JAVA PROGRAMMING

Chapter 3

Classes, Methods, and Interfaces

Objectives

- Develop code that declares classes, interfaces, and enums, and includes the appropriate use of package and import statements.
- Develop code that declares an interface. Develop code that implements or extends one or more interfaces. Develop code that declares an abstract class. Develop code that extends an abstract class.

Objectives (cont.)

- Develop code that declares both static and non-static methods. Develop code that declares and uses a variable-length argument list.
- Develop constructors for one or more of the classes. Given a class declaration, determine if a default constructor will be created, and if so, determine the behavior of that constructor. Given a nested or non-nested class listing, write code to instantiate the class.

Classes, Methods, and Interfaces

- **Using Methods**
- Working with Classes and Objects
- Understanding Enums
- Inheritance
- Writing and Invoking Constructors
- Writing and Using Interfaces

Using Methods

- Methods represent operations on data and also hold the logic to determine those operations
- Using methods offer two main advantages:
  - A method may be executed (called) repeatedly from different points in the program: decrease the program size, the effort to maintain the code and the probability for an error
  - Methods help make the program logically segmented, or modularized: less error prone, and easier to maintain

Defining a Method

- A method is a self-contained block of code that performs specific operations on the data by using some logic
- *Method declaration*:
  - Name
  - Parameter(s)
  - Argument(s)
  - Return type
  - Access modifier

Defining a Method

- The syntax for writing a method in Java:

```
<modifier>  <returnType>  <methodName> ( <Parameters>) {
// body of the method. The code statements go here.
}
```

```
public int square (int number) {
    return number*number;
}
```

```
int myNumber = square(2);
```

The Static Methods and Variables

- The static methods and variables are shared by all the instances of a class
- The `static` modifier may be applied to a variable, a method, and a block of code inside a method
- Because a static element of a class is visible to all the instances of the class, if one instance makes a change to it, all the instances see that change.

Listing 3-1. *RunStaticExample.java*

```
1. class StaticExample {
2.    static int staticCounter=0;
3.    int  counter=0;
4.    StaticExample() {
5.          staticCounter++;
6.          counter++;
7.    }
8. }
9. class RunStaticExample {
10.    public static void main(String[] args) {
11.        StaticExample se1 = new StaticExample();
12.        StaticExample se2 = new StaticExample();
13.        System.out.println("Value of staticCounter for se1: " +
           se1.staticCounter);
14.           System.out.println("Value of staticCounter for se2: " +
                   se2.staticCounter);
15.          System.out.println("Value of counter for se1: " + se1.counter);
16.          System.out.println("Value of counter for se2: " + se2.counter);
17.          StaticExample.staticCounter = 100;
18.           System.out.println("Value of staticCounter for se1: " +
                     se1.staticCounter);
19.           System.out.println("Value of staticCounter for se2: " +
                      se2.staticCounter);
20.    }
21. }
```

The Static Methods and Variables

- A static variable is initialized when a class is loaded, whereas an instance variable is initialized when an instance of the class is created
- A static method also belongs to the class. It can be called even before a single instance of the class exists
- A static method can only access the static members of the class

The Static Code Block

- A class can also have a static code block outside of any method
- The code block does not belong to any method, but only to the class
- executed before the class is instantiated, or even before the method main() is called

Listing 3-2. *RunStaticCodeExample.java*

```
1.  class StaticCodeExample {
2.  static int counter=0;
3.  static {
4.       counter++;
5.      System.out.println("Static Code block: counter: " + counter);
6.  }
7.  StaticCodeExample() {
8.          System.out.println("Construtor:  counter: " + counter);
9.  }
10.}
11.  public class  RunStaticCodeExample {
12.  public static void main(String[] args) {
13.        StaticCodeExample sce = new StaticCodeExample();
14.        System.out.println("main: counter:" + sce.counter);
15.  }
16.}
```

Methods with a Variable Number of Parameters

The rules to define variable-length parameters:

- There must be only one variable-length parameters list.
- If there are individual parameters in addition to the list, the variable-length parameters list must appear last inside the parentheses of the method.
- The variable-length parameters list consists of a type followed by three dots and the name.

Listing 3-3. *VarargTest.java*

```
1. import java.io.*;
2. class MyClass {
3.     public void printStuff(String greet, int... values) {
4.         for (int v : values ) {
5.             System.out.println( greet + ":" + v);
6.         }
7.     }
8.  }
9.   class VarargTest {
10.        public static void main(String[] args) {
11.          MyClass mc = new MyClass();
12.          mc.printStuff("Hello", 1);
13.          mc.printStuff("Hey", 1,2);
14.          mc.printStuff("Hey you", 1,2,3);
15.        }
16.    }
```

JavaBeans Naming Standard for Methods

- A JavaBean is a special kind of Java class that is defined by following certain rules:
  - The private variables / properties can only be accessed through getter and setter methods
  - The getter and setter methods must be public so that anyone who uses the bean can invoke them.
  - A setter method must have the void return type and must have a parameter that represents the type of the corresponding property
  - A getter method does not have any parameter and its return type matches the argument type of the corresponding setter method.

JavaBeans Naming Standard for Methods

```
public class ScoreBean {
 private double meanScore;
 // getter method for property meanScore
  public double getMeanScore() {
     return meanScore;
  }
// setter method to set the value of the property meanScore
  public void setMeanScore(double score) {
     meanScore = score;
  }
}
```

- `getMeanScore()` and `setMeanScore()`, correspond to the variable (property) `meanScore`

Classes, Methods, and Interfaces

- Using Methods
- **Working with Classes and Objects**
- Understanding Enums
- Inheritance
- Writing and Invoking Constructors
- Writing and Using Interfaces

Working with Classes and Objects

- A class is a template that contains the data variables and the methods that operate on those data variables following some logic

- Class members:
  - Variables represent the state of an object
  - Methods constitute its behavior

Defining Classes

- The general syntax:
  `<modifier> class <className> { }`
- `<className>` specifies the name of the class
- `class` is the keyword
- `<modifier>` specifies some characteristics of the class:
  - *Access modifiers*: `private`, `protected`, *default* and `public`
  - *Other modifiers*: `abstract`, `final`, and `strictfp`

Defining Classes - Example

Listing 3-4. *ClassRoom.java*

```
1. class ClassRoom {
2 . private String roomNumber;
3.  private int totalSeats = 60;
4.  private static int totalRooms = 0;

5.   void setRoomNumber(String rn) {
6.     roomNumber = rn;
7.   }
8.   String getRoomNumber() {
9.     return roomNumber;
10.  }
11.  void setTotalSeats(int seats) {
12.      totalSeats = seats;
13.  }
14.  int getTotalSeats() {
15.     return totalSeats;
16.  }
17. }
```

Creating Objects

```
<className>  <variableName> = new <classConstructor>
```

- `<variableName>`: the name of the object reference that will refer to the object that you want to create
- `<className>`: the name of an existing class
- `<classConstructor>`: a constructor of the class
- The right side of the equation creates the object of the class specified by `<className>` with the `new` operator, and assigns it to `<variableName>` (i.e. `<variableName>` points to it)

Creating Objects - Example

```
class ClassRoomManager {
   public static void main(String[] args)
   {
      ClassRoom  roomOne  =  new ClassRoom();
      roomOne.setRoomNumber("MH227");
      roomOne.setTotalSeats(30);

      System.out.println("Room number: " + roomOne.getRoomNumber());
      System.out.println("Total seats: " + roomOne.getTotalSeats());
   }
}
```

Nested Classes

- allows you to define a class (like a variable or a method) inside a top-level class (outer class or enclosing class)

```
class <OuterClassName> {
  // variables and methods for the outer class
  …
     class  <NestedClassName> {
        // variables and methods for the nested class
        …
      }
}
```

Nested Classes

- an instance of an inner class can only exist within an instance of its outer class

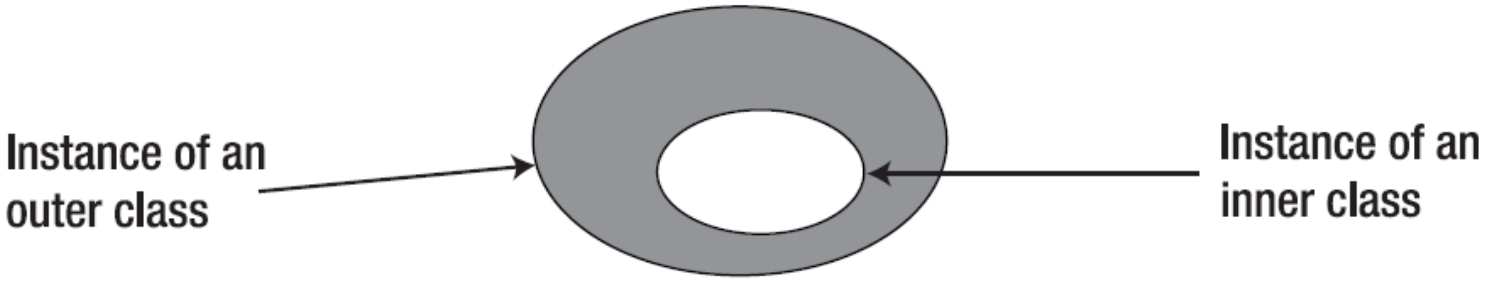

Figure 3-1. *The instance of an inner class has direct access to the instance variables and methods of an instance of the outer class.*

Listing 3-5. *TestNested.java*

```
1. class TestNested {
2.  public static void main(String[] args) {
3.    String ext = "From external class";
4.    MyTopLevel mt = new MyTopLevel();
5.    mt.createNested();
6.    MyTopLevel.MyInner inner = mt.new MyInner();
7.    inner.accessInner(ext);
8.  }
9. }
10.  class MyTopLevel{
11.    private String top = "From Top level class";
12.     MyInner minn = new MyInner();
13.     public void createNested() {
14.          minn.accessInner(top);
15.     }
16.     class MyInner {
17.         public void accessInner(String st) {
18.            System.out.println(st);
19.         }
20.      }
21.  }
```

Classes, Methods, and Interfaces

- Using Methods
- Working with Classes and Objects
- **Understanding Enums**
- Inheritance
- Writing and Invoking Constructors
- Writing and Using Interfaces

Understanding Enums

- useful when you want a variable to hold only a predetermined set of values
- define an enum variable in two steps:
  1. Define the enum type with a set of named values
  2. Define a variable to hold one of those values

```
enum AllowedCreditCard {VISA, MASTER_CARD, AMERICAN_EXPRESS};

AllowedCreditCard visa = AllowedCreditCard.VISA;
```

```
System.out.println("The allowed credit card value: " + visa)
```

Methods of the Enum Class

- subclass of the Java class Enum

Table 3-1. *Some Methods of the Enum Class*

Method
`final boolean equals(Object obj)`
`final String name()`
`String toString()`
`static Enum valueOf(Class enumClass, String name)`

Listing 3-8. *EnumColorTest.java*

```
1. public class EnumColorTest {
2.   public static void main(String[] args) {
3.       Colors c = new Colors();
4.       c.color = Colors.ThreeColors.RED;
5.       System.out.println(c.color);
6.    }
7. }
8. class Colors {
9.     enum ThreeColors {BLUE, RED, GREEN}
10.    ThreeColors  color;
11.}
```

The output of this code follows:

```
RED
```

Classes, Methods, and Interfaces

- Using Methods
- Working with Classes and Objects
- Understanding Enums
- **Inheritance**
- Writing and Invoking Constructors
- Writing and Using Interfaces

Inheritance

- enable the programmer to write a class based on an already existing class - the parent class / super class
- The subclass inherits (reuses) the non-private members of the super class
- The keyword to derive a class from another class is `extends`

```
class ComputerLab extends ClassRoom {
 }
```

Listing 3-10. *TestComputerLab.java*

```
1. class TestComputerLab {
2.  public static void main(String[] args) {
3.    ComputerLab cslab = new ComputerLab();
4.    cslab.printSeatInfo();
5.    System.out.println("Total seats in the class room:
          "+cslab.getTotalSeats());
6.    }
7.  }

8. // Class ComputerLab
9.   class ComputerLab extends ClassRoom {
10.    int  totalComputers = 30;
11.    String labAssistant="TBA";
12.   void printSeatInfo() {
13.       System.out.println("There are " + getTotalSeats() + " seats, and "+
           totalComputers + "  computers in this computer lab.");
15.   }
16.   String getLabAssistant(){
17.      return labAssistant;
18.   }
19.   void setLabAssistant(String assistant){
20.      this.labAssistant = assistant;
21.   }
22.  }
```

Classes, Methods, and Interfaces

- Using Methods
- Working with Classes and Objects
- Understanding Enums
- Inheritance
- **Writing and Invoking Constructors**
- Writing and Using Interfaces

Writing and Invoking Constructors

- The constructor of a class has the same name as the class and has no explicit return type
- The new operator allocates memory for the instance, and executes the constructor to initialize the memory

```
ComputerLab csLab = new ComputerLab();
```

1. Allocates memory for an instance of class ComputerLab
2. Initializes the instance variables of class ComputerLab
3. Executes the constructor ComputerLab()

Caution

- If you do not provide any constructor for a class you write, the compiler provides the default constructor for that class
- You can also define non default constructors with parameters
- The constructor may be called:
  - from inside the class: from within another constructor, using `this()` or `super()`
  - from outside the class: with the `new` operator

`this()` and `super()`

- use `this()` to call another constructor in the same class, and use `super()` to call a constructor in the superclass
- must appear in the beginning the constructor
- Without a super call or a this call, the compiler places a super call in the beginning of the constructor's body
- Before executing the body of a constructor in a class that is being instantiated, a constructor in the superclass must be executed

Listing 3-11. *TestConstructors.java*

```
1.   class TestConstructors {
2.       public static void main(String[] args) {
3.            new MySubSubClass();
4.        }
5.
6.     }
7. // Class MySuperClass
8.    class MySuperClass {
9.         int superVar = 10;
10.        MySuperClass(){
11.        System.out.println("superVar: " + superVar);
12.        }
13.        MySuperClass(String message) {
14.          System.out.println(message + ": " + superVar);
15.        }
16.    }
17.  // Class MySubClass inherits from MySuperClass
18.    class MySubClass extends MySuperClass {
19.          int subVar = 20;
20.          MySubClass() {
21.            super("non default super called");
22.            System.out.println("subVar: " + subVar);
23.          }
24.     }
```

```
25.  // Class MySubSubClass inherits from MySubClass
26.     class MySubSubClass extends MySubClass {
27.         int subSubVar = 30;
28.         MySubSubClass() {
29.           this("A non-deafult constructor of MySubSubClass");
30.           System.out.println("subSubVar: " + subSubVar);
31.          }
32.          MySubSubClass(String message){
33.             System.out.println(message);
34.          }
35.     }
```

Example - Compiler Error

```
class A {
    int myNumber;
  A(int i) {
    myNumber =  i;
  }
}
class B extends A {
   String myName;
   B (String name) {
    myName = name;
  }
}
```

Key Points

- A constructor of a class has the same name as the class, and has no explicit return type
- A class may have more than one constructor
- If the programmer defines no constructor in a class, the compiler will add the default constructor with no arguments
- If there are one or more constructors defined in the class, the compiler will not provide any constructor

Key Points (cont.)

- A constructor may have zero or more parameters
- From outside the class, a constructor is always called with the `new` operator
- From inside the class, it may be called from another constructor with the `this` or `super` operator
- Unlike other methods, the constructors are not inherited
- If there is no `super` call in a constructor, the default `super` call is placed by the compiler

Classes, Methods, and Interfaces

- Using Methods
- Working with Classes and Objects
- Understanding Enums
- Inheritance
- Writing and Invoking Constructors
- **Writing and Using Interfaces**

Writing and Using Interfaces

- Java supports single inheritance - a subclass in Java can have only one superclass
- If multiple inheritance is needed: use an interface
- An interface is a template that contains some method declarations - no implementation
- The class that inherits from an interface must provide the implementation for the methods declared in the interface

Interface Declaration

- You define an interface by using the keyword `interface`
- The methods declared in an interface are implicitly `public` and `abstract`
- The data variables declared in an interface are inherently constants and are visible from all the instances of the class that implements the interface

```
interface <InterfaceName> {
 <dataType1> <var1>;
 <dataType2> <var2>;
 <ReturnType1> <methodName1> ( );
 <ReturnType2> <methodName2>(<parameters>);
} // interface definition ends here.
```

More Features

- A class can extend only one class by using the keyword `extends`, but it can inherit from one or more interfaces by using the keyword `implements`
- An interface can also extend one or more interfaces by using the keyword `extends`
- An interface cannot implement any interface or class

Listing 3-12. *TestInterface.java*

```
1.  interface ParentOne {
2.    int pOne = 1;
3.    void printParentOne();
4.  }
5.  interface ParentTwo {
6.    int pTwo = 2;
7.    void printParentTwo();
8.  }
9.  interface Child extends ParentOne, ParentTwo{
10.         int child = 3;
11.         void printChild();
12. }
13.
```

```
14. class InheritClass implements Child {
15.     public void printParentOne(){
16.         System.out.println(pOne);
17.     }
18.     public void printParentTwo(){
19.         System.out.println(pTwo);
20.     }
21.     public void printChild(){
22.         System.out.println(child);
23.     }
24.  }
25.  class TestInterface {
26.     public static void main(String[] args){
27.       InheritClass ic = new InheritClass();
28.       ic.printParentOne();
29.       ic.printParentTwo();
30.       ic.printChild();
31.     }
32.  }
```

Important Points about Interfaces

- All interface variables are inherently `public`, `static`, and `final`
- All interface methods are inherently `public` and `abstract`
- Because interface methods are inherently abstract, they cannot be declared `final`, `native`, `strictfp`, or `synchronized`
- A class can extend another class and implement one or more interfaces at the same time
- An interface cannot extend a class
- An interface cannot implement another interface or class

Legal and Illegal Cases

Table 3-2. *Examples of Legal and Illegal Use of extends and implements for Classes C1 and C2 and Interfaces I1 and I2*

Example	Legal?	Reason
`class C1 extends C2 { }`	Yes	A class can extend another class.
`interface I1 extends I2 { }`	Yes	An interface can extend another interface.
`class C1 implements I1 { }`	Yes	A class can implement an interface.
`class C1 implements I1, I2 { }`	Yes	A class can implement multiple interfaces.
`class C1 extends I1 { }`	No	A class cannot extend an interface; it has to implement it.
`class C1 extends C2, C3 { }`	No	A class cannot extend multiple classes.
`interface I1 implements I2 { }`	No	An interface cannot implement another interface.
`interface I1 extends I2 { }`	Yes	An interface can extend an interface.

Example	Legal?	Reason
`interface I1 extends I2, I3 { }`	Yes	An interface can extend multiple interfaces.
`interface I1 extends C1 { }`	No	An interface cannot extend a class.
`interface I1 implements C1 { }`	No	An interface cannot implement a class.
`class C1 extends C2 implements I1 { }`	Yes	A class can extend one other class and implement one or more interfaces.
`class C1 implements I1 extends C2 { }`	No	`extends` must come before `implements`.