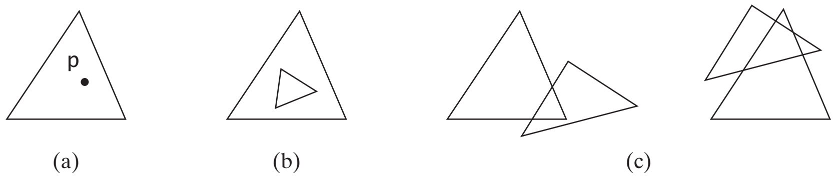


FIGURE 10.22 (a) A point is inside the triangle. (b) A triangle is inside another triangle. (c) A triangle overlaps another triangle.

Draw the UML diagram for the class and then implement the class. Write a test program that creates a `Triangle2D` objects `t1` using the constructor `new Triangle2D(new MyPoint(2.5, 2), new MyPoint(4.2, 3), new MyPoint(5, 3.5))`, displays its area and perimeter, and displays the result of `t1.contains(3, 3)`, `r1.contains(new Triangle2D(new MyPoint(2.9, 2), new MyPoint(4, 1), MyPoint(1, 3.4)))`, and `t1.overlaps(new Triangle2D(new MyPoint(2, 5.5), new MyPoint(4, -3), MyPoint(2, 6.5)))`.

(*Hint*: For the formula to compute the area of a triangle, see Programming Exercise 2.19. To detect whether a point is inside a triangle, draw three dashed lines, as shown in Figure 10.23. If the point is inside a triangle, each dashed line should intersect a side only once. If a dashed line intersects a side twice, then the point must be outside the triangle. For the algorithm of finding the intersecting point of two lines, see Programming Exercise 3.25.)

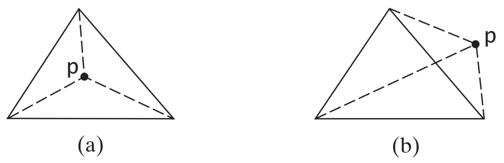


FIGURE 10.23 (a) A point is inside the triangle. (b) A point is outside the triangle.

***10.13** (*Geometry: the* `MyRectangle2D` *class*) Define the `MyRectangle2D` class that contains:

- Two `double` data fields named `x` and `y` that specify the center of the rectangle with getter and setter methods. (Assume that the rectangle sides are parallel to `x-` or `y-` axes.)
- The data fields `width` and `height` with getter and setter methods.
- A no-arg constructor that creates a default rectangle with (`0`, `0`) for (`x`, `y`) and `1` for both `width` and `height`.
- A constructor that creates a rectangle with the specified `x`, `y`, `width`, and `height`.

- A method getArea() that returns the area of the rectangle.
- A method getPerimeter() that returns the perimeter of the rectangle.
- A method contains(double x, double y) that returns true if the specified point (x, y) is inside this rectangle (see Figure 10.24a).
- A method contains(MyRectangle2D r) that returns true if the specified rectangle is inside this rectangle (see Figure 10.24b).
- A method overlaps(MyRectangle2D r) that returns true if the specified rectangle overlaps with this rectangle (see Figure 10.24c).

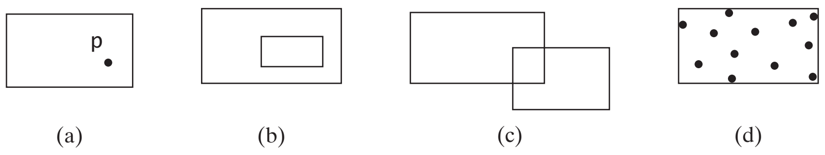


Figure 10.24 A point is inside the rectangle. (b) A rectangle is inside another rectangle. (c) A rectangle overlaps another rectangle. (d) Points are enclosed inside a rectangle.

Draw the UML diagram for the class and then implement the class. Write a test program that creates a MyRectangle2D object r1 (new MyRectangle2D(2, 2, 5.5, 4.9)), displays its area and perimeter, and displays the result of r1.contains(3, 3), r1.contains(new MyRectangle2D(4, 5, 10.5, 3.2)), and r1.overlaps(new MyRectangle2D(3, 5, 2.3, 5.4)).

***10.14** (*The* MyDate *class*) Design a class named MyDate. The class contains:

- The data fields year, month, and day that represent a date. month is 0-based, i.e., 0 is for January.
- A no-arg constructor that creates a MyDate object for the current date.
- A constructor that constructs a MyDate object with a specified elapsed time since midnight, January 1, 1970, in milliseconds.
- A constructor that constructs a MyDate object with the specified year, month, and day.
- Three getter methods for the data fields year, month, and day, respectively.
- A method named setDate(long elapsedTime) that sets a new date for the object using the elapsed time.

Draw the UML diagram for the class and then implement the class. Write a test program that creates two MyDate objects (using new MyDate() and new MyDate(34355555133101L)) and displays their year, month, and day.

(*Hint*: The first two constructors will extract the year, month, and day from the elapsed time. For example, if the elapsed time is 561555550000 milliseconds, the year is 1987, the month is 9, and the day is 18. You may use the GregorianCalendar class discussed in Programming Exercise 9.5 to simplify coding.)

***10.15** (*Geometry: the bounding rectangle*) A bounding rectangle is the minimum rectangle that encloses a set of points in a two-dimensional plane, as shown in Figure 10.24d. Write a method that returns a bounding rectangle for a set of points in a two-dimensional plane, as follows:

```
public static MyRectangle2D getRectangle(double[][] points)
```

The **Rectangle2D** class is defined in Programming Exercise 10.13. Write a test program that prompts the user to enter five points and displays the bounding rectangle's center, width, and height. Here is a sample run:

```
Enter five points: 1.0 2.5 3 4 5 6 7 8 9 10 ↵Enter
The bounding rectangle's center (5.0, 6.25), width 8.0, height 7.5
```

Section 10.9

***10.16** (*Divisible by* **2** *or* **3**) Find the first ten numbers with **50** decimal digits that are divisible by **2** or **3**.

***10.17** (*Square numbers*) Find the first ten square numbers that are greater than **Long.MAX_VALUE**. A square number is a number in the form of n^2. For example, 4, 9, and 16 are square numbers. Find an efficient approach to run your program fast.

***10.18** (*Large prime numbers*) Write a program that finds five prime numbers larger than **Long.MAX_VALUE**.

***10.19** (*Mersenne prime*) A prime number is called a *Mersenne prime* if it can be written in the form $2^p - 1$ for some positive integer p. Write a program that finds all Mersenne primes with $p \leq 100$ and displays the output as shown below. (*Hint*: You have to use **BigInteger** to store the number, because it is too big to be stored in **long**. Your program may take several hours to run.)

```
    p        2^p - 1

    2           3
    3           7
    5          31
...
```

***10.20** (*Approximate e*) Programming Exercise 5.26 approximates e using the following series:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ \ldots \ + \frac{1}{i!}$$

In order to get better precision, use **BigDecimal** with **25** digits of precision in the computation. Write a program that displays the **e** value for **i** = **100**, **200**, . . . , and **1000**.

10.21 (*Divisible by* **5** *or* **6**) Find the first ten numbers greater than **Long.MAX_VALUE** that are divisible by **5** or **6**.

Sections 10.10–10.11

****10.22** (*Implement the* ***String*** *class*) The **String** class is provided in the Java library. Provide your own implementation for the following methods (name the new class **MyString1**):

```
public MyString1(char[] chars);
public char charAt(int index);
public int length();
public MyString1 substring(int begin, int end);
public MyString1 toLowerCase();
public boolean equals(MyString1 s);
public static MyString1 valueOf(int i);
```

****10.23** (*Implement the `String` class*) The `String` class is provided in the Java library. Provide your own implementation for the following methods (name the new class `MyString2`):

```
public MyString2(String s);
public int compare(String s);
public MyString2 substring(int begin);
public MyString2 toUpperCase();
public char[] toChars();
public static MyString2 valueOf(boolean b);
```

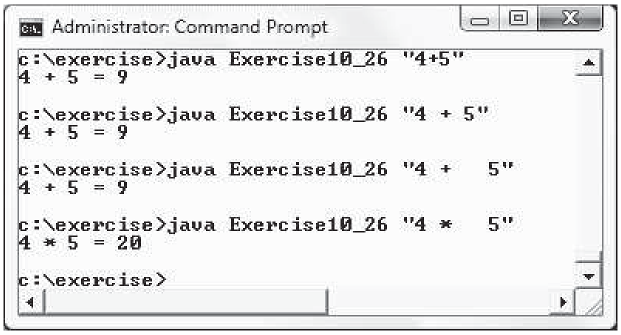
10.24 (*Implement the `Character` class*) The `Character` class is provided in the Java library. Provide your own implementation for this class. Name the new class `MyCharacter`.

****10.25** (*New string `split` method*) The `split` method in the `String` class returns an array of strings consisting of the substrings split by the delimiters. However, the delimiters are not returned. Implement the following new method that returns an array of strings consisting of the substrings split by the matching delimiters, including the matching delimiters.

```
public static String[] split(String s, String regex)
```

For example, `split("ab#12#453", "#")` returns `ab`, `#`, `12`, `#`, `453` in an array of `String`, and `split("a?b?gf#e", "[?#]")` returns `a`, `b`, `?`, `b`, `gf`, `#`, and `e` in an array of `String`.

***10.26** (*Calculator*) Revise Listing 7.9, Calculator.java, to accept an expression as a string in which the operands and operator are separated by zero or more spaces. For example, `3+4` and `3 + 4` are acceptable expressions. Here is a sample run:


Administrator: Command Prompt

```
c:\exercise>java Exercise10_26 "4+5"
4 + 5 = 9

c:\exercise>java Exercise10_26 "4 + 5"
4 + 5 = 9

c:\exercise>java Exercise10_26 "4 +   5"
4 + 5 = 9

c:\exercise>java Exercise10_26 "4 *   5"
4 * 5 = 20

c:\exercise>
```

****10.27** (*Implement the `StringBuilder` class*) The `StringBuilder` class is provided in the Java library. Provide your own implementation for the following methods (name the new class `MyStringBuilder1`):

```
public MyStringBuilder1(String s);
public MyStringBuilder1 append(MyStringBuilder1 s);
public MyStringBuilder1 append(int i);
public int length();
public char charAt(int index);
public MyStringBuilder1 toLowerCase();
public MyStringBuilder1 substring(int begin, int end);
public String toString();
```

**10.28 (*Implement the `StringBuilder` class*) The `StringBuilder` class is provided in the Java library. Provide your own implementation for the following methods (name the new class `MyStringBuilder2`):

```
public MyStringBuilder2();
public MyStringBuilder2(char[] chars);
public MyStringBuilder2(String s);
public MyStringBuilder2 insert(int offset, MyStringBuilder2 s);
public MyStringBuilder2 reverse();
public MyStringBuilder2 substring(int begin);
public MyStringBuilder2 toUpperCase();
```

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 (§§11.7–11.8).
- 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 earlier in the book, the procedural paradigm focuses on designing methods and the object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

why inheritance?

Inheritance is an important and powerful feature for reusing software. Suppose you need to define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so as to avoid redundancy and make the system easy to comprehend and easy to maintain? The answer is to use inheritance.

11.2 Superclasses and Subclasses

Key Point

Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).

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

VideoNote
Geometric class hierarchy

Consider geometric objects. Suppose you want to design the classes to model geometric objects such as circles and rectangles. Geometric objects have many common properties and behaviors. They can be drawn in a certain color and be filled or unfilled. Thus a general class **GeometricObject** can be used to model all geometric objects. This class contains the properties **color** and **filled** and their appropriate getter and setter methods. Assume that this class also contains the **dateCreated** property and the **getDateCreated()** and **toString()** methods. The **toString()** method returns a string representation of the object. Since a circle is a special type of geometric object, it shares common properties and methods with other geometric objects. Thus it makes sense to define the **Circle** class that extends the **GeometricObject** class. Likewise, **Rectangle** can also be defined as a subclass of **GeometricObject**. Figure 11.1 shows the relationship among these classes. A triangular arrow pointing to the superclass is used to denote the inheritance relationship between the two classes involved.

subclass
superclass

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

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

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

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

avoid naming conflicts

Note

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

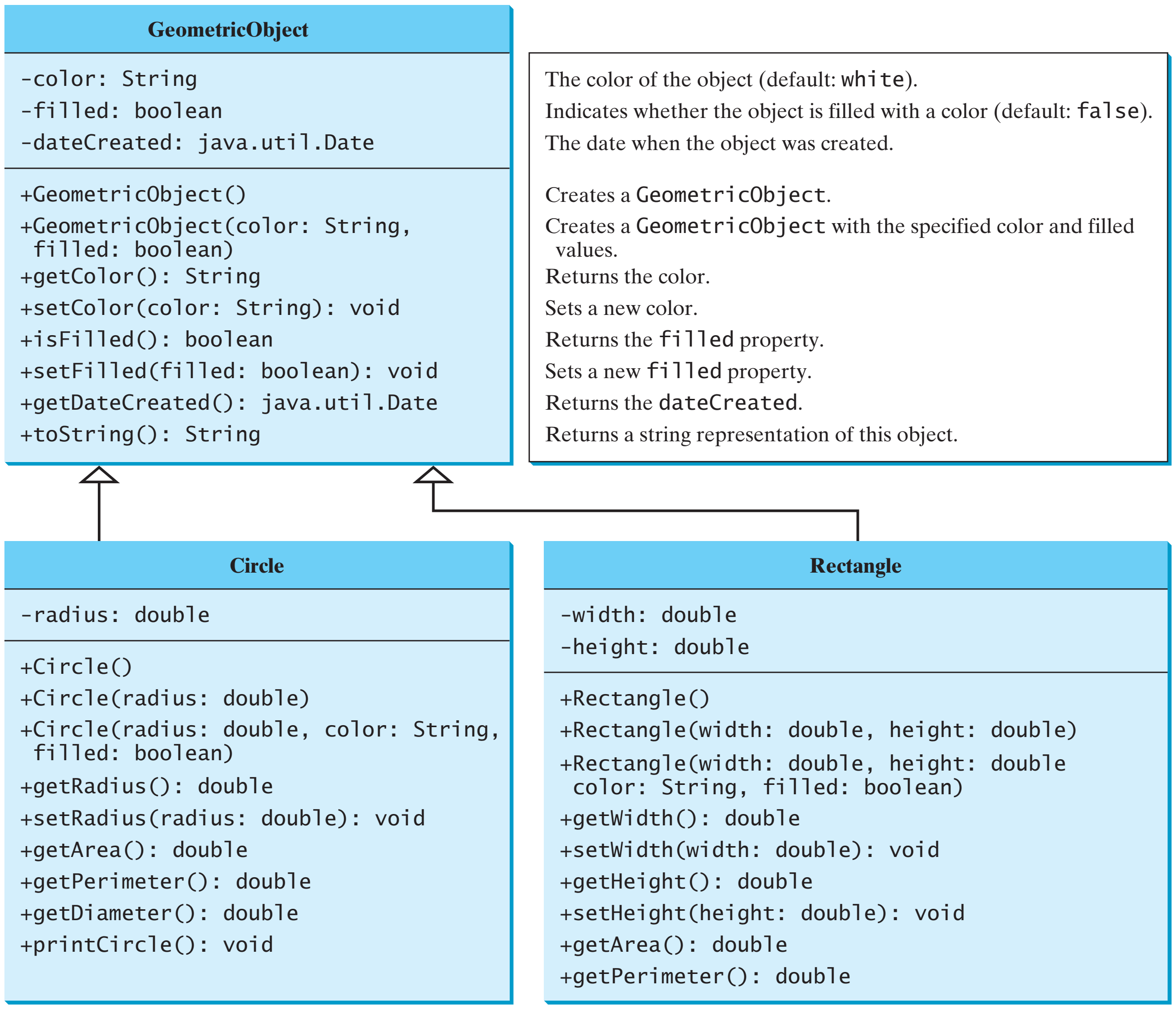

GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String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

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

LISTING 11.1 SimpleGeometricObject.java

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

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

  /** Construct a geometric object with the specified color
    *  and filled value */
  public SimpleGeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;
  }

  /** Return color */
  public String getColor() {
    return color;
  }

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

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

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

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

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

LISTING 11.2 CircleFromSimpleGeometricObject.java

extends superclass
data fields

constructor

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

  public CircleFromSimpleGeometricObject() {
  }

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

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

methods

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

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

Subclass
Superclass

```
public class Circle extends GeometricObject
```

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

The overloaded constructor **Circle(double radius, String color, boolean filled)** is implemented by invoking the **setColor** and **setFilled** methods to set the **color** and **filled** properties (lines 12–17). These two public methods are defined in the superclass **GeometricObject** and are inherited in **Circle**, so they can be used in the **Circle** class.

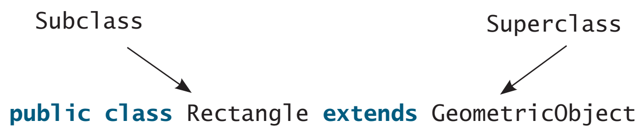
private member in superclass

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

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

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

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



```
public class Rectangle extends GeometricObject
```

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

Listing 11.3 RectangleFromSimpleGeometricObject.java

extends superclass

data fields

constructor

methods

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

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

LISTING 11.4 `TestCircleRectangle.java`

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

`Circle` object
invoke `toString`
invoke `getColor`

`Rectangle` object
invoke `toString`

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

Note the following points regarding inheritance:

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

more in subclass

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

private data fields

nonextensible is-a

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

no blind extension

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

multiple inheritance

single inheritance

- Some programming languages allow you to derive a subclass from several classes. This capability is known as *multiple inheritance*. Java, however, does not allow multiple inheritance. A Java class may inherit directly from only one superclass. This restriction is known as *single inheritance*. If you use the **extends** keyword to define a subclass, it allows only one parent class. Nevertheless, multiple inheritance can be achieved through interfaces, which will be introduced in Section 13.4.

Check Point

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

11.2 What keyword do you use to define a subclass?

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

11.3 Using the super Keyword

Key Point

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

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

Section 9.14, The **this** Reference, introduced the use of the keyword **this** to reference the calling object. The keyword **super** refers to the superclass of the class in which **super** appears. It can be used in two ways:

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

11.3.1 Calling Superclass Constructors

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

The syntax to call a superclass's constructor is:

```
super(), or super(parameters);
```

The statement **super()** invokes the no-arg constructor of its superclass, and the statement **super(arguments)** invokes the superclass constructor that matches the **arguments**. The statement **super()** or **super(arguments)** must be the first statement of the subclass's constructor; this is the only way to explicitly invoke a superclass constructor. For example, the constructor in lines 12–17 in Listing 11.2 can be replaced by the following code:

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

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

Equivalent

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

In any case, constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain. When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor before performing its own tasks. If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its own tasks. This process continues until the last constructor along the inheritance hierarchy is called. This is called *constructor chaining*.

constructor chaining

Consider the following code:

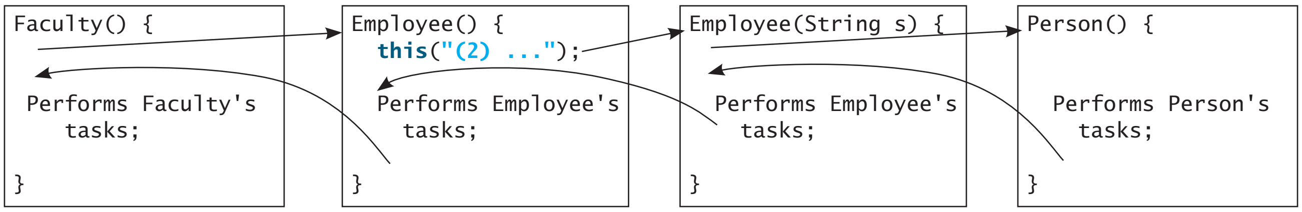
```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Performs Faculty's tasks");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Performs Employee's tasks ");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
```

invoke overloaded constructor

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

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

The program produces the preceding output. Why? Let us discuss the reason. In line 3, `new Faculty()` invokes `Faculty`'s no-arg constructor. Since `Faculty` is a subclass of `Employee`, `Employee`'s no-arg constructor is invoked before any statements in `Faculty`'s constructor are executed. `Employee`'s no-arg constructor invokes `Employee`'s second constructor (line 13). Since `Employee` is a subclass of `Person`, `Person`'s no-arg constructor is invoked before any statements in `Employee`'s second constructor are executed. This process is illustrated in the following figure.



```
Faculty() {          Employee() {            Employee(String s) {    Person() {
                       this("(2) ...");

  Performs Faculty's   Performs Employee's     Performs Employee's     Performs Person's
    tasks;               tasks;                  tasks;                  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(parameters);
```

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

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

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

Check Point

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

```
class A {
  public A() {
    System.out.println(
      "A's no-arg constructor is invoked");
  }
}

class B extends A {
}

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

(a)

```
class A {
  public A(int x) {
  }
}

class B extends A {
  public B() {
  }
}

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

(b)

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

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

11.4 Overriding Methods

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

Key Point

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

method overriding

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

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

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

toString in superclass

The `toString()` method is defined in the `GeometricObject` class and modified in the `Circle` class. Both methods can be used in the `Circle` class. To invoke the `toString` method defined in the `GeometricObject` class from the `Circle` class, use `super.toString()` (line 7).

no super.super.methodName()

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

Several points are worth noting:

override accessible instance method

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

cannot override static method

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

Check Point

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

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

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

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

11.5 Overriding vs. Overloading

Key Point

Overloading means to define multiple methods with the same name but different signatures. Overriding means to provide a new implementation for a method in the subclass.

You learned about overloading methods in Section 6.8. To override a method, the method must be defined in the subclass using the same signature and the same return type.

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

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

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

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

(a)

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

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

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

(b)

When you run the `Test` class in (a), both `a.p(10)` and `a.p(10.0)` invoke the `p(double i)` method defined in class `A` to display `10.0`. When you run the `Test` class in (b), `a.p(10)` invokes the `p(int i)` method defined in class `A` to display `10`, and `a.p(10.0)` invokes the `p(double i)` method defined in class `B` to display `20.0`.

Note the following:

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

override annotation

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

toString in superclass

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

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

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

Check Point

11.11 Identify the problems in the following code:

```java
public class Circle {
  private double radius;

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

  public double getRadius() {
    return radius;
  }

  public double getArea() {
    return radius * radius * Math.PI;
  }
}

class B extends Circle {
  private double length;

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

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

11.12 Explain the difference between method overloading and method overriding.

11.13 If a method in a subclass has the same signature as a method in its superclass with the same return type, is the method overridden or overloaded?

11.14 If a method in a subclass has the same signature as a method in its superclass with a different return type, will this be a problem?

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

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

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

Key Point

Every class in Java is descended from the `java.lang.Object` *class.*

If no inheritance is specified when a class is defined, the superclass of the class is `Object` by default. For example, the following two class definitions are the same:

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

Equivalent

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

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

toString()

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

```
public String toString()
```

string representation

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

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

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

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

Note

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

print object

11.7 Polymorphism

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

Key Point

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

First, let us define two useful terms: subtype and supertype. A class defines a type. A type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*. Therefore, you can say that `Circle` is a subtype of `GeometricObject` and `GeometricObject` is a supertype for `Circle`.

subtype

supertype

The inheritance relationship enables a subclass to inherit features from its superclass with additional new features. A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa. For example, every circle is a geometric object, but not every geometric object is a circle. Therefore, you can always pass an instance of a subclass to a parameter of its superclass type. Consider the code in Listing 11.5.

LISTING 11.5 PolymorphismDemo.java

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

polymorphic call

polymorphic call

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

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

what is polymorphism?

11.8 Dynamic Binding

Key Point

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

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

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

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

declared type

actual type

dynamic binding

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

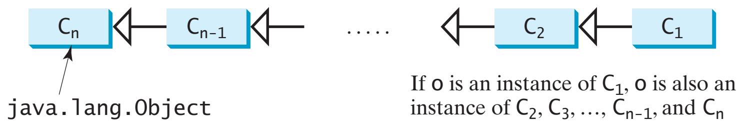

C_n
C_n-1
.....
C_2
C_1
java.lang.Object
If o is an instance of C_1, o is also an instance of C_2, C_3, …, C_n-1, and C_n

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()

```
public class DynamicBindingDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  @Override
  public String toString() {
```

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

override toString()

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

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

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

Invoking `m(new Student())` (line 4) causes the `toString` method defined in the `Student` class to be invoked; invoking `m(new Person())` (line 5) causes the `toString` method defined in the `Person` class to be invoked; and invoking `m(new Object())` (line 6) causes the `toString` method defined in the `Object` class to be invoked.

matching vs. binding

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

Check Point

11.17 What is polymorphism? What is dynamic binding?

11.18 Describe the difference between method matching and method binding.

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

11.20 What is wrong in the following code?

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

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

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

11.21 Show the output of the following code:

```java
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)

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

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

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

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

(b)

11.22 Show the output of following program:

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

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

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

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

11.23 Show the output of following program:

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

```
class A {
  int i = 7;

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

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

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

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

11.9 Casting Objects and the `instanceof` Operator

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

Key Point

casting object

In the preceding section, the statement

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

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

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

implicit casting

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

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

```
Student b = o;
```

explicit casting

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

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

upcasting
downcasting

It is always possible to cast an instance of a subclass to a variable of a superclass (known as *upcasting*), because an instance of a subclass is *always* an instance of its superclass. When casting an instance of a superclass to a variable of its subclass (known as *downcasting*), explicit casting must be used to confirm your intention to the compiler with the **(SubclassName)** cast notation. For the casting to be successful, you must make sure that the object to be cast is an instance of the subclass. If the superclass object is not an instance of the subclass, a runtime

ClassCastException

ClassCastException occurs. For example, if an object is not an instance of **Student**, it cannot be cast into a variable of **Student**. It is a good practice, therefore, to ensure that the object is an instance of another object before attempting a casting. This can be accomplished

instanceof

by using the ***instanceof*** operator. Consider the following code:

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

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

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

lowercase keywords

Note

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

casting analogy

Tip

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

Listing 11.7 demonstrates polymorphism and casting. The program creates two objects (lines 5–6), a circle and a rectangle, and invokes the **displayObject** method to display them (lines 9–10). The **displayObject** method displays the area and diameter if the object is a circle (line 15), and the area if the object is a rectangle (lines 21–22).

LISTING 11.7 CastingDemo.java

```
 1  public class CastingDemo {
 2    /** Main method */
 3    public static void main(String[] args) {
 4      // Create and initialize two objects
 5      Object object1 = new CircleFromSimpleGeometricObject(1);
 6      Object object2 = new RectangleFromSimpleGeometricObject(1, 1);
 7
 8      // Display circle and rectangle
 9      displayObject(object1);
10      displayObject(object2);
11    }
12
13   /** A method for displaying an object */
14   public static void displayObject(Object object) {
15     if (object instanceof CircleFromSimpleGeometricObject) {
16       System.out.println("The circle area is " +
17         ((CircleFromSimpleGeometricObject)object).getArea());
18       System.out.println("The circle diameter is " +
19         ((CircleFromSimpleGeometricObject)object).getDiameter());
20     }
21     else if (object instanceof
22                  RectangleFromSimpleGeometricObject) {
23       System.out.println("The rectangle area is " +
24         ((RectangleFromSimpleGeometricObject)object).getArea());
25     }
26   }
27 }
```

polymorphic call

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 5–6), then invokes the `displayObject` method to display the information on these objects (lines 9–10).

In the `displayObject` method (lines 14–26), explicit casting is used to cast the object to `Circle` if the object is an instance of `Circle`, and the methods `getArea` and `getDiameter` are used to display the area and diameter of the circle.

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

Explicit casting to `Circle` (lines 17, 19) and to `Rectangle` (line 24) is necessary because the `getArea` and `getDiameter` methods are not available in the `Object` class.

Caution

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

precedes casting

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

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

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

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

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

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

Check Point

11.24 Indicate true or false for the following statements:

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

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

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

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

Are the following Boolean expressions true or false?

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

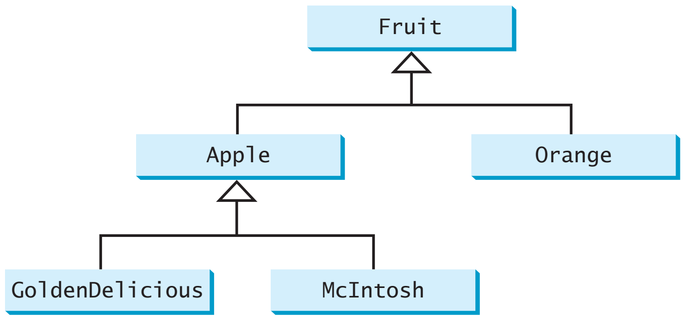
b. Can the following statements be compiled?

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

c. Can the following statements be compiled?

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

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



Assume that the following code is given:

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

Answer the following questions:

a. Is `fruit instanceof Fruit`?

b. Is `fruit instanceof Orange`?

c. Is `fruit instanceof Apple`?

d. Is `fruit instanceof GoldenDelicious`?

e. Is `fruit instanceof McIntosh`?

f. Is `orange instanceof Orange`?

g. Is `orange instanceof Fruit`?

h. Is `orange instanceof Apple`?

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

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

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

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

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

11.27 What is wrong in the following code?

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

class Apple extends Fruit {
}

class Fruit {
}
```

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

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:

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

== vs. equals

Note

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

equals(Object)

Caution

Using the signature equals(SomeClassName obj) (e.g., equals(Circle c)) to override the equals method in a subclass is a common mistake. You should use equals(Object obj). See CheckPoint Question 11.29.

Check Point

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

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

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

```
class Circle {
  double radius;

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

(a)

```
class Circle {
  double radius;

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

(b)

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

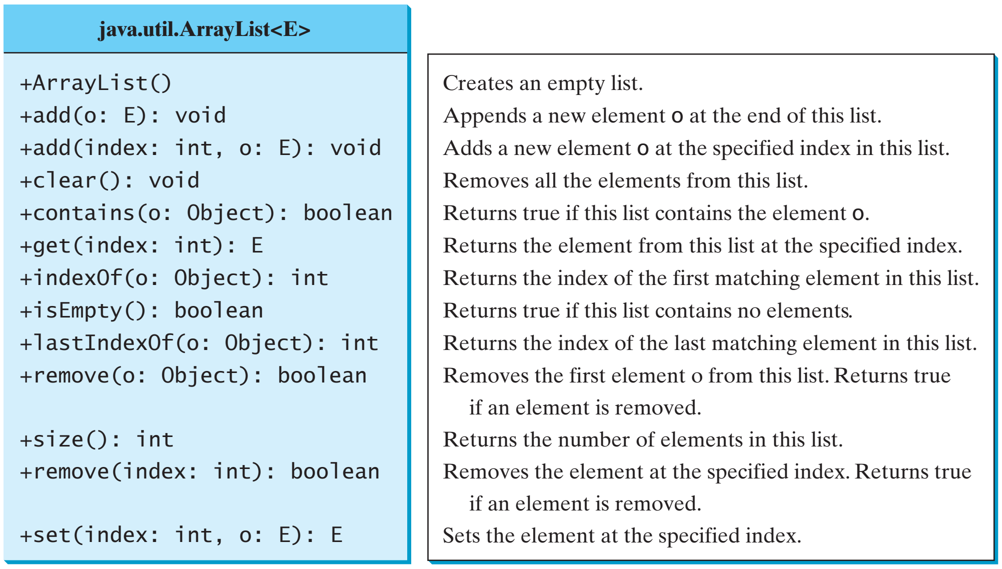
11.11 The ArrayList Class

Key Point

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

VideoNote
The ArrayList class

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



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

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

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

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

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

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

Note

Since JDK 7, the statement

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

can be simplified by

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

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

type inference

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

LISTING 11.8 `TestArrayList.java`

import `ArrayList`

```
import java.util.ArrayList;

```

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

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

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

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

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

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

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

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

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

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

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

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

```
List size? 6
Is Miami in the list? True
The location of Denver in the list? 1
Is the list empty? false
[London, Xian, Paris, Seoul, Tokyo]
Tokyo Seoul Paris Xian London
The area of the circle? 12.566370614359172
```

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

add(Object)

```
[London]
```

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

```
[London, Denver]
```

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

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

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

size()

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

add(index, Object)

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

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

remove(Object)

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

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

remove(index)

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

The statement in line 43 is same as

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

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

toString()

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

get(index)

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

array vs. ArrayList

Once an array is created, its size is fixed. You can access an array element using the square-bracket notation (e.g., `a[index]`). When an `ArrayList` is created, its size is `0`.

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

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

You cannot use the **get(index)** and **set(index, element)** methods if the element is not in the list. It is easy to add, insert, and remove elements in a list, but it is rather complex to add, insert, and remove elements in an array. You have to write code to manipulate the array in order to perform these operations. Note that you can sort an array using the **java.util.Arrays.sort(array)** method. To sort an array list, use the **java.util.Collections.sort(arraylist)** method.

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

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

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

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

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

LISTING 11.9 DistinctNumbers.java

create an array list

contained in list?

add to list

```
 1  import java.util.ArrayList;
 2  import java.util.Scanner;
 3
 4  public class DistinctNumbers {
 5    public static void main(String[] args) {
 6      ArrayList<Integer> list = new ArrayList<>();
 7
 8      Scanner input = new Scanner(System.in);
 9      System.out.print("Enter integers (input ends with 0): ");
10      int value;
11
12      do {
13        value = input.nextInt(); // Read a value from the input
14
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      for (int i = 0; i < list.size(); i++)
21        System.out.print(list.get(i) + " ");
22    }
23  }
```

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

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

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

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

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

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

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

11.30 How do you do the following?

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

b. Append an object to a list?

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

d. Find the number of objects in a list?

e. Remove a given object from a list?

f. Remove the last object from the list?

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

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

11.31 Identify the errors in the following code.

```
ArrayList<String> list = new ArrayList<>();
list.add("Denver");
list.add("Austin");
list.add(new java.util.Date());
String city = list.get(0);
list.set(3, "Dallas");
System.out.println(list.get(3));
```

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

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

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

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

11.34 Explain why the following code is wrong.

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

11.12 Useful Methods for Lists

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

array to array list

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

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

array list to array

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

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

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

sort a list

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

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

max and min methods

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

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

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

shuffle method

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

Check Point

11.35 Correct errors in the following statements:

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

11.36 Correct errors in the following statements:

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

11.13 Case Study: A Custom Stack Class

This section designs a stack class for holding objects.

Key Point

VideoNote
The MyStack class

Section 10.6 presented a stack class for storing **int** values. This section introduces a stack class to store objects. You can use an **ArrayList** to implement **Stack**, as shown in Listing 11.10. The UML diagram for the class is shown in Figure 11.4.

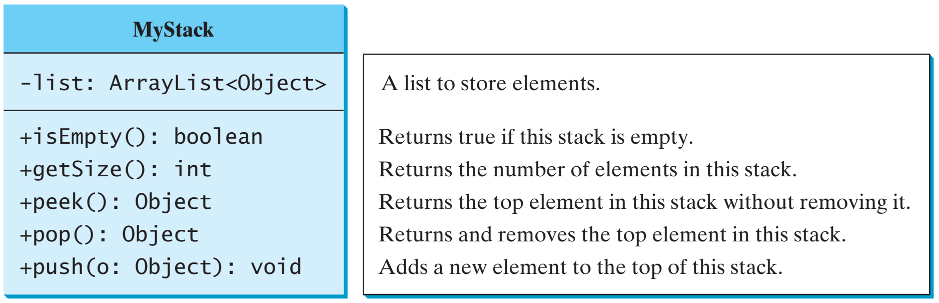

MyStack
-list: ArrayList<Object>
+isEmpty(): boolean
+getSize(): int
+peek(): Object
+pop(): Object
+push(o: Object): void
A list to store elements.
Returns true if this stack is empty.
Returns the number of elements in this stack.
Returns the top element in this stack without removing it.
Returns and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

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

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

array list

stack empty?

get stack size

peek stack

remove

push

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

An array list is created to store the elements in the stack (line 4). The **isEmpty()** method (lines 6–8) returns **list.isEmpty()**. The **getSize()** method (lines 10–12) returns **list.size()**. The **peek()** method (lines 14–16) retrieves the element at the top of the stack without removing it. The end of the list is the top of the stack. The **pop()** method (lines 18–22) removes the top element from the stack and returns it. The **push(Object element)** method (lines 24–26) adds the specified element to the stack. The **toString()** method (lines 28–31) defined in the **Object** class is overridden to display the contents of the stack by invoking **list.toString()**. The **toString()** method implemented in **ArrayList** returns a string representation of all the elements in an array list.

composition
is-a
has-a

Design Guide

In Listing 11.10, **MyStack** contains **ArrayList**. The relationship between **MyStack** and **ArrayList** is *composition*. While inheritance models an *is-a* relationship, composition models a *has-a* relationship. You could also implement **MyStack** as a subclass of **ArrayList** (see Programming Exercise 11.10). Using composition is better, however, because it enables you to define a completely new stack class without inheriting the unnecessary and inappropriate methods from **ArrayList**.

11.14 The **protected** Data and Methods

Key Point

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

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

why protected?

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

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

$$\xrightarrow{\text{Visibility increases}}$$

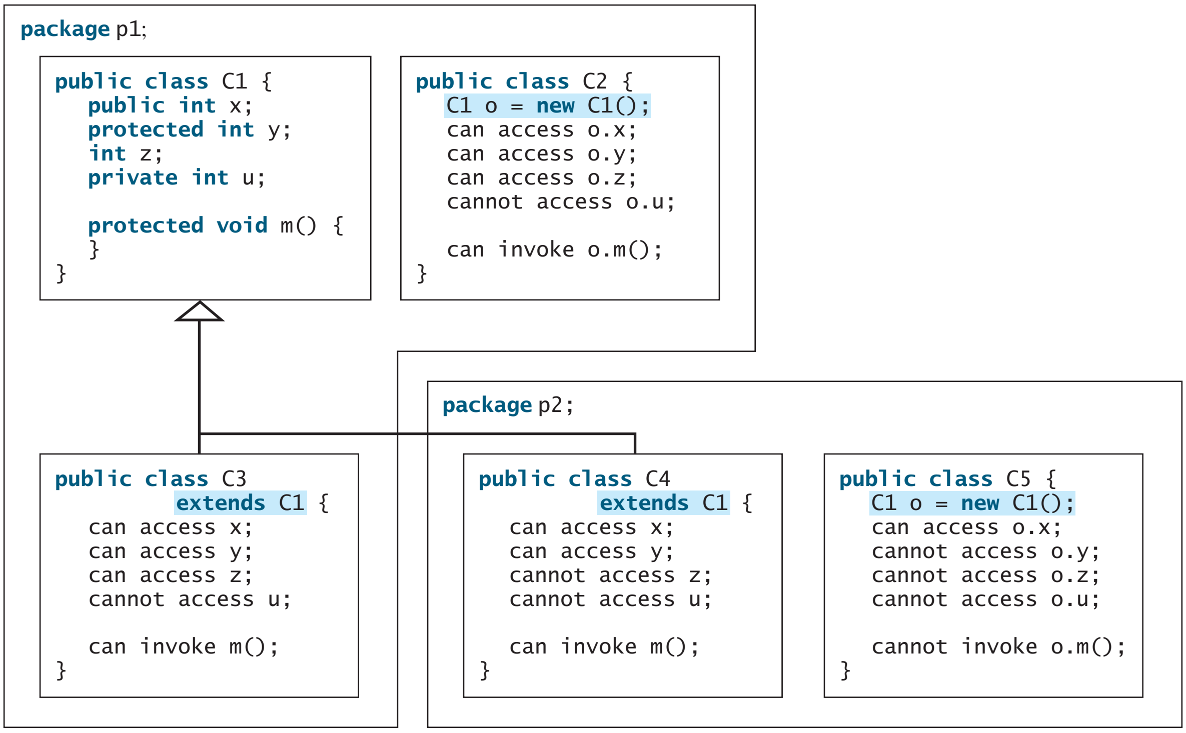
private, default (no modifier), protected, public

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

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

TABLE 11.2 Data and Methods Visibility

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



```
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

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

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

11.39 In the following code, the classes `A` and `B` are in the same package. If the question marks in (a) are replaced by blanks, can class `B` be compiled? If the question marks are replaced by `private`, can class `B` be compiled? If the question marks are replaced by `protected`, can class `B` be compiled?

```
package p1;

public class A {
  ___?___ int i;

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

(a)

```
package p1;

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

(b)

11.40 In the following code, the classes `A` and `B` are in different packages. If the question marks in (a) are replaced by blanks, can class `B` be compiled? If the question marks are replaced by `private`, can class `B` be compiled? If the question marks are replaced by `protected`, can class `B` be compiled?

```
package p1;

public class A {
  ___?___ int i;

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

(a)

```
package p2;

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

(b)

11.15 Preventing Extending and Overriding

Key Point

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

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

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

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

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

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

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

Note

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

Check Point

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

11.42 Indicate true or false for the following statements:

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

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

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

d. A final class can have instances.

e. A final class can be extended.

f. A final method can be overridden.

Key Terms

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

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 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 that a class is final and cannot be extended and to indicate that a method is final and cannot be overridden.

Quiz

Answer the quiz for this chapter online at www.cs.armstrong.edu/liang/intro10e/quiz.html.

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

For the formula to compute the area of a triangle, see 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 email address. A student has a class status (freshman, sophomore, junior, or senior). Define the status as a constant. An employee has an office, salary, and date hired. 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 then 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. The `Transaction` class is defined as shown in Figure 11.6.

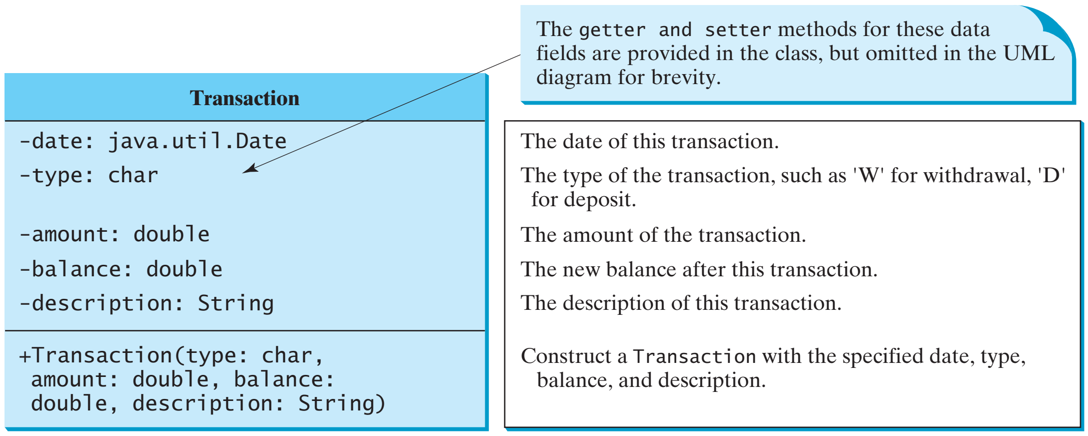

The getter and setter methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
Transaction
-date: java.util.Date
-type: char
-amount: double
-balance: double
-description: String
+Transaction(type: char, amount: double, balance: double, description: String)
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 account holder name, interest rate, balance, and all transactions.

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

```
Enter the array size n: 4 ↵
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 and 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 5 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 5 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 separated by exactly one space. Here is a sample run:

```
Enter ten integers: 34 5 3 5 6 4 33 2 2 4 ↵
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 union 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 in the output. Here is a sample run:

```
Enter five integers for list1: 3 5 45 4 3
Enter five integers for list2: 33 51 5 4 13
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, then enter the points clockwise, and display the area of the polygon. Here is a sample run of the program:

```
Enter the number of the points: 7
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 -3.5
The total area is 250.075
```

****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:

```
What is 5 + 9? 12
Wrong answer. Try again. What is 5 + 9? 34
Wrong answer. Try again. What is 5 + 9? 12
You already entered 12
Wrong answer. Try again. What is 5 + 9? 14
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, 5 in an array list. 2 and 5 appear an odd number of times in the array list. So, `n` is 10.) Here are sample runs:

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

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