4

Java as an OOP Language

Java is an object oriented programming language. A Java program consists of a set of objects that interact and communicate with each other. Classes are the blueprints or construction plans for these objects and specify the properties/state and behaviour of objects. This chapter deals with concepts in object oriented programming in the context of Java.

4.1 Defining Classes

Classes are the fundamental building blocks of any object-oriented language.

A class describes the data and behaviour associated with instances of that class. When a class is instantiated, an object is created: this object has properties and behaviour similar to other instances of the same class. The data associated with a class or object is stored in variables. The behaviour associated with a class or object is implemented by means of methods. Methods are similar to the functions or procedures of procedural languages such as C or Pascal.

A class can be defined as follows:

```
class First
{
    Body of the class
}
```

The keyword class is used to define a class. The keyword is followed by the name of the class (in the above example First). The body of the class is contained within curly brackets. The body consists of statements related to constructors, methods and variables.

Every class defined in Java is a child of the Object class.

4.1.1 Creating instance and class variables

A class usually contains variables and methods. Certain specific rules have to be followed for defining variables and methods within a class. This section deals with the definition of different types of variable, namely instance variables, constants and class variables.

4.1.1.1 Instance variables

Instance variables are declared and defined in almost the same way as local variables, the main difference being their location in the class definition. Variables are considered instance variables if they are declared outside a method definition. It is customary to define instance variables just after the first line of the class definition.

4.1.1.2 Constants

A constant variable or constant is a variable whose value never changes.

Constants are used to define shared values for all the methods of an object so that object-wide values that will never change can be given meaningful names. In Java, only instance and class variables can be constants (not local variables). To declare a constant, the keyword final is used. It should be placed before declaration of the variable and should include an initial value for that variable, as shown in the following examples:

```
final float pi = 3.14;
final boolean debug = false;
final int maxsize = 40000;
```

Constants can be useful for naming various states of an object and testing them. For a test label that can be aligned left, right or centre, the values can be defined as constant integers.

```
final int LEFT = 0;
final int RIGHT = 1;
final int CENTER = 2;
```

4.1.1.3 Class variables

Class variables are global to a class and to all instances of that class. Class variables are used for communication between different objects within the same class. These variables are also used for tracking global states among a set of objects. The static keyword is used in the class declaration to declare a class variable. Some examples of this are given below:

```
static int sum;
static final int maxObjects = 10;
```

4.1.2 Defining methods

In Java, method definition usually consists of four fundamental parts, the name of the method, the object type or the data type that the method returns (referred to as return type), the list of parameters and the body of the method.

The first three constituents of the method definition are referred to as *method declaration* or *method signature*. The method declaration usually gives the important information about the method itself.

In other programming languages, method names (function, subroutine or procedure names) are unique; however, in Java, different methods can have the same name, but a different return type or argument list.

> The process of having methods with the same name but with different return type is referred to as *method overloading*.

In Java, a method is defined as follows:

Parameter list

```
returntype methodname (type1 arg1, type2 arg2, type3 arg3...)
{
    body of the method
}
```

In the above illustration, *returntype* refers to the type of value the method returns. It can be a primitive data type, class name or void (if the method does not return a value at all). If the method returns an array object, the array brackets are placed after the return type or after the parameter list.

The two ways of returning array type in a method are depicted below:

Type I For example, the method signature

```
int[ ] method1(int a1, int a2)
{
    //body of the method
}
```

can also be written as follows:

```
int method1(int a1, int a2)[ ]
{
    //body of the method
}
```

Out of these two representations, the first one is most commonly used.

The parameter list in the method definition is a set of variable declarations, separated by commas, inside parentheses (). These parameters become local variables in the body of the method, whose values are the objects or primitives passed in when the method is called.

The body of the method can consist of statements, expressions, method calls to other objects, conditionals, loops and so on.

If return type is real (that is, it is not void), a value should be *returned* explicitly in the body of method. Java provides the keyword return by means of which this can be done.

Program 4.1 shows an example of a class that defines the Calculate method. This method calculates the total marks of five subjects stored in the array marks.

Program 4.1 Defining a method within a class.

```
public class SumOfNum
{
    static int Calculate()
    {
        int total = 0;
        int marks[ ] = {80,65,73,92,67};
        for (int i=0; i<5; i++)
            total += marks[i];
        return total
    }
    public static void main(String args[ ])
    {
        int Sum = Calculate();
        System.out.println("Total marks = :" +Sum);
    }
}
```

The output of the SumOfNum program is as follows:

```
Total marks = : 377
```

4.1.3 Knowing *this*

Programmers sometimes refer to the current object (that is, the object in which the method is contained in the first place) within the body of a method. By referring to the current object, programmers would be able to access the instance variables of that object. In addition, the reference would allow them to pass the current object as an argument to another method. Java provides the keyword this to refer to the current object. The keyword this can be used anywhere the current object might appear. It can be used in dot notation to refer to the object's instance variables, as an argument to a method, as the return value for the current method and so on. This is illustrated in the following examples:

```
t = this.x;              // the x instance variable for this object
this.myMethod(this);     // calls the myMethod method, defined in this class and passes
                            it to the current object
return this;             // returns the current object
```

Program 4.2 illustrates the use of the keyword this. In the program, the main() method creates an object ob of class CircleEx. In the method CircleEx, this.x refers to the first passed value, that is, 10. Similarly this.y and this.radius refers to the second and third values passed to the method CircleEx, respectively (20 and 10). The method display in this program displays the values of x, y and radius when this.display() is invoked from the method CircleEx.

Program 4.2 Illustrating the use of the keyword this.

```
public class CircleEx
{
    int x,y,radius;
    public CircleEx(int x, int y, int radius)
    {
        this.x = x;
        this.y = y;
        this.radius = radius;
        this.display();
    }
    void display()
    {
        System.out.println("value of x is" +x);
        System.out.println("value of x is" +y);
        System.out.println("value of x is" +radius);
    }
    public static void main(String ar[ ])
    {
        System.out.println("Use of keyword this");
        CircleEx ob = new CircleEx(10, 20, 10);
    }
}
```

The output of Program 4.2 is as shown below:

```
Use of keyword this
value of x is 10
value of x is 20
value of x is 10
```

In most programs, this can be omitted entirely. Programmers can refer to both instance variables and method calls defined in the current class simply by invoking their name, this is implicit in those references. This is shown below:

```
t = x          // the x instance variable for this object
myMethod(this) // call the myMethod method, defined in this class
```

The keyword this is a reference to the current instance of a class. It should be used only within the body of an instance method definition. Class methods (methods declared with the keyword static) cannot use this.

4.1.4 Variable scope and method definitions

The scope of a variable specifies the region of the source program where that variable is known, accessible and can be used. In Java, the declared variable has a definite scope. When a variable is defined within a class, its scope determines whether it can be used only within the defined class or outside of the class also.

Local variables can be used only within the block in which they are defined. The scope of instance variables covers the entire class, so they can be used by any of the methods within that class.

Variables must be declared before usage within a scope. Variables that are declared in an outer scope can also be used in an inner scope. The unique way in which Java checks for the scope of a given variable makes it possible to create a variable in an inner scope such that a definition of that variable hides its original value. Program 4.3 makes this point clear.

Program 4.3 Checking the scope of a variable.

```
class ScopeOfVariables
{
    static int var = 10;
    ScopeOfVariables()
    {
        //variable var from outer scope is accessed here
        System.out.println("Variable accessed from outer scope =" +var);
    }
    public static void main(String args[ ])
    {
        int var 25;
                    //variable var from current scope is accessed here
        ScopeOfVariables sc= new ScopeOfVariables();
```

```
        System.out.println("Variable accessed from current scope =" +var);
    }
}
```

The output of Program 4.3 is given below:

```
Variable accessed from outer scope = 10
Variable accessed from current scope = 25
```

The ScopeOfVariables class has two variables defined with the same name (var). One variable (local variable) is defined within the main method, which is initialized to 25, and another variable (instance variable) is declared within the class, which is initialized to 10. When an instance of the class ScopeOfVariables is created, its constructor is invoked. As variable var is accessed inside the constructor, it will check for declaration of the variable in the current scope (that is, within braces). As the declaration is not found in its current scope, it goes to upper level and checks the variable declaration. Since it is defined in the class (outside the method), now, the variable takes the value 10 and the program displays the first line of the output. On the other hand, in the main method, the local variable var is accessed in its current scope. Here, the local variable hides the instance variable. So, this variable takes the value 25 and the println() method will print the second line of the output.

In order to avoid confusion it is better to give different names to local variables and instance variables.

Another way to get around this particular problem is to use the keyword this. The instance variable can be referred to by using this.var in the above program. By referring explicitly to the instance variable using its object, scope conflict can be avoided.

4.1.4.1 Other definitions that can cause bugs

Another situation in variable naming is when a variable that occurs in a super-class is redefined in the sub-class. This can create subtle bugs in the program. This would lead methods that were intended to change the value of an instance variable to change the wrong ones.

Another bug might occur when an object is cast from one class to another. In such cases, the value of instance variable is likely to change because it is accessing the value from the super-class instead of the child class.

It is recommended that before defining variables in a sub-class, the programmers take note of the variables in each of the super classes corresponding to that class and avoid such problems. This would help in preventing duplication of variables which can lead to bugs.

4.1.5 Passing arguments to methods

There are mainly two ways of passing arguments to methods:

- Pass by value
- Pass by reference

Java directly supports passing by value; however, passing by reference will be accessible through reference objects.

4.1.4.1 Pass by value

When the arguments are passed using the *pass by value* mechanism, only a copy of the variables are passed which has the scope within the method which receives the copy of these variables. The changes made to the parameters inside the methods are not returned to the calling method. This ensures that the value of the variables in the calling method will remain unchanged after return from the calling method.

Program 4.4 Pass by value.

```
import java.io.*;
public class Sum
{
    static int CalculateTotal(int n1)
    {
        int total=0;
        int marks[] = new int[3];
        try
        {
            BufferedReader br= newBufferedReader( new InputStreamReader(System.in));
            for(int i=0; i<n1; i++)
            {
                marks[i]= Integer.parseInt(br.readLine());
                total+=marks[i];
            }
        }
        catch(Exception e)
        {
            System.out.println("Array out of range");
        }
        return total;
    }
```

```
    public static void main(String args[ ]) throws IOException
    {
        int n,Max;
        System.out.println("Enter the numbers :");
        Max = CalculateTotal(3);
        System.out.println("Sum of the numbers is:" +Max);
    }
}
```

The output of Program 4.4 is as shown below:

```
Enter the numbers:
2
3
4
Sum of the numbers is: 9
```

In Program 4.4 the value 3 has been passed to the method CalculateTotal. The argument n1 of CalculateTotal method takes its value as 3 and performs the rest of the execution of program. The try and catch blocks used in the program will be explained in Chapter 5.

4.1.4.2 Pass by reference

In the *pass by reference* mechanism, when parameters are passed to the methods, the calling method returns the changed value of the variables to the called method. The call by reference mechanism is not used by Java for passing parameters to the methods. Rather it manipulates objects by reference and hence all object variables are referenced. Program 4.5 illustrates the call by reference mechanism in Java.

Program 4.5 Program to illustrate pass by reference.

```
import java.io.*;
class swap_ref
{
    static void swap(First ob)
    {
        int temp;
        temp=ob.a;
        ob.a=ob.b;
        ob.b=temp;
    }
```

```
    public static void main(String args[ ])
    {
        First ob=new First(10,20);
        System.out.println("Before SWAP");
        System.out.println("a="+ob.a+"b="+ob.b);
        swap(ob);
        System.out.println("After SWAP");
        System.out.println("a="+ob.a+"b="+ob.b);
    }
}
class First
{
    int a;
    int b;
    First(int x, int y)
    {
        a = x;
        b = y;
    }
}
```

The output of Program 4.5 is as shown below:

```
Before SWAP
a = 10 b = 20
After SWAP
a = 20 b =10
```

Here, the method swap is invoked with object ob as parameter, and any manipulation of this object within the method will affect the original objects. Thus, the values of a and b are interchanged after method swap is called.

4.1.6 Class methods

Apart from class and instance variables, Java also has class and instance methods. The differences between the two types of method are analogous to the differences between class and instance variables. Class methods are available to any instance of the class itself and can be made available to other classes. Therefore, some class methods can be used anywhere, regardless of whether an instance of the class exists or not.

In order to define class methods, the keyword static is used. This keyword should be placed in front of the method definition.

For example, the following illustration defines a class method named Max.

```
static int Max(int arg1, int arg2) { ... }
```

In Java, each data type has a wrapper class. Wrapper classes for integer, float and boolean data types are Integer, Float and Boolean, respectively. By using class methods defined in these classes, objects can be converted into data types and vice versa. For example, the parseInt() class method in the Integer class takes a string and a radix (base) and returns the value of that string as an integer:

```
int count = Integer.parseInt("42", 10) // returns 42
```

Usually methods that operate on a particular object and affect it in one way or the other are defined as instance methods. Methods that provide some general utility but do not directly affect an instance of that class are declared as class methods.

4.1.7 Overloading methods

The process of defining methods with same name but with different functionalities is termed *method overloading*. For example, an overloaded draw() method can be used to draw anything, from a circle to an image. Methods with same name, namely, draw, but with different arguments can be used for all cases.

> When a method in an object is called, Java verifies its name and argument type so that the appropriate method definition is executed.

To create an overloaded method, several different method definitions are created in the class with the same name but with different parameter lists (either in number or type of arguments). Java allows method overloading as long as each parameter list for the method in question is unique.

Java differentiates overloaded methods based on the number and type of parameters and not on the return type of the method. A compiler error would occur when two methods with the same name and same parameter list but different return types are created.

Program 4.6 gives an example of an overloaded method.

Program 4.6 Method overloading.

```
class MethOverLoad
{
    public static void First()
    {
        System.out.println("Without any arguments");
    }
```

```
    public static void First(int a, int b)
    {
        System.out.println(a+b);
    }
    public static void main(String args[ ])
    {
        First();
        First(10,20);
    }
}
```

The output of Program 4.6 is the following:

```
Without any arguments
30
```

In the above class, two methods are defined with same name, that is, First(). The first method does not contain any arguments but displays a string when called. The second method has two integers in its parameter list. Invoking this method gives the sum of the two integers. When First() is invoked from the main method it displays the text, and when First(10,20) is invoked, it adds up the two numbers and gives 30 as output.

4.1.8 Constructor methods

Constructor methods initialize new objects when they are created. Unlike regular methods, constructor methods cannot be called directly. They are called automatically when a new object is created. When an object is created in Java using the keyword new the following things happen:

- Memory is allocated for the new object.
- Instance variables of the object are initialized, either to their initial values or to default values (0 for numbers, null for objects, false for boolean, '\0' for characters.)
- A constructor method is invoked. (A class may have one or more constructor methods with different arguments lists. Based on the number of parameters and their data types, the corresponding constructor will be invoked.)

If a class does not have any defined special constructor methods, the instance variables of the new object should be set or other methods that are needed by the object to initialize itself should be called.

By defining constructor methods in classes, the initial values of instance variables can be set. Java also facilitates overloading of constructors. Constructor overloading helps to create an object that has specific properties defined on the basis of the arguments given in the new expression.

4.1.8.1 Basic constructors

Constructors look a lot like regular methods, with two basic differences:

- Constructors always have the same name as the class.
- Constructors do not have a return type.

Program 4.7 shows a simple class by the name Identity. The constructor method for Identity takes two arguments: a string object representing its name and an integer that stands for its age.

Program 4.7 Using constructors.

```
class Identity
{
    String name;
    int age;
    Identity (String n, int a)
    {
        name = n;
        age = a;
    }
    void printIdentity()
    {
        System.out.print("Name is" + name);
        System.out.println(".Age is" + age);
    }
    public static void main (String args[ ])
    {
        Identity I;
        I = new Identity("Shyam", 30);
        I.printIdentity ();
        System.out.println("--------");
        I = new Identity("Johnny", 3);
        I.print Identity ();
        System.out.println("--------");
    }
}
```

The output of this program is as follows:

```
Name is Shyam
Age is 30
--------
```

```
Name is Johnny
Age is 3
--------
```

The Identity class has three methods. The first is the constructor method which initializes the two instance variables of the class based on the arguments to new. The Identity class also includes a method called printIdentity() so that the object can 'introduce' itself and a main() method to test each of these things.

4.1.8.2 Calling another constructor

Some constructors may be supersets of other constructors defined within a class. This implies that the constructor might have the same behaviour as the other constructor plus other additional behaviour. Rather than duplicating identical behaviour in multiple constructors in a class, it is recommended to just call the first constructor from inside the body of the second constructor. Java provides a special syntax for doing this. A constructor defined within the current class can be called using the keyword this. The syntax would be as follows:

```
this(arg1, arg2, arg3...);
```

The arguments to this() are the arguments to the constructor.

4.1.8.3 Overloading constructors

Similar to methods, constructors can take many different numbers and types of parameters in the body of one program, enabling the programmer to create objects with desired functionality.

Program 4.8 shows the example of a class Rectangle that has overloaded constructors.

Program 4.8 Using overloaded constructors.

```
import java.awt.Point;
class Rectangle
{
    int x1 = 0;
    int y1 = 0;
    int x2 = 0;
    int y2 = 0;
    Rectangle(int x1, int y1, int x2, int y2)
    {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
```

```
    this.y2 = y2;
  }
  Rectangle(Point topLeft, Point bottomRight)
  {
    x1 = topLeft.x;
    y1 = topLeft.y;
    x2 = bottomRight.x;
    y2 = bottomRight.y;
  }
  Rectangle(Point topLeft, int w, int h)
  {
    x1 = topLeft.x;
    y1 = topLeft.y;
    x2 = (x1 + w);
    y2 = (y1 + h);
  }
  void printRect()
  {
    System.out.print("Rectangle: <" + x1 + "," + y1);
    System.out.println("," + x2 + "," + y2 + ">");
  }
  public static void main(String args[ ])
  {
    Rectangle rect;
    System.out.println("Calling Rectangle with coordinates 5, 5, 25, 25:");
    rect = new Rectangle(5, 5, 25, 25);
    rect.printRect();
    System.out.println("----------");
    System.out.println("Calling Rectangle with points (10,10), (20,20):");
    rect = new Rectangle(new Point(10,10), new Point(20,20));
    rect.printRect();
    System.out.println("----------");
    System.out.print("Calling Rectangle with point (10,10)");
    System.out.println(" width (50) and height (50):");
    rect = new Rectangle(new Point(10,10), 50, 50);
    rect.printRect();
    System.out.println("----------");
  }
}
```

The output of Program 4.8 is the following:

```
Calling Rectangle with coordinates 5, 5, 25, 25:
MyRect: <5, 5, 25, 25>
----------
Calling Rectangle with points (10,10), (20,20):
MyRect: <10, 10, 20, 20>
----------
Calling Rectangle with point (10,10), width (50) and height (50):
MyRect: <10, 10, 60, 60>
----------
```

4.1.9 Inheritance, polymorphism and abstract classes

A class represents a set of objects that share the same structure and behaviour. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behaviour by providing the instance methods that express the behaviour of the objects. Object oriented programming really distinguishes itself from traditional programming in that it allows classes to express the similarities among objects that share some, but not all, of their structure and behaviour. Such similarities can be expressed using the properties of inheritance and polymorphism.

4.1.9.1 Inheritance

The term inheritance refers to the fact that one class can inherit a part or all of its structure and behaviour from another class. The class that inherits the property from another class (parent class) is termed as child class (or sub-class). For example, if class B inherits all the properties of class A, then class B is called the child class and class A is called the parent class. (Sometimes the terms *derived class* and *base class* or *sub-class* and *super-class* are also used, respectively, to denote the child class and parent class.)

A sub-class can add to the structure and behaviour that it inherits. It can also replace or modify inherited behaviour (though not inherited structure). The relationship between the sub-class and super-class is sometimes shown by a diagram in which the sub-class is shown below its super-class is placed above and connected to it. This is illustrated in Figure 4.1.

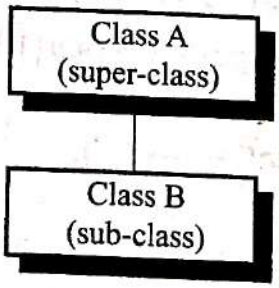

Fig. 4.1 A representation of class structure.

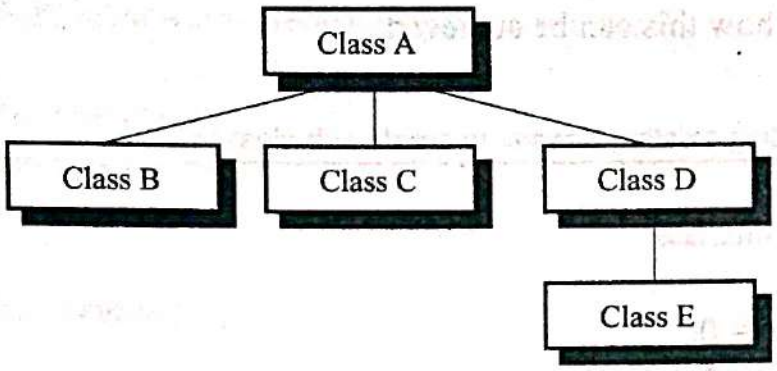

Fig. 4.2 A network of classes.

In order to inherit the properties of the parent class, the keyword **extends** is used. The syntax is as follows:

```
class B extends A
{
    .
    .   // additions and modifications of class A.
    .
}
```

Several classes can be sub-classes of the same super-class. The set of sub-classes, which may be referred to as *sibling classes*, share some structures and behaviours namely, the ones they inherit from their super-class. The super-class expresses these shared structures and behaviours. In Figure 4.2, classes B, C and D are sub-classes of class A.

Inheritance can also extend over several *generations* of classes. In Figure 4.2, class E is a sub-class of class D which in turn is a sub-class of class A. In this case, class E is considered to be a sub-class of class A even though it is not a direct sub-class.

4.1.9.2 Extending existing classes

The existing class can be extended to create a sub-class. The syntax for this is as given below:

```
class subclass-name extends existing-class-name
{
    .
    .   // Changes and additions.
    .
}
```

Program 4.9 shows how this can be achieved.

Program 4.9 Extending existing classes to create sub-classes.

```
class PrintClass
{
    int x = 0;
    int y = 1;
    void Display()
    {
        System.out.println("x is" + x + ", y is" + y);
        System.out.println("I am an instance of the class" +
        this.getClass().getName());
    }
}
class PrintSubClass extends PrintClass
{
    int z = 3;
    public static void main(String args[])
    {
            PrintSubClass obj = new PrintSubClass();
            obj.Display();
    }
}
```

The output of this program is the following:

```
x is 0, y is 1
I am an instance of the class PrintSubClass
```

As mentioned, a sub-class can also be the super-class with respect to another class. This process, by means of which a sub-class becomes a parent class of a different class is termed *multi-level inheritance*. Program 4.10 gives an illustration of multi-level inheritance.

Program 4.10 Creating and using multi-level inheritance.

```
class Simple
{
    double width;
    double height;
    double depth;
```

```
    Simple(double width, double height, double depth)
    {
        this.width=width;
        this.height=height;
        this.depth=depth;
    }
    double volume()
    {
        return (width*height*depth);
    }
}
class Weight extends Simple
{
    double weight;
    Weight(double w1,double h1,double d1,double weight)
    {
        super(w1,h1,d1);
        this.weight=weight;
    }
    void write()
    {
        System.out.println("width:" + width);
        System.out.println("height:" +height);
        System.out.println("depth:" + depth);
        System.out.println("volume:" + volume());
        System.out.println("weight:" + weight);
    }
}
class Volume extends Box
{
    double volume;
    Volume(double w2,double h2,double d2,double weight1)
    {
        super(w2,h2,d2,weight1);
    }
    public static void main(String args[ ])
    {
        Volume box1=new Volume(10,20,30,40);
        box1.write();
    }
}
```

The output of Program 4.10 is the following:

```
Width: 10.0
Height: 20.0
Depth: 30.0
Volume: 6000.0
Weight: 40.0
```

In Program 4.10, the keyword super is used. super is used to point out the constructor in the super- or parent class. When super-class constructors are initialized within the sub-class, objects and parameters are passed to the super-class using the keyword super.

4.1.10 Overriding methods

When the method of an object is called, Java seeks the method definition in the class of that object. If it does not find a match with the right signature, it passes the method call up the class hierarchy, until a definition is found. *Method inheritance* implies that methods within sub-classes can be used without having to duplicate the code; however, an object may have to respond to the same methods but have different behaviour when that method is called. Such cases call for *method overriding*. Overriding a method involves defining a method in a sub-class that has the same signature as some method in a super-class. When that method is called, the method in the sub-class is found and executed instead of the one in the super-class.

4.1.10.1 Creating methods that override previously existing methods

To override a method in a class, another method is created in its sub-class that has the same signature (name, return type, and parameter list) as the original method. When the method is called within the sub-class, since Java executes the first method definition it finds with the signature, the original method definition is effectively hidden.

Program 4.11 Overriding methods.

```
class SimpleClass
{
    public void display()
    {
        System.out.println("Overriding Method");
    }
    public class OverrideClass extends SimpleClass
    {
        public void display()
```

```
    {
        System.out.println("Overriding Simple Class Method");
    }
    public static void main(String args[ ]);
    {
        OverrideClass OC = new OverrideClass();
        OC.display;
    }
  }
}
```

The output of Program 4.11 is as shown below:

```
Overriding Simple Class Method
```

Program 4.11 shows a class OverrideClass that extends another class SimpleClass. Here, the method display() in the sub-class is overridden by the super-class method display().

Usually method overriding is implemented for the following reasons:

- to replace the definition of original method completely
- to add new functionality to the original method

Sometimes, however, the programmer would require the original method. In such cases, the method is called from within a method definition using the keyword **super**.

This can be done as shown below:

```
void myMethod (String a, String b)
{
  super.myMethod(a, b);   // method in the super class
}
```

The keyword super, is somewhat like keyword *this*. It is a placeholder for the super-class of the child class. It can be used in any place where the keyword this is used; however it refers to the super-class rather than to the current class.

Program 4.12 illustrates the way to override methods.

Program 4.12 Overriding two different methods with same name in a super-class and sub-class.

```
class X
{
    int i,j;
    X(int a, int b)
```

```
        {
            i = a;
            j = b;
        }
        void show()
        {
            System.out.println("i and j:" +i + " "+j);
        }
    }
    class Y extends X
    {
        int k;
        Y(int a, int b, int c)
        {
            super(a,b);
            k=c;
        }
        void show()
        {
            System.out.println("k :"+k);
        }
    }
    class OverrideClassA
    {
        public static void main(String args[ ])
        {
            Y subob = new Y(1,2,3);
            subob.show()    //this calls show() in Y
        }
    }
```

The output of Program 4.12 is as shown below:

```
k: 3
```

In Program 4.12, when method show() is invoked on an object of class *Y*, the method defined in class *Y* is used. That is, the method show() inside Y overrides the method declared in X.

4.1.10.2 Overriding constructors

Super-class constructors cannot be overridden as the constructors have the same name as their class. To be able to access a constructor in a sub-class with the same number and data type of

arguments as in the super-class, it must be defined in the sub-class itself. When constructors are defined in a sub-class, the corresponding super-class constructors are called during the creation of objects of that sub-class. This ensures that the initialization of inherited parts of the objects takes place similar to the way the super-class initializes its objects. Thus, defining constructors explicitly in the sub-class will override or overload super-class constructors.

Generally, super-class methods are called using the following syntax:

```
super.methodname(arguments-list)
```

This syntax cannot be used in the case of constructors, as constructors do not have a method name. The constructors of the super-class are called without having method name as shown below:

```
super(arguments-list)
```

Java follows specific rules for the use of keyword super()

1. The first calling method must be super() in the constructor's definition. If the super() method is not specified, Java will implicitly call the super() method with no arguments.
2. Super() is used to call a constructor method with the appropriate arguments from the immediate super-class. This may, in turn, call the constructor of its super-class and so on. The usage of super() in the constructor is similar to the use of keyword this().
3. To use super() in the constructor of sub-class, a constructor with the same signature must exist in the super-class.

A constructor in the super-class with exactly the same signature as the constructor in the sub-class need not be called from the sub-class. Only those constructors for which values need to be initialized in the sub-class should be called. The programmer can therefore simply create sub-classes that have constructors with entirely different signatures from any of the super-class constructors. Program 4.13 illustrates how this can be done.

Program 4.13 Using constructors with different signatures in the sub-class and super-class.

```
import java.io.*;
class One
{
    int num;
    One(int n)
    {
        this.num = n;
    }
    void show()
    {
        System.out.println("num"+num);
    }
}
```

```
class Two extends one
{
    int num1;
    Two(int x, int y)
    {
        super(x);
        num1 = y;
    }
    void show()
    {
        System.out.println("sub class number"+num1);
        System.out.println("super class number"+num);
        super.show();
    }
}
class SampleClass
{
    public static void main(String args[ ])
    {
        Two t = new Two(100,200);
        t.show();
    }
}
```

The output of SampleClass program is as shown below:

```
sub class number 200
super class number 100
num100
```

Program 4.13 defines a class SimpleClass. It creates an object of class Two which extends the class One. The class One has only one constructor, which assigns the value of *n* to *num*. The class Two has an additional instance variable (*y* in this example) and defines a constructor to initialize *x* and *y*. The constructor defined in the program calls the constructor method belonging to class One to initialize the instance variable *x* of class Two.

4.1.11 Finalizer methods

Finalizer methods are almost the opposite of constructor methods. A constructor method is used to initialize an object, while finalizer methods are called just before the object is garbage-collected and its memory reclaimed. The syntax of the finalizer method is simply finalize(). The Object class

defines a default finalizer method. To create a finalizer method, override the finalize() method using the following signature:

```
protected void finalize() throws Throwable
{
super.finalize();
}
```

The body of finalize() method can include several objects including super.finalize(). This object allows the parent class to finalize an object, if necessary. The finalize() method can be called at any time; however, calling finalize() does not trigger an object to be garbage-collected. Only removing all references to an object will cause it to be marked for deleting.

Finalizer methods are best used for optimizing the removal of an object (for example, by removing references to other objects) by releasing external resources that have been acquired (for example, external files), or for other behaviours that may make it easier for that object to be removed.

4.2 Modifiers

Modifiers are keywords used to define the scope and behaviour of classes, methods and variables in Java. Java has a wide variety of modifiers that can be categorized as shown below:

- Modifiers for controlling access to a class, method or variable, which are: public, protected and private. (These are also termed *access modifiers*).
- The static modifier for creating class methods and variables.
- The abstract modifier, for creating abstract classes and methods.
- The final modifier, for finalizing the implementations of classes, methods and variables.
- The synchronized and volatile modifiers, which are used for threads.
- The native modifier, which is used for creating native methods.

Given below are some examples of the above modifiers.

```
public class MyApplet extends Java.applet.Applet { ... }
private boolean SwitchState;
static final double pi = 3.14;
protected static final int MAXNUMELEMENTS = 128;
public static void main(String args[ ]) { ... }
```

It should be remembered that there cannot be more than one access modifier for a class, method or variable. All the modifiers are essentially optional, none have to appear in a declaration. Good object oriented programming style suggests that modifiers should be added to best describe the intended use of, and restrictions on, the things that are being declared. In special situations (for example, inside an interface), certain modifiers are implicitly defined.

4.2.1 Controlling access to methods and variables

While a class may define various methods and variables, not all of them are useful, and some may even be harmful, if they are not used in the way they are intended to be used. Access control is about controlling visibility.

From the point of class and object design, the important modifiers are those that control the visibility of and access to variables and methods inside the classes. In the following text the importance of access control and the methods to ensure different levels of visibility are described.

4.2.1.1 Why access control is important

When a method or variable is visible to another class, methods of the second class can reference (that is, call or modify) that method or variable. Protecting those methods and instance variables limits the visibility and the use of those methods and variables. As a designer of a class or an entire hierarchy of classes, it is a good idea to define what the external appearance of a class is going to be, which variables and methods will be accessible for other users of that class, and which ones are for internal use only. This is called *encapsulation* and is an important feature of object oriented design.

> Encapsulation is the process of hiding the internal parts of an object implementation and allowing access to that object only through a defined interface.

4.2.1.2 The four Ps of protection

Java provides four levels of protection for methods and instance variables: public, private, protected and package. Before applying protection levels to a program, one should know what each form means and understand the fundamental relationships that a method or variable within a class can have to the other classes in the system.

Package Methods and variables with package protection are visible to all other classes in the same package but not outside that package. Most of the time, the programmer needs to be more explicit when protection is defined for methods and variables of a class. Package protection is the default level of protection. Package protection is not an explicit modifier, it can be added to the definition of a method or variable.

Private The keyword private limits the visibility of methods and instance variables to the class in which they are defined. For example, private instance variable can be used by methods inside the same class but cannot be seen or used by any other class or object. In addition, neither private variables nor private methods are inherited by sub-classes.

Private protection ensures that methods and variables are accessible only to other methods in the same class. To create a private method or instance variable, add the private modifier to its definition as shown in Program 4.14:

Program 4.14 Implementing private access specifier for variables and methods.

```
class PrivateClass
{
    private String name = "Java";
    private int age = 20;

    public void show()
    {
        System.out.println("Name"+name);
        System.out.println("Age"+age);
    }
    private void show1()
    {
        System.out.println("Name"+name);
        System.out.println("Age"+age);
    }
}
class PrivateDemo
{
    public static void main(String args[ ])
    {
        PrivateClass p1 = new PrivateClass();
        p1.show();   //accessible
        //p1.show(); //not accessible as show1() is declared as private method
    }
}
```

The output of Program 4.14 is given below:

```
Name    Java
Age     20
```

Though the variables Name and Age in Program 4.14 are declared as private, all instance variables are private by default. Here, the method show() is public and hence it can be accessed from the method main(); however, if we include p1.show1() (as mentioned in the last comment line) in the program, compilation of the program will give an error because the method show1() is private and hence cannot be accessible from the method main().

Public The opposite of private, and the least restrictive form of protection, is public. A method or variable that is declared with the modifier public is accessible to the class in which it is defined and all the sub-classes of that class, as well as all the classes in the package and any other classes outside that package.

In many ways, public is similar to the default package. Both allow methods and variables to be accessed by other classes in the same package. The difference occurs when the package is created. Variables and methods with package protection can be used in classes that exist in the same package. But if the class is imported from outside the package, those methods and variables will not be accessible unless they have been declared public. The syntax for declaring a variable or method public is as shown below:

```
public Add()
{
}
```

Protected The final form of protection available in Java concerns the relationship between a class and its present and future sub-classes declared inside or outside a package. These sub-classes are much closer to a particular class than to any other 'outside' classes, for the following reasons:

- Sub-classes usually 'know' more about the internal implementation of a super-class.
- Sub-classes are often written by a reliable programmer who knows the source code.
- Sub-classes are frequently modified as to enhance the representation of the data within a parent class.

To support a special level of visibility reserved for sub-classes that are required to be somewhat less restricted than private, Java offers an intermediate level of access between package and private called, protected. (The definition of packages and the way to use them are given in detail in Section 4.3). The keyword protected implies that those methods and variables are accessible to all classes inside the package and only sub-classes outside the package.

A summary of types of protection The differences between various types of protection can be unclear particularly in the case of protected methods and variables. Table 4.1 describes different types of protection (access specifiers).

Table 4.1 Different types of protection.

Visibility	Public	Protected	Package (No modifier)	Private
From the same class	yes	yes	yes	yes
From any class in the same package	yes	yes	yes	no
From any class outside the package	yes	No	no	no
From a sub-class in the same package	yes	yes	yes	no
From a sub-class outside the same package	yes	yes	no	no

4.2.2 Method protection and inheritance

Setting up protections in new classes with new methods is easy. Programmers need to decide on the design first and apply the right modifiers. When sub-classes are created, certain methods are overridden; however, programmers should take into account the protection type of the original method.

In Java, the new method that overrides the original method cannot be more private than the original method; however, the new method can be more public. Java specifies certain rules for inherited methods:

- Methods declared public in a super-class must also be public in all sub-classes (this, by the way, is why most of the applet methods are public).
- Methods declared protected in a super-class must either be protected or public in sub-classes but they cannot be private.
- Methods declared private are not inherited and therefore this rule does not apply.
- Methods for which the protection type has not been declared (the implicit package protection) can take more private protection types in sub-classes.

4.2.3 Creating accessor methods

Accessor methods are used for initializing and accessing the value of instance variables. The value of these instance variables can be used further in the program. For creating accessor methods, it is required to create two methods among which one method is used to initialize the value and other is used to retrieve the value. An accessor method makes the program more readable and understandable. Moreover, accessor methods are similar to any other method, as can be seen from Program 4.15.

Program 4.15 Implementing accessor methods.

```
//A program to find area of a square.
import java.io.*;

class AccessorMethodClass
{
    int side;

    //This method is used for initializing the value of global variable 'side'
    public void setSideValue(int a)
    {
        side = a;
    }

    //This method is used for getting the value of the global value
    public int getSideValue()
```

```
    {
        return side;
    }
    public static void main(String args[ ])
    {
        int area, side;
        accessorMethodClass ob=new accessorMethodClass();
        ob.setSideValue(10);        //Initializing the value of global variable
        side = ob.getSideValue();   //Retrieving the value of Global variable
        area = side * side;         //Finding the area of square.
    System.out.println("The area of Square is" +area);
    }
}
```

The output of Program 4.15 is as shown below:

```
The area of Square is 100
```

In Program 4.15 the global variable side is initialized by the method setSideValue(int a) and its value is retrieved by the method getSideValue(). Here, the words like set and get make the functionality of accessor methods more clear even though it is not a part of any rules or specifications.

Accessor methods can also have names similar to their instance variable and Java performs the appropriate operation based on the use of these methods and variables, as shown in Program 4.16.

Program 4.16 Using accessor methods II.

```
//A program to find the area of a square which is equal to square of the side.
import java.io.*;
class AccessorMethodClass
{
    int side;
    //Note that method name and variable name is same
    public void side(int a)
    {
        side = a;
    }
    public int getSideValue()
```

```
    {
        return side;
    }
    public static void main(String args[ ])
    {
        int area, sidevalue;
        AccessorMethodClass ob = new AccessorMethodClass();
        ob.side(10);
        sidevalue = ob.getSideValue();
        area = sidevalue*sidevalue;
        System.out.println("The area of Square is"+area);
    }
}
```

The output of Program 4.16 is as shown below:

```
The area of Square is 100
```

There are some problems with the convention used for having same name as described in the above program.

- Using same name for the instance variable and method may make the program less readable and understandable. Providing meaningful names to the variables and methods with respect to their functionality is a good programming practice.
- Adopting this type of convention violates the essentials of structured programming.

4.2.4 Class variables and methods

To create a class variable or method, include the word static in front of the method's name. The modifier static typically comes after any protection modifiers. Given below is an example that illustrates how a class variable may be created in a program.

Program 4.17 Declaring variables and methods.

```
import java.io.*;
public class CircleClass
{
    public static float pi = 3.14f;
    public static float area(float r)
    {
        return pi * r * r;
    }
```

```
    public static float perimeter(float r)
    {
        return 2 * pi * r;
    }
    public static void main(String args[ ])throws IOException
    {
        float A1, P1;
        System.out.println("Enter the radius of the Circle:");
        BufferedReader br = new BufferedReader(new InputStreamReader(Sytem.in));
        int n = Integer.parseInt(br.readLine());
        A1 = area(n);
        System.out.println("Area of the Circle is:" +A1);
        P1 = perimeter(n);
        System.out.println("Perimeter of the Circle is:"+P1);
    }
}
```

The output of Program 4.17 is as shown below:

```
Enter the radius of the Circle:
5
Area of the Circle is: 78.5
Perimeter of the Circle is: 31.400002
```

Both class variables and methods can be accessed using standard dot notation by placing either the class name or an object to the left side of the dot; however, the convention that is followed is always to use the name of the class and not that of the object, to clarify that a class variable is being used and to help the reader to know instantly that the variable is global to all instances. This is as shown in the following example.

```
float circumference = 2 * Circle.pi * getRadius();
float randomNumer = Math.random();
```

An interesting example of using class variables, which brings out how their properties can be used effectively is now discussed. Program 4.18 shows a class called StudentClass that uses class and instance variables to keep track of number of students in a particular course.

Program 4.18 Effectively using class and instance variables.

```
import java.io.*;
public class StudentClass
{
    private static int numberOfStudents=0;
```

```
    private static void newStudent()
    {
        numberOfStudents++;
    }
    StudentClass()//Defining Constructor
    {
        StudentClass.newStudent();
    }
    protected static void showStudentCount()
    {
        System.out.println("The number of Students is"+numberOfStudents);
    }
    public static void main(String args[ ])
    {
        System.out.println("INITIALLY");
        Student.showStudentCount();
        int no_students =10, num=0;
        while(num<10)
        {
            new StudentClass();
            num++;
        }
        System.out.println("AFTER ADDING MORE STUDENTS");
        StudentClass =new StudentClass();
        st.showStudentCount();
    }
}
```

The output of Program 4.18 is shown below:

```
INITIALLY
The number of Students is 0
AFTER ADDING MORE STUDENTS
The number of Students is 11
```

In Program 4.18, the class StudentClass has class variable numberOfStudents which holds the number of students in a course. A class variable is initialized whenever its class is created. It is declared static because the count of the students is required to be retained across the creation of objects. It is also declared private so that it can only be accessed by the member methods and not by any method outside the class. Initially, the variable numberOfStudents is initialized to 0. Two

methods are defined: method newStudent() increases the count of the number of students and method showStudentCount() prints the number of students.

In the main method, the count of the students is printed and then new instances of class Student are created (in the while loop). During creation of instances of class StudentClass, the constructor StudentClass() automatically increments the count of the students.

4.2.5 Finalizing classes, methods and variables

The modifier final is used to finalize classes, methods and variables. Finalizing a thing effectively 'freezes' the implementation or value of that thing. More specifically, here is how final works with classes, variables and methods, respectively:

- When the modifier final is applied to a class, it means that the class cannot be inherited.
- When final is applied to a variable, it means that the variable is constant.
- When final is applied to a method in a class, it means that the method cannot be overridden in the sub-classes.

The above points are now discussed in more detail.

4.2.5.1 Finalizing classes

In order to finalize a class, the modifier final is added to the class definition. Typically, it is added after protection modifiers such as private or public. This is done as shown below:

```
public final class FinalClass1
{
    ...
}
```

A class is declared final for the following reasons:

- To prevent inheritance
- For better efficiency. Final classes allow programmers to rely on instances of only that class and optimize those instances

4.2.5.2 Finalizing variables

The value of a finalized variable cannot be changed. It is then, effectively, a constant. To declare constants in Java, final variables with initial values are used. This declaration can be done in a program as follows:

```
public class FinalClass2
{
    public static final int ConstantInteger = 100;
```

```
    public final String ConstantString = "Java Book";
}
```

Local variables (those inside blocks of code surrounded by braces; for example, in while or for loops) cannot be declared final.

4.2.5.3 Finalizing methods

Methods that cannot be overridden are known as finalized methods. The implementations of final methods cannot be redefined in sub-classes. The syntax for declaring a method final is the following.

```
public class FinalMethodClass
{
    public final void One()
    {
        ...
    }
}
```

Declaring methods final improves their efficiency.

4.2.6 Abstract classes and methods

One way of declaring the functionality of a class or method is by using the keyword **abstract**. When classes are arranged in an inheritance hierarchy, the presumption is that 'higher' classes are more abstract and general, whereas 'lower' sub-classes are more concrete and specific. Often, when hierarchies of classes are designed, common design and implementations are factored into a shared super-class. This super-class will not have any instances; its sole reason for existing is to act as a common, shared repository of information that its sub-classes use. In simpler terms, a class with minimal functionality is known as an abstract class. Usually the sub-class of an abstract class contains the functionality. To define a class as **abstract**, the keyword **abstract** is used. The syntax for using **abstract** is as follows:

```
public abstract class ExampleAbstractClass
{
    ...
}
```

Abstract classes can never be instantiated, but they can contain class and instance variables and methods.

In addition, abstract classes can also contain abstract methods. An abstract method is a method signature with no implementation, and sub-classes of the abstract class are expected to provide the

implementation for that method. Abstract methods function in the same manner as abstract classes: they are a way of factoring common behaviour into super-classes and then providing specific concrete uses of those behaviours in sub-classes. Abstract methods can only exist inside abstract classes. This is because abstract methods cannot be called and they have no implementation. These points may be summarized as shown below:

- *Abstract classes* are classes whose sole purpose is to provide common information for sub-classes. Abstract classes can have no instances.
- *Abstract methods* are methods with signatures, but no implementation. The sub-classes of the class that contains that abstract method must provide its actual implementation.

Concrete classes function in a way just opposite to abstract classes. They are classes that can be instantiated. Concrete methods are those that have actual implementations. Abstract methods are declared with the modifier abstract which, usually, is placed after the protection modifiers but before the modifiers static or final. In addition, they have no body.

Program 4.19 is an example that shows three simple classes: one abstract class and two concrete classes. The abstract class CarClass consists of (i) one instance variable regno, (ii) one constructor, (iii) two abstract methods, namely, Steering() and Brake(), and (iv) one concrete method display(), which is defined normally. The two concrete classes are Maruti and Santro, which are the sub-classes of class CarClass. These two sub-classes provide the implementation of methods Steering() and Brake() and inherits the remaining behaviour from CarClass.

Program 4.19 Using abstract and concrete classes and methods.

```
abstract class CarClass
{
    int regno;
    CarClass(int r)
    {
        regno = r;
    }
    abstract void Steering(int dir);
    abstract void Brake(int force);
    public void display()
    {
        System.out.println("Car Registration Number is" +regno);
    }
}
class Maruti extends CarClass //Concrete sub-class of CarClass class
```

```
{
    Maruti(int regno)
    {
        super(regno);
    }
    void Steering(int dir)
    {
        System.out.println("Maruti Details");
        System.out.println("Maruti Uses Normal Steering");
    }
    void Brake(int force)
    {
        System.out.println("Maruti Uses Gas Brakes");
    }
}
class Santro extends CarClass    //Another concrete sub-class of CarClass
{
    Santro(int regno)
    {
        super(regno);
    }
    void Steering(int dir)
    {
        System.out.println("Santro Details");
        System.out.println("Santro Uses Power Steering");
    }
    void Brake(int force)
    {
        System.out.println("Santro Uses Air Brakes");
    }
}
class SampleCar
{
    public static void main(String args[ ])
    {
        Maruti m = new Maruti(1002);
        Santro s = new Santro(5440);
        CarClass ref, ref1;
        ref = m;
        ref.display();
        ref.Steering(3);
```

```
        ref.Brake(20);
        ref1 = s;
        ref1.display();
        ref1.Steering(6);
        ref1.Brake(205);
    }
}
```

The output of Program 4.19 is as shown below:

```
Car Registration Number is 1002
Maruti Details
Maruti Uses Normal Steering
Maruti Uses Gas Brakes
Car Registration Number is 5440
Santro Details
Santro Uses Power Steering
Santro Uses Air Brakes
```

An object of abstract class cannot be created since it contains an abstract method (which is only declared but not defined). A sub-class, which extends an abstract class, must implement the abstract methods declared in abstract class. An object can be created for a concrete class, but not for abstract class. Hence, any attempt to create an object of abstract class will give a compilation error.

4.3 Packages

A package in Java is a group of related classes, interfaces and sub-packages. For example, java.io package contains classes and interfaces for managing various kinds of input and output. The default package is java.lang, which includes primary classes and interfaces essential for Java language. It also consists of wrapper class, Strings and multithreading. Packages provide a high-level layer of access protection and name space management.

Advantages of using packages are listed below:

- They help in organizing classes into units. Just as a computer hard disk contains folders or directories to organize files and applications, packages allow organization of classes into groups.
- They reduce problems with conflicts in names. As the number of Java classes grows, so does the likelihood that the same class name would be repeated. This would result in clashes while naming and unwanted errors. Packages allow programmers to 'hide' classes so that conflicts can be avoided.

- Packages provide protection to variables, methods and classes in finer ways than can be done on a class-by-class basis.
- Packages can be used to identify classes. Furthermore, when proper naming conventions are adopted, packages can be used to identify the programmer / organization.

Although a package is typically a collection of classes, they can also contain other packages, thereby forming yet another level of organization somewhat analogous to the inheritance hierarchy. Each 'level' usually represents a smaller, more specific grouping of classes. The Java class library itself is organized along these lines. The top level is called Java, the next level includes names such as IO, net, util and awt.

4.3.1 Using packages

In order to access a package within a particular program the keyword import followed by the package name is used. All the classes, variables and methods within a package can be imported. Java provides three mechanisms of using a class contained in a package:

- If the class is contained within the package java.lang (for example, System or Date), then the class can be utilized by just giving its reference. This is possible because java.lang is the default package.
  For example,

  ```
  System.out.println("Learning packages");
  ```

  Since System is a class in the default package, there is no need of importing java.lang package or class java.lang.system. So, the class System can be used by just writing its class name, wherever it is used in a program.
- If the class is in some other package (other than default package), that class can be referred by its full name (for example as java.awt.Font). Classes that are used frequently are imported individually (or a whole package of classes). After a class or a package has been imported, it can be referred to using its class name.
- If classes are not explicity associated with a package, Java puts them into an unnamed default package. These classes can be accessed by just invoking their class names from anywhere in the code.

4.3.2 Using package and class names in full

To refer to a class in some other package, the syntax is to use the full name of the class preceded by the package name. In such cases, classes or packages need not be imported specifically. Instead, the following signature can be followed.

```
java.awt.Font f = new java.awt.Font()      // This accesses the Font class from
                                           // the java.awt package.
```

For classes that are scarcely used in a program, it makes more sense to use their full names. Whereas, the keyword import should be utilized if classes of a particular package are used multiple times (or if the packages and sub-packages' names are lengthy).

4.3.3 The import command

As mentioned earlier, to refer to classes belonging to other packages, the import keyword is used. This keyword ensures that the desired package with its contents is available to the program in which it is invoked. Using import, the entire package or individual class can be imported. The syntax for this is as in the following example:

```
import java.util.Vector;
import java.awt.*
```

When the asterisk (*) is used, it implies that all the public classes, methods and variables can be accessed by the class in a program in which the package is imported.

The line import java.awt.*, in the above example, imports all the public classes in that package, but does *not* import sub-packages such as image and peer. To import all the classes in a complex package hierarchy, it is necessary to explicitly import each level of the hierarchy. Java also prevents use of partial class names. For example, L* will not import all the classes that begin with L, every class name has to be spelt out in full. Usually import statements are written at the top line within the program, before any class definitions (but after the package definition).

4.3.4 Name conflicts

After a class or a package of classes is imported, it is referred to by its name, without the package identifier; however, an explicit reference is required when there are multiple classes with the same name from different packages. For example, consider the case when a program imports two packages, as shown below,

```
import java.util.*;
import java.sql.*;
```

Both packages, java.util and java.sql, contain a class called Date. The meaning and implementation of these two classes, however, are entirely different. It would be difficult to understand from which of the packages the Date class is being referred to, especially when the Date class is referred in the program as follows:

```
Date date = new Date();
```

In such cases, the Java compiler will neither complain about a naming conflict nor refuse to compile the program. As a result, the programmer will have to refer to the appropriate class by using the full package name as given below:

```
java.util.Date date = new java.util.Date();
```

4.3.5 Creating packages

Java allows programmers to create their own packages. Package creation is quite easy. The three steps that follow are used to create a package:

Defining the package name The name of the package that is to be created is determined in this step. The choice of a package name depends on how its classes will be used. One convention for naming packages that has been recommended by Sun Microsystems is to use the Internet domain name with the elements reversed.

So, for example, if Sun Microsystems were following their own recommendation, their packages would be referred to using the name com.sun.java rather than just java. If the Internet domain name is idrbt.ac.in, the package name might be in.ac.idrbt.

The idea is to make sure that the package name is unique. Although packages can hide conflicting class names, the protection stops there. It is impossible to ensure that there is no conflict in package names. By convention, package names tend to begin with a lowercase letter, to distinguish them from class names.

When java.awt.Font is imported it is clearly seen that Font is the class and awt is the package. This convention helps reduce name conflicts.

Creating a directory structure Step two is to create a directory structure that matches the package name. The name of the directory should be same as that of the package name (if the package consists of only one name); however, if the package name is a composite, as, for example, idrbt.ac.in, a directory by name idbrt is created, followed by one called ac (which is within the directory idrbt) and a directory by name in (which is within the directory ac). Classes and source files are usually contained in the directory in.

Using the command package to add the class to a package The final step is to put the class inside packages. This is achieved by adding the command package to the source files. This command gives a direction to imply that this class goes inside this package. The single package command, if any, must be the first line of the code, after any comments or blank lines and before any import commands. If a class does not have a package command in it, that class is contained in the default package and can be used by any other class. When using packages, however, it should be ensured that all classes should belong to some package. The use of the package command is illustrated in Program 4.20.

Program 4.20 Creating and using packages.

```
package SamplePackage;
import java.io.*;
public class PackageClass
{
    int from,to;
```

```
    PackageClass(int f,int t)
    {
        from=f;
        to=t;
    }
    public void doubling()
    {
        System.out.println("Values from" +from+ "to" +to+ "on doubling give");
        for(int i=from;i<=to;i++)
        {
            int k=i<<1;
            System.out.println(k);
        }
    }
    public static void main(String args[ ])
    {
        PackageClass pkg=new PackageClass(100,110);
        pkg.doubling();
    }
}
```

If the program has been saved in c:\mydirectory then the following steps specify how to compile and run the program.

```
c:>mydirectory >javac –d .PackageClass.java
```

This command will store the class file in the path c:\mydirectory\SamplePackage. If SamplePackage directory does not exist already then it is created during the compilation of the file. In the above command, d specifies the destination directory in which class files have to be stored and the dot . refers to the present directory. After creating the directory SamplePackage, to run the program the following command is used.

```
c:>mydirectory >java SamplePackage.PackageClass
```

However, an error occurs when the command is executed. This is because the path of the directory in which the package exists is not known. To overcome this problem the *class path* to the directory in which package is present has to be set.

Classpath is an environmental variable which makes all classes available to the programmer at an instance. The class path is set as follows:

```
c:>mydirectory> set classpath=c:/mydirectory;%classpath%
```

Following this the program is executed as follows:

```
c:>mydirectory >java SamplePackage.PackageClass
```

Another way to run the program is to move one directory up and then run the program with the package name before the class name.

```
c:>mydirectory >cd SamplePackage
c:>mydirectory >test> java SamplePackage.Packageclass
```

The output given will be the following:

```
Values from 100 to 110 on doubling give
200
202
204
206
208
210
212
214
216
218
220
```

4.3.6 Packages and class protection

By default, classes have package protection, which means that the class is available to all the other classes in the same package but is not visible or available outside that package, not even to sub-packages. It cannot be imported or referred to by name. Classes with package protection are hidden inside the package in which they are contained. Package protection comes about when a class is defined as follows:

```
class HiddenClassA extends HiddenClassB
{
    ...
}
```

To make a class visible and importable outside the package, it should be given public protection by adding the public modifier to its definition:

```
public class VisibleClass
{
    ...
}
```

Classes declared as public can be imported by other classes outside the package. When an import statement is used with an asterisk (*), only the public classes inside that package can be accessed. Hidden classes remain hidden and can be used only by the other classes in that package.

Program 4.21 has two classes: Dataelements and Student that give an insight of how packages and class protection are interrelated. The complete code of this program is across two files. The first file contains the package Studentpackage and the second file contains the main program StudentClass.

Program 4.21 Class protection in packages.

File name: Dataelements.java

```
package studentpackage;
public class Dataelements
{
    public int rollno;
    public float marks1, marks2, marks3;
    public String name;
}
```

File name: StudentClass.java

```
import studentpackage.*;
import java.io.*;
class StudentClass
{
    public static void main(String args[ ]) throws IOException
    {
        studentpackage.Dataelements ob=new studentpackage.Dataelements();
        ob.rollno = 10;
        ob.name = "Vishal";
        System.out.println("Roll no is" + ob.rollno);
    }
}
```

The output of the StudentClass program is given below:

```
Roll no is 10
```

In the file Dataelements.java, class Dataelements is defined. The data elements in this class are defined as public and the class itself is accessed using public access specifier. Whenever the program is compiled, the class file Dataelements.class is stored in a folder Studentpackage. The main program accesses the class Dataelements. An instance of the class Dataelements is created using the statement studentpackage.Dataelements. For a method or variables defined in a created

using the statement studentpackage.Dataelements. For a method or variables defined in a class to be accessible from outside the class, all the methods as well as the variables in the called class should be public. If any method or variables is not public then the class will not be accessible, providing the class protection. If the variable rollno is private, the program will give a compilation error.

4.4 Interfaces

Interfaces, like abstract classes and methods, provide templates of behaviour that other classes are expected to implement. Interfaces provide far more functionality to Java and to class and object design than simple abstract classes and methods.

4.4.1 Interfaces and classes

Classes and interfaces, despite their different definitions, have lot of common functionality. Interfaces, like classes, are declared in source files. Like classes, they are compiled using the Java compiler into .class files. And, in most cases, an interface can be used anywhere that a class can be used (as a data type for a variable, as the result of a cast and so on).

Interfaces complement and extend the power of classes, and the two can be treated almost exactly alike. One of the few differences between them is that an interface cannot be instantiated, the keyword new can create only instances of classes.

4.4.2 Creating and extending interfaces

Interfaces look very much like classes. They are declared in much the same way and can be arranged in a hierarchy. There are rules for declaring interfaces that must be followed. These are described in this section.

4.4.2.1 Creating an interface

To create an interface, the keyword interface should be used. The interface definition would be as follows:

```
public interface ExampleInterface
{
    ...
}
```

This is, effectively, the same as defining a class, with the difference that the word interface replaces the word class. An interface can contain methods and constants. The methods within the interface can carry modifiers such as public and abstract. A method inside an interface cannot be declared private or protected.

The following illustration declares an interface which contains a method that is public as well as abstract.

```
public interface ExampleInterface1
{
    public abstract void method1();  //explicitly public and abstract
    void method2();          // effectively public and abstract
}
```

Methods inside interfaces do not have bodies. An interface does not offer any implementation.

In addition to methods, interfaces can also have variables, but those variables must be declared public, static and final (making them constants). Another example, which illustrates the above fact, is the following:

```
public interface ExampleInterface2
{
    public static final int triple = 3;
    int twice= 2;     // becomes public static and final
    public abstract void calculatetwice();    //explicitly public and abstract
    void calculatetriple();        // effectively public and abstract
}
```

Interfaces must have either public or package protection, just like classes; however, interfaces without the public modifier do not automatically convert their methods to public and abstract. Their constants also will not be converted to public. An interface that is not public also has methods and constants that are not public and can be used only by classes and other interfaces in the same package. Interfaces, like classes, can belong to a package. This can be achieved by adding a package statement to the first line of the class file. Interfaces can also import other interfaces and classes from other packages, just as classes can.

4.4.2.2 Defining methods inside an interface

As mentioned in section 4.4.2, methods within interfaces define functionality but do not provide any implementation. These methods can contain parameters. By defining method parameters to be interface types, generic parameters are created that apply to any class that might use this interface. The interface Runnable in the package java.lang is a perfect example of an interface. The following example illustrates the definition of a method within an interface.

```
public interface LinkedList
{
    public void search(Object o);
}
```

4.4.2.3 Extending an interface

As with classes, interfaces can be organized into a hierarchy. When one interface inherits from another interface, that 'sub-interface' acquires all the method definitions and constants that its 'super interface' defines. To extend an interface, the keyword extends is used. The example below shows how an interface LinkList may be extended.

```
public interface TwoWayList extends LinkedList
{
    public void deleteLast(int num);
}
```

Unlike classes, the interface hierarchy has no equivalent of the Object class. This hierarchy is not rooted at any one point. Interfaces can either exist entirely on their own or inherit from another interface. Multiple inheritance is also permitted in interfaces. A single interface can extend as many interfaces as it needs to (separated by commas in the extends part of the definition), and the new interface will contain a combination of all its parent methods and constants. This follows the syntax shown in the following example:

```
public interface EventListener implements ActionListener,MouseListener,KeyListener
{
    ...
}
```

In the above illustration, the interface EventsListener extends several other interfaces. The rules for managing method name conflicts in multiple inheritance are similar to those for classes that use multiple interfaces; methods that differ only in return type will result in a compilation error.

4.4.2.4 Implementing and using interfaces

The keyword implements In order to use the functionality of an interface within a class the implements keyword is used. Program 4.22 illustrates all the features of interfaces that have been described so far. This program computes the total (or percentage) marks for a particular student.

Program 4.22 Implementing and using interfaces.

```
import java.io.*;
                                //Defining an Student Interface
interface StudentInterface
{
    void getStudentData() throws IOException;
    void showStudentData();
```

```
    float calculateTotal();
    float calculatePercentage();
    void menu();
}
class StudentResult implements StudentInterface
{
    int rollno;
    float marks1,marks2,marks3;
    String name;
                        //method to obtain the student Data
    public void getStudentData()throws IOException
    {
        System.out.println("Enter the Roll Number");
        Rollno=Integer.parseInt(dis.readLine());
        DataInputStream dis = new DataInputStream(System.in);
        System.out.println("Enter the Name");
        name = dis.readLine();
        System.out.println("Enter the marks in three subjects:");
        marks1 = Integer.parseInt(dis.readLine());
        marks2 = Integer.parseInt(dis.readLine());
        marks3 = Integer.parseInt(dis.readLine());
    }
    //method to show the student data
    public void showStudentData()
    {
        System.out.println("The Particulars of the Student are");
        System.out.println("Name : " +name);
        System.out.println("Roll No : " +rollno);
        System.out.println("Marks are" +marks1+" "+marks2" "+marks3);
    }
        //method to calculate the total of the marks obtained by a student
    public float calculateTotal()
    {
        return(marks1+marks2+marks3);
    }
    //method to calculate the percentage of the marks obtained by student
    public float calculatePercentage()
    {
        float tot = marks1 + marks2 + marks3;
        return(tot/3);
    }
```

```
    //method to show the menu of the functions that can be performed
    public void menu()
    {
        System.out.println("ENTER 1 FOR CALCULATING TOTAL");
        System.out.println("ENTER 2 FOR CALCULATING PERCENTAGE");
    }
}
class InterfaceClass
{
    public static void main(string args[ ])throws IOException
    {
        StudentResult ob1 = new StudentResult();
        ob1.getStudentData();
        ob1.showsStudentData();
        Float total, avg,percent;
        DataInputStream dis=new DataInputStream(System.in);
        ob1.menu();
        int n=Integer.parseInt(dis.readLine());
        Switch(n)
        {
            case 1:   total = ob1.calculateTotal();
                      System.out.println("The Total marks are "+total);
                      break;
            case 2:   percent = ob1.calculatePercentage();
                      System.out.println("The Percentage of Marks are "+percent);
                      break;
            default:  System.out.println("You have Entered Wrong Choice");
        }
    }
}
```

The output of Program 4.22 as shown below:

```
Enter the Roll Number
23
Enter the Name
xyz
Enter the marks in three subjects:
78
88
```

```
89
The Particulars of the Student are
Name : xyz
Roll No : 23
Marks are 78.0 88.0 89.0
ENTER 1 FOR CALCULATING TOTAL
ENTER 2 FOR CALCULATING PERCENTAGE
2
The Percentage of Marks are : 85.0
```

In Program 4.22, the interface StudentInterface consists of five method declarations. The functionality of each method is defined in the class StudentResult which implements the interface StudentInterface. In the method getStudentData, student details are captured. Method showStudentData displays the details of a student. Methods calculateTotal and calculatePercentage calculates total marks obtained by the student and percentage of the marks obtained by the student respectively. The method main in the class InterfaceClass uses the methods declared in the interface StudentInterface.

Since interfaces provide only abstract method definitions, they should be implemented within that class, using the same method signatures from the interface. Once an interface is included, all the methods within it have to be implemented.

After the class implements an interface its sub-classes will inherit those new methods (and can override or overload them). If the class inherits from a super-class that implements a given interface, it is not necessary to use the keyword implements in the class definition.

Implementing multiple interfaces Unlike the singly inherited class hierarchy, several interfaces can be used in a class. The class will implement the combined behaviour of all the included interfaces. To include multiple interfaces in a class, interface names are separated with commas.

Complications may arise from implementing multiple interfaces. For one thing, there can be two different interfaces defining the same method. This problem can be solved in three ways:

- If the methods in both interfaces have identical signatures, a method that satisfies the definitions given in both interfaces is implemented.
- If the methods have different parameter lists, it is a simple case of method overloading. In such cases, both method signatures are implemented and each definition satisfies its respective interface definition.
- If the methods have the same parameter lists but differ in return type, a method that satisfies both is created. (It must be kept in mind that method overloading is triggered by parameter lists, not by return type.) In this case, trying to compile a class that implements both interfaces will produce a compiler error.

4.4.3 Other uses of interfaces

Interfaces can be used in all the situations where a class is used. A variable can be declared to be of a type referring to an interface in the following manner. In this example, the variable RunnableObject is of the type Runnable, where Runnable is an interface.

```
Runnable RunnableObject = new AnimationClass()
```

When a variable is declared to be of a type referring to an interface, it simply means that any object the *variable* refers to is expected to have implemented that *interface*. This implies that the object is expected to understand all the methods that interface specifies. It assumes that a promise made between the designer of the interface and its eventual implementors has been kept. In the above example, because RunnableObject contains an object of the type Runnable, the assumption is that RunnableObject.run() can be called.

The important thing to realize here is that although RunnableObject is expected to have the run() method. The code could be written long before any classes that qualify are actually implemented (or even created). In traditional object oriented programming, a class is created with stub implementations to get the same effect.

(For additional information on the Class String Tokenizer see http://www.universitiespress.com/downloads/books/81-7371-572-6.pdf - Section 4.5 *The String Tokenizer*.)

Objective type questions

1. _____ is a method with the same name as the class that is generally used to initialize the instance variables and does not return any value, not even void.
2. A sub-class defining a method that has same signature and structure as that of a method in the parent class is called _____.
3. The process of having methods with same name but having different parameter list and return types is called _____.
4. The data members in Interface are _____ and _____.
5. The _____ package is imported when using the StringTokenizer class.
6. What would be the output if the following code is executed?

```
import java.io.*;
protected class MyClass
{
    public static void main(String args[ ])
    {
        System.out.println("My First Class");
    }
}
```