

Name: _____ () Date: _____

Chapter 9: Designing Classes

Lesson Objectives

After completing the lesson, the student will be able to

- *understand the principles of object-oriented programming*
- *describe objects and classes, and use classes to model objects*
- *use UML graphical notation to describe classes and objects*
- *demonstrate how to define classes and create objects*
- *create objects using constructors*
- *access objects via object reference variables*
- *define a reference variable using a reference type*
- *access an object's data and methods using the object member access operator*
- *define data fields of reference types and assign default values for an object's data fields*
- *distinguish between instance and static variables and methods*
- *define private data fields with appropriate getter and setter methods*
- *define toString() methods to return a string representation of an object*
- *encapsulate data fields to make classes easy to maintain*
- *develop methods with object arguments and differentiate between primitive-type arguments and object-type arguments*
- *determine the scope of variables in the context of a class*
- *use the keyword this to refer to the calling object itself*
- *create objects for primitive values using the wrapper classes*
- *simplify programming using automatic conversion between*
- *primitive types and wrapper class types*
- *apply class abstraction to develop software*
- *discover the relationships between classes*
- *design programs using the object-oriented paradigm*

9.1 The OOP Principles

OOP is a programming methodology that helps to organize and build complex programs through the use of principles namely; **encapsulation**, **abstraction**, **inheritance** and **polymorphism**. Java uses these principles to provide great flexibility, modularity, and reusability in developing software.

9.1.1 Encapsulation

Encapsulation is the mechanism that binds together methods and the data it manipulates, and keeps both safe from outside interference and misuse. Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition. Typically, only the object's own methods (**accessors** and **mutators**) can directly inspect or manipulate its data member.

An accessor is a method that is used to ask an object about itself. In OOP, these are usually in the form of properties, which have a `get()` method, which is an accessor method. However, accessor methods are not restricted to properties and can be any public method that gives information about the state of the object.

A mutator is a method that is used to modify the state of an object, while hiding the implementation of exactly how the data gets modified. It is the `set()` method that let the caller modify the data members behind the scenes.

Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state. Encapsulation leads to information hiding. A benefit of encapsulation is that it can reduce system complexity.

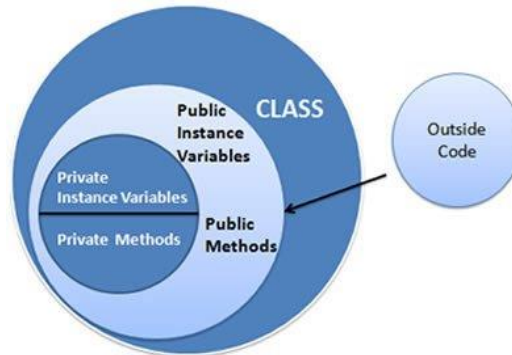


Figure 9.1 Illustration of a Class Encapsulation

One way to think about encapsulation is as a protective wrapper that prevents the methods and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the methods and data inside the wrapper is tightly controlled through a well-defined interface. See Figure 9.1.

In Java, the basis of encapsulation is the class. A class defines the structure and behavior (data and methods) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, **objects are referred to as instances of a class**. Thus, a class is a logical construct; an object has physical reality. When you create a class, you will specify the methods and data that constitute that class. **Collectively, these elements are called members of the class**. Specifically, **the data defined by the class are referred to as member data or instance variables**. **The code that operates on that data is referred to as member methods or just methods**. An object can be viewed as a black box, where you do not need to know the internal workings. However, you are still able to interact with it through its interfaces (methods). A good example is the mobile phone.

In properly written Java programs, the methods define how the member variables can be used. **The behavior and interface of a class are defined by the methods that operate on its instance data**. Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. **Each method or member variable in a class may be marked private or public**. **The public interface of a class represents everything that external users of the class need to know, or may know**. **The private methods and data can only be accessed by code that is a member of the class**. Therefore, **any other code that is not a member of the class cannot access a private method or variable**.

Since the private members of a class may only be accessed by other parts of your program through the class' public methods, you have to ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class.

9.1.2 Abstraction

Abstraction denotes a model, a view, or some other focused representation for an actual item. Abstraction is the process of removing characteristics from something in order to reduce it to a set of essential characteristics and functionality. Through the process of abstraction, a programmer hides all but the relevant characteristics about an object in order to reduce complexity and increase efficiency. In java, we use abstract class and interface to achieve abstraction. For example: phone call, we do not need to know the internal processing.

In the same way that abstraction sometimes works in abstract art. The object that remains is a representation of the original, with unwanted detail omitted. The resulting object itself can be referred to as an abstraction, meaning a named entity made up of selected attributes and behavior specific to a particular usage of the originating entity. Abstraction is related to both encapsulation and data hiding.

In the process of abstraction, the programmer tries to ensure that the entity is named in a manner that will make sense and that it will have all the relevant aspects included and none of the extraneous ones. A real-world analogy of abstraction might work like this: You (the object) are arranging to meet a blind date and are deciding what to tell them so that they can recognize you in the restaurant.

In short, data abstraction is nothing more than the implementation of an object that contains the same essential properties and actions we can find in the original object we are representing.

9.1.3 Inheritance

Inheritance is the process by which one object acquires the properties of another object. Inheritance is a way to reuse code of existing objects, or to establish a sub-type from an existing object, or both, depending upon programming language support. In classical inheritance where objects are defined by classes, classes can inherit attributes and behavior from pre-existing classes called **base classes, superclasses, parent classes or ancestor classes**. **The resulting classes are known as derived classes, subclasses or child classes**. The relationships of classes through inheritance give rise to a hierarchy.

A subclass is a modular, derivative class that inherits one or more properties from another class (called the superclass). The properties commonly include class data variables, properties, and methods or functions. The superclass establishes a common interface and foundational functionality, which specialized subclasses can inherit, modify, and supplement. The software inherited by a subclass is considered reused in the subclass. In some cases, a subclass may customize or redefine a method inherited from the superclass.

9.1.4 Polymorphism

Polymorphism (from Greek, meaning “many forms”) is a feature that allows one interface (one name) to be used for a general class of actions. **Polymorphism manifests itself by having multiple methods all with the same name, but slightly different functionality.**

There are 2 basic types of polymorphism:

Method overloading - means to define multiple methods with the same name but different signatures.

Method overriding - means to provide a new implementation for a method in the subclass. For method overriding to take place, there must be inheritance taking place.

Inheritance and Polymorphism will be covered in OOPII.

9.2 Class Design

The Circle class in Figure 9.2 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 (template) for Circle objects. The illustration of class templates and objects can be standardized using Unified Modeling Language (UML) notation as shown in Figure 9.3.

```

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;
    }
}

```

Annotations in the diagram:

- Data field:** Points to the `double radius = 1;` line.
- Constructors:** Points to the `Circle()` and `Circle(double newRadius)` blocks.
- Method:** Points to the `getArea()`, `getPerimeter()`, and `setRadius()` blocks.

Figure 9.2 A Circle class definition with constructors and methods

In the class diagram:

The data field is denoted as `dataFieldName: dataType`

The constructor is denoted as `ClassName(parameterName: parameterType)`

The method is denoted as `methodName(parameterName: parameterType): returnType`

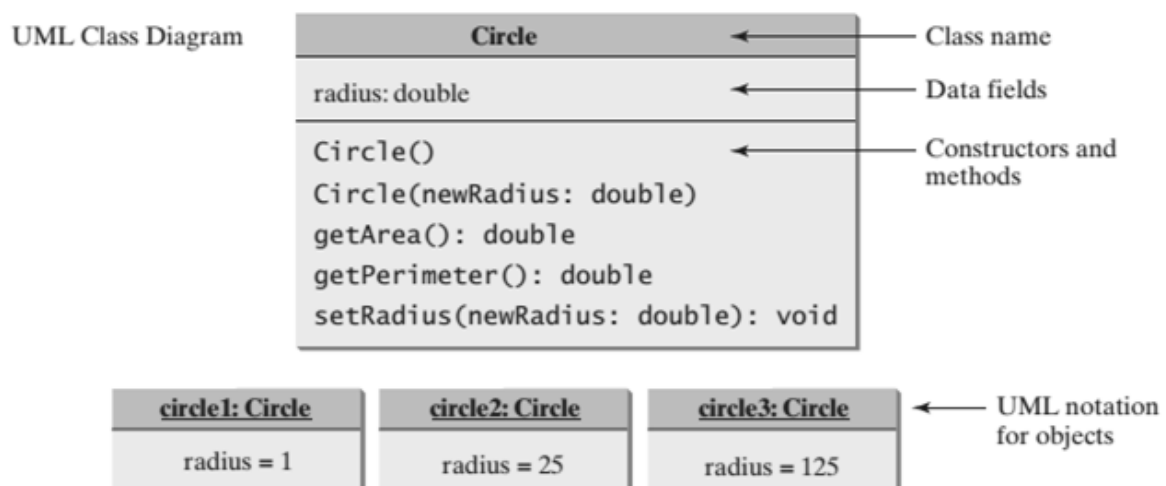


Figure 9.3 Classes and objects can be represented using UML notation.

```

1 public class TestCircle {
2     public static void main(String[] args) {
3
4         Circle c1 = new Circle(); // Create a circle object with radius 1
5         System.out.println("The area of the circle of radius "
6             + c1.radius + " is " + c1.getArea());
7
8         Circle c2 = new Circle(25); // Create a circle object with radius 25
9         System.out.println("The area of the circle of radius "
10            + c2.radius + " is " + c2.getArea());
11        // Create a circle object with c2 content using copy constructor
12        Circle c3 = new Circle(c2);
13
14        c2.radius = 100; // or c2.setRadius(100) // Modify c2 object's radius
15        System.out.println("The area of the circle of radius "
16            + c2.radius + " is " + c2.getArea());
17
18        System.out.println("The area of the circle (copy of c2) of radius"
19            + c3.radius + " is " + c3.getArea());
20    }
21 }

```

PROGRAM 9-1A

```

1 public class Circle {
2     double radius;
3
4     Circle() { // No arg constructor
5         radius = 1;
6     }
7     // Construct a circle with a specified radius
8     Circle(double newRadius) { //Overloaded constructor
9         radius = newRadius;
10    }
11    Circle(Circle original){ // Copy constructor
12        radius = original.radius;
13    }
14    double getArea() { // Return the area of this circle
15        return radius * radius * Math.PI;
16    }
17    double getPerimeter() { // Return the perimeter of this circle
18        return 2 * radius * Math.PI;
19    }
20    void setRadius(double newRadius) { // Set a new radius for this circle
21        radius = newRadius;
22    }
23 }

```

PROGRAM 9-1B

PROGRAM OUTPUT

```

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 100.0 is 31415.926535897932
The area of the circle (copy of c2) of radius 25.0 is 1963.4954084936207

```

The program contains two classes. The first of these, `TestCircle`, is the `main` class (test driver). Its sole purpose is to test the second class, `Circle`. 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.

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 `TestCircle.java`, since `TestCircle` is public. Each class in the source code is compiled into a `.class` file. When you compile `TestCircle.java`, two class files `TestCircle.class` and `Circle.class` are generated, as shown in Figure 9.4.

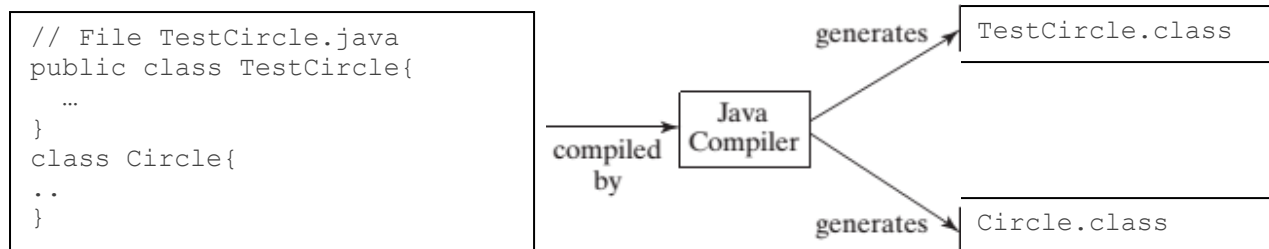


Figure 9.4 Each class in the source code file is compiled into a `.class` file.

9.3 Constructing Objects Using Constructors

A constructor is invoked to create an object using the `new` operator. Constructors are a special kind of method. They have three peculiarities:

- A constructor must have the same name as the class itself.
- Constructors do not have a return type—not even `void`.
- Constructors are invoked using the `new` operator when an object is created.

Constructors play the role of initializing objects. 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,

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

In this case, `Circle()` is a method, not a constructor. 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 (no-arg constructor) defined in the `Circle` class, and `new Circle(25)` creates an object using the second constructor (overloaded 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-argument 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.**

9.4 Copy Constructor

A copy constructor is a constructor with a single argument of the same type as the class. The copy constructor should create an object that is a separate, independent object but with the instance variables set so that it is an exact copy of the argument object.

PROGRAM 9-1A, line 12 uses the copy constructor defined in PROGRAM 9-1B. The copy constructor, or any other constructor, creates a new object of the class `Circle`. This happens automatically and is not shown in the code for the copy constructor. The code for the copy constructor then goes on to set the instance variables to the values equal to those of its one parameter, `original`. But the new date created is a separate object even though it represents the same date. Consider the following code:

```
Circle circle3 = new Circle(circle2);
```

After this code is executed, both `circle2` and `circle` represent the circle with radius 25, but they are two different objects. So, if we change one of these objects, it will not change the other.

9.5 Accessing Objects via Reference Variables

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 heap memory. They can be accessed via reference variables.

9.5.1 Reference Variables and Reference Types

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;
```

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. 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.

9.5.2 Accessing an Object's Data and Methods

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.

Recall that you use `Math.methodName(arguments)` (e.g., `Math.pow(3, 2.5)`) to invoke a method in the `Math` class. You cannot invoke `getArea()` using `Circle.getArea()`. **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 non-static. It must be invoked from an object using `objectRefVar.methodName(arguments)` (e.g., `myCircle.getArea()`)

9.5.3 Anonymous Object

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**.

9.6 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. 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. **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.** 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 9.5. The `Circle` class defines the instance variable `radius`, the static variable `numberOfObjects`, the instance methods `getRadius()`, `setRadius()`, `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:

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

UML Notation:
underline: static variables or methods

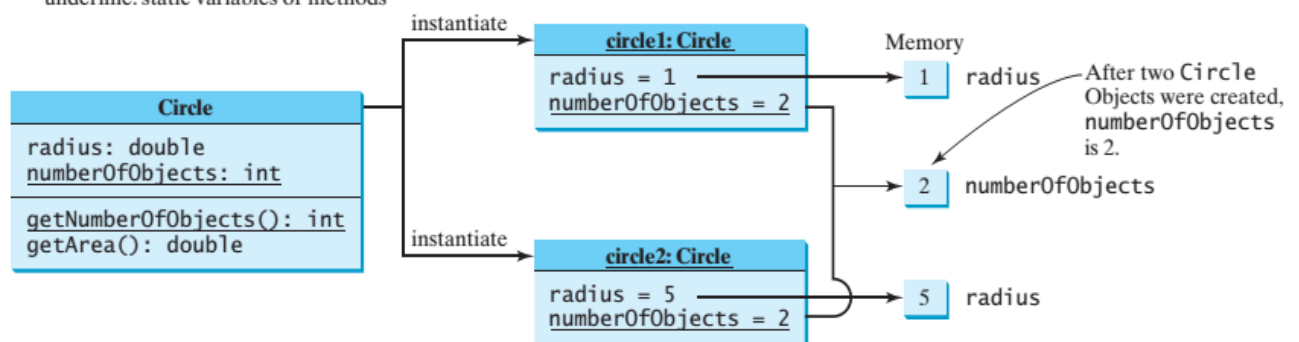


Figure 9.5 Instance variables belong to the instances and have memory storage independent of one another. Static variables are shared by all the instances of the same class.

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 `CircleStatic`, is defined in PROGRAM 9-2A:

<pre> 1 public class CircleStatic { 2 double radius; 3 static int numberOfObjects = 0; // The number of objects created 4 CircleStatic() { // Construct a circle with radius 1 5 radius = 1; 6 numberOfObjects++; 7 } 8 // Construct a circle with a specified radius 9 CircleStatic(double newRadius) { 10 radius = newRadius; 11 numberOfObjects++; 12 } 13 static int getNumberOfObjects() { // Return numberOfObjects 14 return numberOfObjects; 15 } 16 double getArea() { // Return the area of this circle 17 return radius * radius * Math.PI; 18 } 19 }</pre>	PROGRAM 9-2A
--	---------------------

Method `getNumberOfObjects()` in `CircleStatic` is a static method. 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. PROGRAM 9-2B demonstrates how to use static variables and methods.

1	<code>public class TestCircleStatic{</code>	PROGRAM 9-2B
2	<code> public static void main(String[] args) {</code>	
3	<code> System.out.println("Before creating objects");</code>	
4	<code> System.out.println("The number of Circle objects is " +</code>	
5	<code> CircleStatic.numberOfObjects);</code>	
6		
7	<code> CircleStatic c1 = new CircleStatic();// Create c1</code>	
8		
9	<code> // Display c1 BEFORE c2 is created</code>	
10	<code> System.out.println("\nAfter creating c1");</code>	
11	<code> System.out.println("c1: radius (" + c1.radius + ") and number of</code>	
12	<code>Circle</code>	
13	<code>objects (" + c1.numberOfObjects + ")");</code>	
14		
15	<code> CircleStatic c2 = new CircleStatic(5); // Create c2</code>	
16	<code> c1.radius = 9; // Modify c1</code>	
17		
18	<code> // Display c1 and c2 AFTER c2 was created</code>	
19	<code> System.out.println("\nAfter creating c2 and modifying c1");</code>	
20	<code> System.out.println("c1: radius (" + c1.radius + ") and number of</code>	
21	<code>Circle</code>	
22	<code>objects (" + c1.numberOfObjects + ")");</code>	
23	<code> System.out.println("c2: radius (" + c2.radius + ") and number of</code>	
24	<code>Circle</code>	
	<code>objects (" + c2.numberOfObjects + ")");</code>	
	<code> }</code>	
	<code>}</code>	

PROGRAM OUTPUT

```
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 `TestCircleStatic.java`, the Java compiler automatically compiles `CircleStatic.java` if it has not been compiled since the last change. Static variables and methods can be accessed without creating objects. Line 4 displays the number of objects, which is 0, since no objects have been created.

The `main` method creates two circles, `c1` and `c2` (lines 7, 14). The instance variable `radius` in `c1` is modified to become 9 (line 16). 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 7), and it becomes 2 after `c2` is created (line 14).

`c1.numberOfObjects` (line 21) and `c2.numberOfObjects` (line 23) are better replaced by `CircleStatic.numberOfObjects`. This improves readability, because other programmers can easily recognize the static variable. You can also replace `CircleStatic.numberOfObjects` with `CircleStatic.getNumberOfObjects()`.

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.** The relationship between static and instance members are summarized in the following diagram:

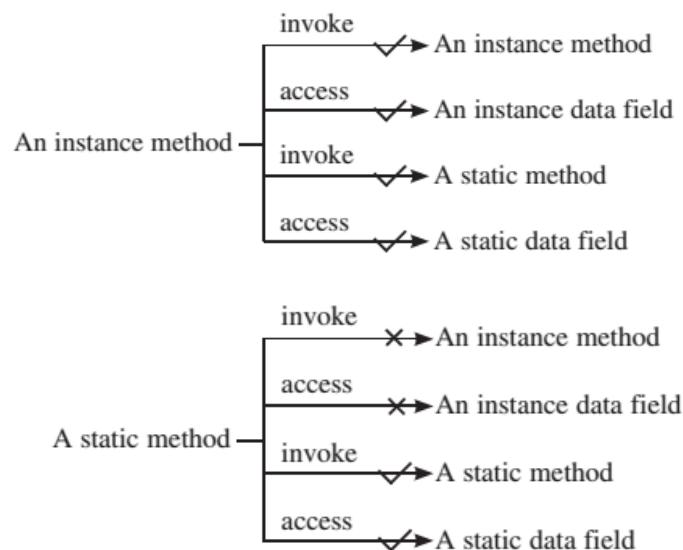


Figure 9.6 Relationship between static and instance members.

9.7 Visibility Modifiers

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

In addition to the `public` and default visibility modifiers, **Java provides the `private` and `protected` modifiers for class members.** The `private` modifier makes methods and data fields accessible only from within its own class. Figure 9.7 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.

<pre>package p1; public class C1 { public int x; int y; private int z; public void m1() { } void m2() { } private void m3() { } }</pre>	<pre>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(); } }</pre>	<pre>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(); } }</pre>
---	---	---

Figure 9.7 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 9.8, C1 can be accessed from C2 but not from C3.

<pre>package p1; class C1 { ... }</pre>	<pre>package p1; public class C2 { can access C1 }</pre>	<pre>package p2; public class C3 { cannot access C1; can access C2; }</pre>
--	---	--

Figure 9.8 A non-public class has package-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 9.9b, an object `c` of class `C` cannot access its private members, because `c` is in the `Test` class. As shown in Figure 9.9a, an object `c` of class `C` can access its private members, because `c` is defined inside its own class.

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

(a) This is okay because object `c` is used inside the class `C`. (b) This is wrong because `x` and `convert` are private in class `C`.

Figure 9.9 An object can access its private members if it is defined in its own class.

Note: 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.

9.8 Data Field Encapsulation

Making data fields private protects data and makes the class easy to maintain. The data fields `radius` and `numberOfObjects` in the `CircleStatic` class in PROGRAM 9-2A can be modified directly (e.g., `c1.radius = 5` or `CircleStatic.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., `CircleStatic.numberOfObjects = 10`).
- Second, the class becomes difficult to maintain and vulnerable to bugs. Suppose you want to modify the `CircleStatic` class to ensure that the radius is non-negative after other programs have already used the class. You have to change not only the `CircleStatic` class but also the programs that use it, because the clients may have modified the radius directly (e.g., `c1.radius = -5`).

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 referred to as a 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:

```
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 9.10. The new `Circle` class, named `CirclePrivate`, is defined in PROGRAM 9-3A:

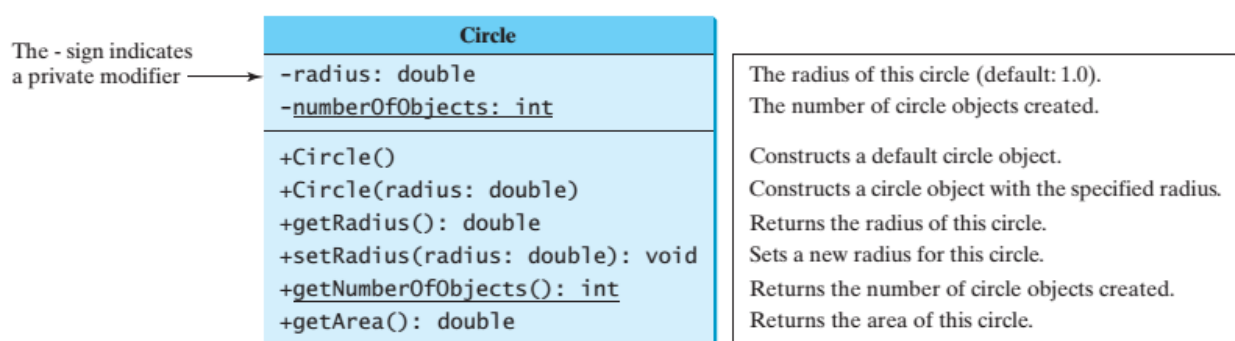


Figure 9.10 The `Circle` class encapsulates circle properties and provides getter/setter methods.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23	<pre> public class CirclePrivate { private double radius = 1; private static int numberOfObjects = 0; public CirclePrivate() { // Construct a circle with radius 1. numberOfObjects++; } public CirclePrivate(double newRadius) { radius = newRadius; numberOfObjects++; } public double getRadius() { return radius; // Return radius } public void setRadius(double newRadius) { radius = (newRadius >= 0) ? newRadius : 0; // Set a new radius } public static int getNumberOfObjects() { return numberOfObjects; // Return numberOfObjects } public double getArea() { return radius * radius * Math.PI; // Return the area of this circle } } </pre>	PROGRAM 9-3A
---	--	---------------------

The `getRadius()` method (lines 13–15) returns the radius, and the `setRadius(newRadius)` method (line 16–18) 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 do not need to change the client programs. This makes the class easy to maintain. PROGRAM 9-3B gives a client program that uses the `Circle` class to create a `Circle` object and modifies the radius using the `setRadius()` method.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	<pre> public class TestCirclePrivate { public static void main(String[] args) { // Create a circle with radius 5.0 CirclePrivate myCircle = new CirclePrivate(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 " + CirclePrivate.getNumberOfObjects()); } } </pre>	PROGRAM 9-3B
---	---	---------------------

PROGRAM OUTPUT

```

The area of the circle of radius 5.0 is 78.53981633974483
The area of the circle of radius 5.5 is 95.03317777109125
The number of objects created is 1

```

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.

9.9 Passing Objects to Methods

Passing an object to a method is to pass the reference of the object. PROGRAM 9-4 passes the `myCircle` object as an argument to the `printCircle()` method:

<pre> 1 public class passObject { 2 public static void main(String[] args) { 3 // CirclePrivate is defined in PROGRAM 8-2A 4 CirclePrivate myCircle = new CirclePrivate(5.0); 5 printCircle(myCircle); 6 } 7 8 public static void printCircle(CirclePrivate c) { 9 System.out.println("The area of the circle of radius " 10 + c.getRadius() + " is " + c.getArea()); 11 } 12 } 13 </pre>	PROGRAM 9-4
---	--------------------

PROGRAM OUTPUT

The area of the circle of radius 5.0 is 78.53981633974483

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.** PROGRAM 9-5 demonstrates the difference between passing a primitive type value and passing a reference value.

<pre> 1 public class Test { 2 public static void main(String[] args){ 3 CirclePrivate myCircle = 4 new CirclePrivate(1); 5 int n = 5; 6 printAreas(myCircle, n); // Print areas for radius 1, 2, 3, 4, and 7 5. 8 System.out.println("\n" + "Radius is " + myCircle.getRadius()); 9 System.out.println("n is " + n); 10 } 11 12 public static void printAreas(// Print a table of areas for radius 13 CirclePrivate c, int times){ 14 System.out.println("Radius \t\tArea"); 15 while (times >= 1) { 16 System.out.println(c.getRadius() + "\t\t" + c.getArea()); 17 c.setRadius(c.getRadius() + 1); 18 times--; 19 } 20 } 21 } </pre>	PROGRAM 9-5
---	--------------------

PROGRAM OUTPUT

```

Radius Area
1.0 3.141592653589793
2.0 12.566370614359172
3.0 29.274333882308138
4.0 50.26548245743669
5.0 78.53981633974483
Radius is 6.0
n is 5

```

The `CirclePrivate` class is defined in PROGRAM 9-2A. The program passes a `CirclePrivate` 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.

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`.

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.

9.10 Design Guide (Optional Reading)

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. It is a common design error to define an instance method that should have been defined as static.

9.11 Define toString() method

If you want to represent any object as a string, `toString()` method comes into existence. The `toString()` method returns the string representation of the object. The `Object` class `toString()` method converts an `Object` into a `String` that contains information about the `Object`.

If you print any object, java compiler internally invokes the `toString()` method on the object. So overriding the `toString()` method, returns the desired output, it can be the state of an object etc. depends on your implementation.

```
1  class Student{
2      private int index;
3      private String name;
4      private String city;
5
6      Student(int index, String name, String city){
7          this.index=index;
8          this.name=name;
9          this.city=city;
10     }
11     public String toString(){ //overriding the toString() method
12         return index + " " + name + " " + city;
13     }
14     public static void main(String args[]){
15         Student s1=new Student(101, "Raj", "London");
16         Student s2=new Student(102, "Vijay", "New York");
17         System.out.println(s1); //compiler writes here s1.toString()
18         System.out.println(s2); //compiler writes here s2.toString()
19     }
20 }
```

PROGRAM 9-6**PROGRAM OUTPUT**

```
101 Raj London
102 Vijay New York
```

9.12 Scope of Variables

9.12.1 Scope of Local Variables

The scope of a variable is the part of the program where the variable can be referenced. A variable defined inside a method is referred to as a local variable. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared and assigned a value before it can be used.

A parameter is actually a local variable. The scope of a method parameter covers the entire method. A variable declared in the initial-action part of a for-loop header has its scope in the entire loop. However, a variable declared inside a for-loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable, as shown in Figure 9.11.

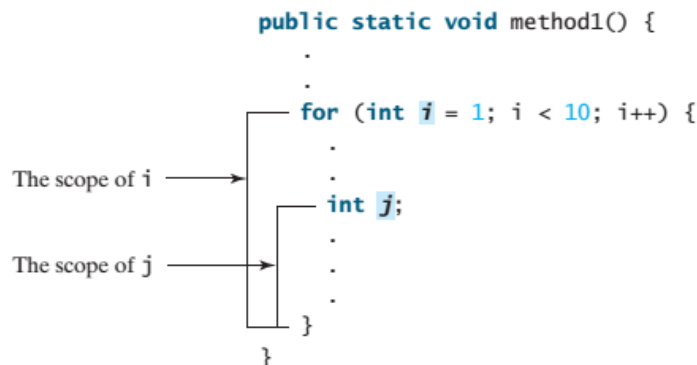


Figure 9.11 A variable declared in the initial action part of a for-loop header has its scope in the entire loop.

You can declare a local variable with the same name in different blocks in a method, but you cannot declare a local variable twice in the same block or in nested blocks, as shown in Figure 9.12.

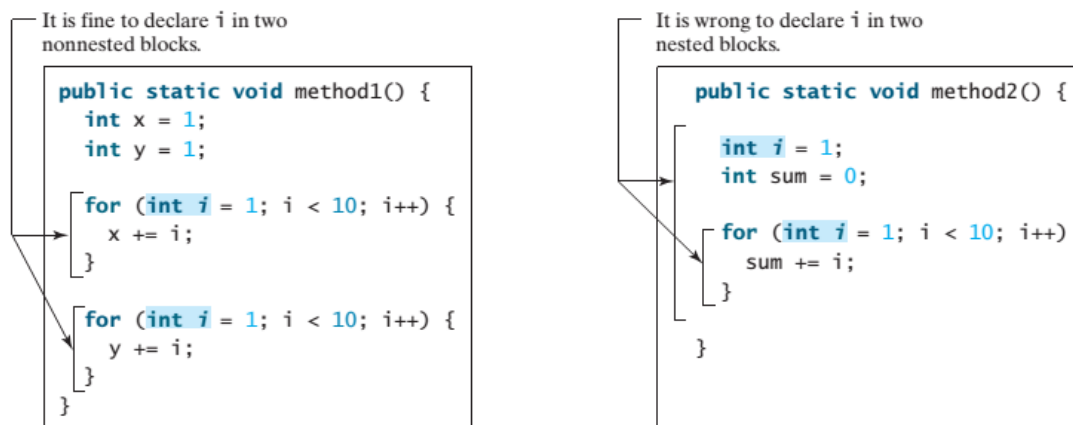


Figure 9.12 A variable can be declared multiple times in non-nested blocks, but only once in nested blocks.

Do not declare a variable inside a block and then attempt to use it outside the block. Here is an example of a common mistake:

```

for (int i = 0; i < 10; i++) {
}
System.out.println(i);

```

The last statement would cause a syntax error, because variable `i` is not defined outside of the for loop.

9.12.2 Scope of Instance and Static Variables

The **scope of instance and static variables is the entire class, regardless of where the variables are declared**. 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 9.13(a). 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 9.13(b).

```
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 9.13 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 non-nesting blocks. 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`. Reason:

- `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.

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

9.13 this Reference

The keyword `this` refers to the object itself. It can also be used inside a constructor to invoke another constructor of the same class. 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 Figure 9.14(a) uses `this` to reference the object's radius and invokes its `getArea()` method explicitly. The `this` reference is normally omitted, as shown in Figure 9.14(b). However, the `this` reference is needed to reference hidden data fields or invoke an overloaded constructor.

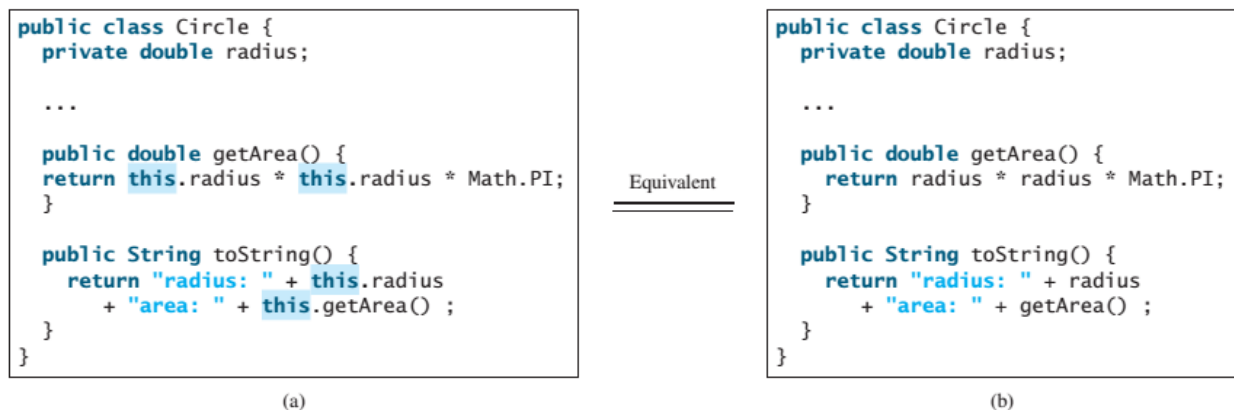


Figure 9.14 Using `this` reference to invoke an objects method and reference data members

9.13.1 Using this to Reference 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 9.15(a).

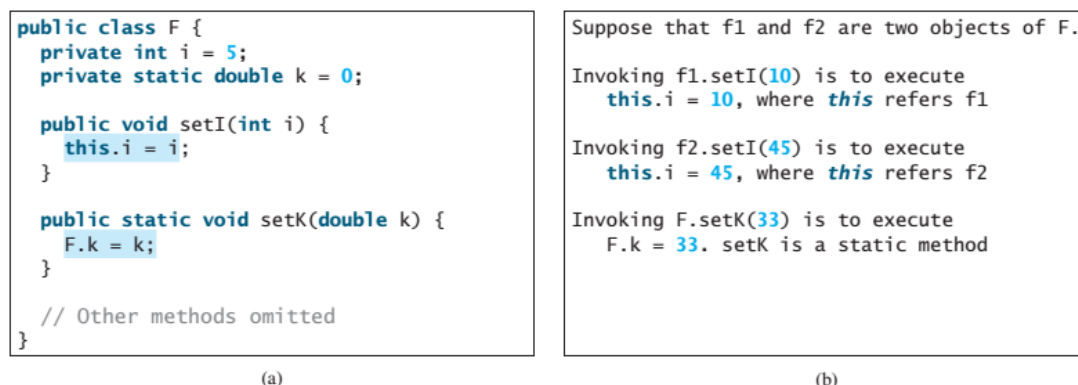


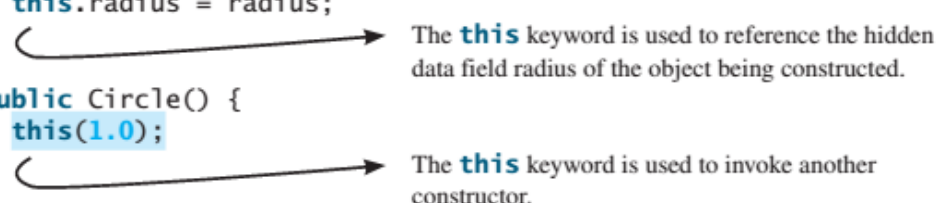
Figure 9.15 The keyword `this` refers to the calling object that invokes the method.

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 `setK()`, as shown in Figure 9.14(b). 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.

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

9.14 Processing Primitive Data Type Values as Objects

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.

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 9.16.

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")`.

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.

java.lang.Integer	java.lang.Double
-value: int <u>+MAX_VALUE: int</u> <u>+MIN_VALUE: int</u>	-value: double <u>+MAX_VALUE: double</u> <u>+MIN_VALUE: double</u>
+Integer(value: int) +Integer(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longValue(): long +floatValue(): float +doubleValue(): double +compareTo(o: Integer): int +toString(): String <u>+valueOf(s: String): Integer</u> <u>+valueOf(s: String, radix: int): Integer</u> <u>+parseInt(s: String): int</u> <u>+parseInt(s: String, radix: int): int</u>	+Double(value: double) +Double(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longValue(): long +floatValue(): float +doubleValue(): double +compareTo(o: Double): int +toString(): String <u>+valueOf(s: String): Double</u> <u>+valueOf(s: String, radix: int): Double</u> <u>+parseDouble(s: String): double</u> <u>+parseDouble(s: String, radix: int): double</u>

Figure 9.16 The wrapper classes provide constructors, constants, and conversion methods for manipulating various data types.

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);
```

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;

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;
```

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");
```

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).

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.
```

9.15 Automatic Conversion between Primitive Types and Wrapper Class Types

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.

For instance, the following statement in Figure 9.17(a) can be simplified as in Figure 9.17(b) due to autoboxing.

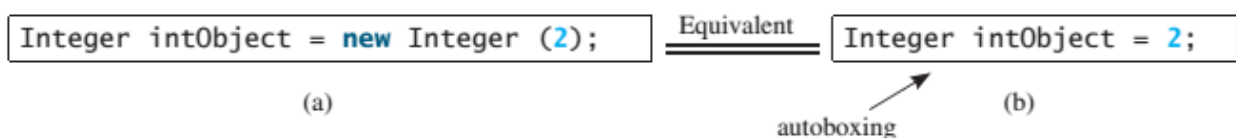


Figure 9.17 Simplification due to autoboxing

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.

9.16 Class Abstraction and Encapsulation

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 let 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 9.17, 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).**

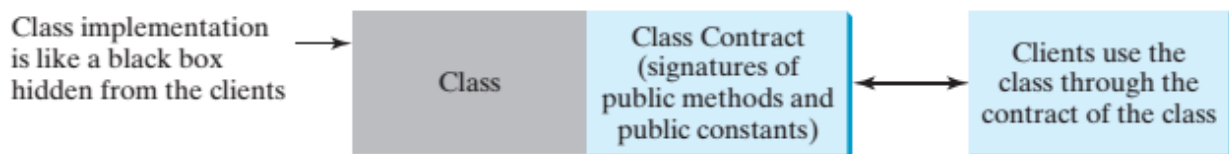


Figure 9.17 Class abstraction separates class implementation from the use of the class.

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 do not 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.

9.17 Class Implementation

Recall in the earlier sections, a class has the following structure.

```
public class ClassName{
    <data members>
    <constructors>
    <methods>
}
```

9.17.1 Data Members

The syntax for declaring data members is:

```
<visibilityModifier> <modifier> <dataType> <dataName>;
```

- <visibilityModifier>: `public`, `private` or `package`
 - Data members should be declared `private` as they are the implementation details of the class and should be invisible to the clients (i.e. other classes).
 - Constants may be declared `public` if they are meant to be used directly by the outside methods. Example: `Math.PI` is a public constant of the `Math` class.
 - Another access mode is **package access** (also known as default access). Package access will be given to your attributes or methods **if you did not state down the visibility modifier explicitly**.
 - Attributes and methods with package access can be accessed by classes belonging to the same package.
 - Note that all classes in the same directory that does not belong to some other package will belong to the **default package**. Thus, if your attribute or method has package access, any class not belonging to any package and in the same directory will have access to it.
 - <modifier>: `static` or nothing (Static data members will be shared by all instances)
- <dataType>: primitive data type (e.g. `int`, `double`) or reference data type (e.g. `String`)
- <dataName>: name of the data member.

9.17.2 Constructor

A constructor is executed when a new instance of the class is created. The syntax for declaring constructor is:

```
visibilityModifier <className>(<parameters>) {
    <statements>
}
```

- **A constructor is called when you create a new object, together with the `new` keyword.** You may not call the constructor in any other ways (like how you do for normal methods).
- **Constructor has the same name as the class but there is no return type.**

- It is possible to have more than one constructor to a class. They are called **overloaded constructors**.

It is good practice to define the constructor; however, it is not a requirement. If no constructor is defined, then the Java compiler will automatically include a default constructor that accepts nothing and does nothing.

9.17.3 Methods

Instance methods

- Operates on an object only (i.e. an instance of the class. Note that the keyword `static` is not used).
- Keyword `void` is used if the method does not return anything.
- An **accessor** (getter methods) is a method that queries the object for some information without changing it.
- The return statement is a special statement that instructs the method to terminate and return an output to the statement that called the method.
- A **mutator** (setter methods) is a method that sets/changes the property of an object.

Class methods (or Static methods)

- A service provided by a class and it is not associated with a particular object. We use the reserved word `static` to define a class method.

Example:

```
public static double getMinCamRes(){ // a class method
    return CAMERA_RES; // a class variable
}
```

- Since a `static` method does not need a calling object (i.e. instance of the class), **a static method cannot use an instance variable or method** (i.e. you may not invoke a non-static method in a static method). However, you may use a static method within another static method.

9.18 Case Studies

9.18.1 Case Study: Building a Handphone Class

- Define a HandPhone class to model handphones. Some properties of a handphone are brand, model, colour, price, weight and talkTime.
- Define two constructors for the Handphone class. One constructor instantiates data members with default values and another allow object to be instantiated with customize values.

```
public class HandPhone{

    <data members> // CAMERA_RES is a class constant
    public static final double CAMERA_RES = 5.0;
    private double talkTime;
    private int weight;
    private String brand;
    private String model;
    private double price;

    <constructors>
    public Handphone(){ //constructor to give default values
        brand = "Samsung";
        model = "S7";
        price = 599;
        talkTime = 120;
        weight = 145;
        // alternatively you can invoke the overloaded
        // constructor. Note: this need to be the first line in
        // the constructor
        // this("Samsung","S7", 599, 120, 145);
    }

    // overloaded constructor to customize values
    public Handphone(String b, String m, double p, double tt, int
    wt){

        brand = b;

        model = m;

        price = p;

        talkTime = tt;

        weight = wt;

    }
}
```

Continue next page →

- Define accessors and mutators for the Handphone class.

```

<methods> // ACCESSORS
public double getTalktime(){
    return this.talkTime;
}
public int getWeight(){
    return this.weight;
}
public String getBrand(){
    return this.brand;
}
public String getModel(){
    return this.model;
}
public double getPrice(){
    return this.price;
}
// MUTATORS
public void setTalktime(double tt){
    this.talkTime = tt;
}
public void setWeight(int wt){
    this.weight = wt;
}
public void setBrand(String b){
    this.brand = b;
}

public void setModel(String m){
    this.model = m;
}
public void setPrice(double p){
    this.price = p;
}
public String toString(){
    return this.brand + " " + this.model + "\nTalk time: " +
    this.talkTime + "\nWeight: " + this.weight + "\nPrice: $" +
    this.price;
}
}

```

```

public class testHandphone {
    public static void main(String[] args) {
        Handphone h = new Handphone();
        System.out.println(h);
        System.out.println("Camera Resolution: " +
            Handphone.CAMERA_RES);
    }
}

```

PROGRAM OUTPUT

```

Samsung S7
Talk time: 120.0
Weight: 145
Price: $599.0
Camera Resolution: 5.0

```

Note: CAMERA_RES can be accessed directly without the need to create an instance object of Handphone because it is defined as static, which means its scope is at class level rather than instance level.

9.18.2 Case Study: Account Class

- Define an Account class to model bank accounts. Some properties of an Account are name, account number, balance, type and interest rate for all account type. Include a static variable to count number of objects created. For simplicity of illustration, account number is the number objects being created.
- Define two constructors for the Account class. One constructor instantiates data members with default values and another allow object to be instantiated with customize values.

```
public class Account{
    <data members>
    //create the class variable to keep track of how many accounts
    public static int numberOfObjects = 0;
    private String name;
    private int accNumber;
    private double balance;
    private String type;
    private double rate;

    public Account(){ //constructor to give default values
        this("Dummy",100.0, "Savings" , 1.5);
    }
    // overloaded constructor to customize values
    public Account(String s, double b, String t, double r){
        numberOfObjects++; // increment static variable count
        this.accNumber = numberOfObjects; // assign acc number based on
        this.name = s; // static count value
        this.balance = b;
        this.type = t;
        this.rate = r;
    }

    <methods>
    // this is a Class method because of the static modifier
    public static int getNumberOfObjects(){
        return numberOfObjects;
    }

    // ACCESSORS
    public String getName(){
        return this.name;
    }

    public int getAccNumber(){
        return this.accNumber;
    }

    public double getBalance(){
        return this.balance;
    }

    public String getType(){
        return this.type;
    }

    public double getRate(){
        return this.rate;
    }
}
```

```

// ACCESSORS
public void transferTo(double amount, Account otherAccount){
    withdraw(amount); //it is possible to call method of same class
    otherAccount.deposit(amount); // deposit to otherAccount object
}

public void deposit(double amount ){
    this.balance += amount;
}

public void withdraw(double amount ){
    this.balance -= amount;
}

public void setAccNumber(int n){
    this.accNumber = n;
}

public void setName(String n){
    this.name = n;
}

public void setBalance(double b){
    this.balance = b;
}

public void setType(String t){
    this.type = t;
}

public void setRate(double r){
    this.rate = r;
}
}

```

```

public class BankAccountTester{
    public static void main(String[] args) {
        Account donaldAccount = new Account("Donald",1000, "Saving", 1.25);
        Account hilaryAccount = new Account("Hilary", 500, "Saving", 1.25);
        System.out.println("Before Transfer");
        System.out.println(donaldAccount);
        System.out.println(hilaryAccount);
        //transfer $50 from Donald account to Hilary account
        donaldAccount.transferTo(50.0, hilaryAccount);
        System.out.println("\nAfter Transfer");
        System.out.println(donaldAccount.toString());
        System.out.println(hilaryAccount.toString());
    }
}

```

PROGRAM OUTPUT

Before Transfer

Customer: Donald Account No: 1 Type: Saving Rate: 1.25 Balance: 1000.0
 Customer: Hilary Account No: 2 Type: Saving Rate: 1.25 Balance: 500.0

After Transfer

Customer: Donald Account No: 1 Type: Saving Rate: 1.25 Balance: 950.0
 Customer: Hilary Account No: 2 Type: Saving Rate: 1.25 Balance: 550.0

9.18.3 Case Study: Car Class

- Define a Car class to model cars. Some properties of a car are brand, model, number plate, speedometer, gear and engine.

```
public class Car {
    private String brand;
    private String model;
    private String numberPlate;
    private int speedometer;
    private Engine engine;
    private Gear gear;
    public Car(String b, String m, String n){
        this.brand = b;
        this.model = m;
        this.numberPlate = n;
        engine = new Engine();
        gear = new Gear();
    }
    public String getbrand(){
        return this.brand;
    }
    public String getModel(){
        return this.model;
    }
    public String getNumberPlate(){
        return this.numberPlate;
    }
    public void setBrand(String s){
        this.brand = s;
    }
    public void setModel(String m){
        this.model = m;
    }
    public void startEngine(){
        this.engine.start();
    }
    public void stopEngine(){
        this.engine.stop();
    }
    public void pushGear(char mode){
        this.gear.changeGear(mode);
    }
    public void pressAccelerator(){
        this.gear.accelerate();
    }
    public void pressBrake(){
        this.gear.brake();
    }
    public char getGearMode(){
        return this.gear.getMode();
    }
    public int showSpeedometer(){
        this.speedometer = this.gear.getSpeed();
        return this.speedometer;
    }
    public String toString(){
        return this.brand + " " + this.model + " " +
            this.numberPlate + " travelling at " + this.speedometer +
            " Km/hour";
    }
}
```

```
public class Gear {
    private int speed;
    private char mode;
    public Gear(){
        this('P');
    }
    public Gear(char m){
        this.mode = m;
    }
    public void changeGear(char m){
        switch(m){
            case 'P': this.speed = 0; break; // park
            case 'D': this.speed = 20; break; // drive
            case 'N': this.speed = 0; break; // neutral
            case 'R': this.speed = 5; break; // reverse
            case 'S': this.speed = 40; break; // sport
        }
        this.mode = m;
    }

    public int getSpeed(){
        return this.speed;
    }
    public char getMode(){
        return this.mode;
    }

    public void accelerate(){
        if(this.mode == 'N' || this.mode == 'P')
            System.out.println("Cannot move. Gear not Engaged.");
        else if ((this.mode == 'D' || this.mode == 'S') && speed <= 200)
            this.speed += 5;
        else
            this.speed += 3;
    }
    public void brake(){
        if (speed > 0)
            this.speed -= 2;
    }
}
```

```
public class Engine {
    private boolean status;
    public Engine(){
        this.status = false;
    }
    public void start(){
        this.status = true;
        System.out.println("Engine On");
    }
    public void stop(){
        this.status = false;
        System.out.println("Engine Off");
    }
    public boolean getStatus(){
        return this.status;
    }
}
```



```
public class testCar {

    public static void main(String[] args) {
        Car taxi = new Car("Mercedes", "E200", "SLK999T");
        taxi.startEngine();

        System.out.println("Gear Mode: " + taxi.getGearMode());
        System.out.println(taxi);
        taxi.pressAccelerator();
        taxi.pushGear('D');
        System.out.println("Gear Mode: " + taxi.getGearMode());

        for(int i=0; i<3; i++){
            taxi.pressAccelerator();
            System.out.println("Speedometer: " + taxi.showSpeedometer());
        }
        System.out.println(taxi);

        for(int i=0; i<3; i++){
            taxi.pressBrake();
            System.out.println("Speedometer: " + taxi.showSpeedometer());
        }
        System.out.println(taxi);
        taxi.stopEngine();
    }
}
```

PROGRAM OUTPUT

```
Engine On
Gear Mode: P
Mercedes E200 SLK999T travelling at 0 Km/hour
Cannot move. Gear not Engaged.
Gear Mode: D
Speedometer: 25
Speedometer: 30
Speedometer: 35
Mercedes E200 SLK999T travelling at 35 Km/hour
Speedometer: 33
Speedometer: 31
Speedometer: 29
Mercedes E200 SLK999T travelling at 29 Km/hour
Engine Off
```

[Reference]

- [1] Introduction to Java Programming Comprehensive Version 10th Ed, Daniel Liang, 2016.
- [2] Java How To Program 10th Ed, Paul Deitel, Harvey Deitel, 2016