Chapter : 4

Classes and Methods

4.0 Objectives
4.1 Introduction
4.2 Declaring a Class
4.3 Methods
4.4 Constructors
  4.4.1 Constructors
  4.4.2 Parameterised Constructors
  4.4.3 This keywords
  4.4.4 Instance Variable hiding
4.5 Garbage Collection
4.6 Overloading
  4.6.1 Overloading Methods
  4.6.2 Overloading Constructors
4.7 Using Objects as Parameters
4.8 Call by Value and Call by Reference
4.9 Returning Objects
4.10 Recursion
4.11 Access Specifiers
4.12 Static Keyword
4.13 Final Keyword
4.14 Nested and Inner Classes
4.15 Inheritance
  4.15.1 Inheritance a class
  4.15.2 Super
  4.15.3 Multilevel Hierarchy
  4.15.4 Method Overriding
  4.15.5 Dynamic Method Dispatch
  4.15.6 Using Abstract Classes
  4.15.7 Final
4.16 Object Class
4.17 Summary
4.18 Check Your Progress - Answers
4.19 Questions for Self - Study
4.20 Suggested Readings

4.0 OBJECTIVES

Dear Students,

After studying this chapter you will able to :

- explain classes and method in Java
- describe constructors
- discuss method overloading and over riding
- explain the use of the static and final keywords
- explain concepts of recursion, call by value, call by reference
- discuss application of nested and inner classes
- discuss inheritance in Java

4.1 INTRODUCTION

A **class** is a logical construct upon which the entire Java language is built. A class defines the shape and nature of an object. Any concept that you wish to implement in a Java program, must be encapsulated within a class. A class defines a new data type. This data type is then used to create objects of that type. Thus a class is a template for an object and an object is an instance of a class.

When you define a class, you specify the data it contains and the code that operates on that data.

The general form of declaring a class is :

```
class classname {
    type instance-variable1;
    type instance-variable2;
    ...
    ...
    type instance-variablen;
    type methodname1(parameter list) {
        body
    }
    type methodname2(parameter list) {
        body
    }
    .....
    ....
    type methodnamen(parameter list) {
        body
```

```
        }
}
```

The variables defined within a class are called *instance variables*. The code is contained within methods. The variables and the methods within a class are called as *members* of the class. It is the methods that determine how the variables of the class i.e. the class' data can be used.

Each instance of a class contains its own copy of the instance variables. This implies that the data for one object is separate and unique from the data of another object. Remember that Java classes do not need to have a **main()** method. You only specify a **main()** method in a class if that class is the starting point of your program. Also remember that applets do not have a **main()** method.

4.2 DECLARING A CLASS

Let us begin our study of classes and methods with the following example.

```
class Volume {
        int length;
        int breadth;
        int height;
}
```

This is a class with the name **Volume** which defines three instance variables length, breadth and height. We have now created a new data type called Volume. We now use this data type to create objects of type Volume. A class declaration only creates a template, it does not create any objects. To actually create a object of the type Volume you use a statement as follows :

```
Volume vol = new Volume();          //create an object vol of type Volume
```

Thus **vol** will be an object of **Volume** and a physical reality. Every object of the type Volume will contain its own copies of the instance variables length, breadth and height.

It is possible to create multiple objects of the same class. Remember that changes to the instance variables of one object will have no effect on the instance variables of another object.

You can create multiple objects for the above class Volume as:

```
Volume vol2 = new Volume();
Volume vol3 = new Volume();
```

The **new** operator dynamically allocates memory for an object and returns a reference to it. This reference is the memory address of the object allocated by **new**. Thus in Java all objects must be dynamically allocated. The advantage of dynamic allocation is that the program can create as many objects as required during program execution. It may also be possible, that due to memory not being available, **new** may not be able to allocate memory for an object. In such

situations a run-time exception occurs.

The above method of declaring objects can also be declared in two steps as follows :

Volume vol1; //declare a reference to object

vol1 = new Volume(); // allocate the object

Thus we can see that the **new** operator dynamically allocates memory to the object. Its general form can be written as :

clas-var = new *Classname*();

where *class-var* is a variable of type *classname*. Note that the class name followed by the pair of parenthesis specifies the constructor for the class. A **constructor** defines what occurs when a class is created. If no explicit constructor is specified, Java automatically supplies the default constructor. We shall study how to define our own constructors subsequently.

It is important to note here that Java's simple types like integers or characters are not implemented as objects, and therefore you do not need to use the **new** operator with them. Java treats objects differently than the simple types.

To access the instance variables we make use of the dot (.) operator. eg. to access the length variable of vol you and assign it a value 10, you can use the following statement:

vol.length = 10;

Having understood the basic concept, let us now write a program to determine the volume by making use of the above class.

```
class Volume {
        int length;
        int breadth;
        int height;
}
class Demo {
        public static void main(String args[ ]) {
                Volume vol = new Volume();
                int v;
                vol.length = 10;
                vol.breadth = 10;
                vol.height = 10;
                v = vol.length * vol.breadth * vol.height;
                System.out.println("Volume is :" + v);
        }
```

}

Remember that the **main()** method is in the class Demo. Therefore save the file with the filename Demo.java. When you compile this file, you will see that two **.class** files are created, one for class Volume and one for class Demo. Thus each class is put into its own **.class** file. You can also put each class in a separate file. To run the program, you execute the Demo class. The output would be :

Volume is 1000

Now consider the following example :

Volume vol1 = new Volume();

Volume vol2 = vol1;

In such a situation it is important to remember that both **vol1** and **vol2** refer to the same object. This means any changes made to **vol1** will affect the object to which **vol2** is refering because they are both the same object. Now a subsequent assignment to vol1 will unhook **vol1** from the original object. However, neither the object nor **vol2** will be affected. eg.

Volume vol1 = new Volume();

Volume vol2 = vol1;

....

vol1 = null;

In this case, vol1 has been set to **null** but vol2 still points to the original object.

This implies that when you assign one object reference variable to another object reference variable, you do not create a copy of the object, you are only making a copy of the reference.

4.1 & 4.2 Check Your Progress.

1. Write True or False

a) A class defines a new data type.

b) Java does not supply a default constructor.

c) In Java, all objects are dynamically allocated.

d) All classes in Java should have a main() method.

2. Fill in the blanks.

a) The ..... operator dynamically allocates memory for an object.

b) A ..... is a template for an object and an object is an..... of a class.

4.3 METHODS

The code that operates on the data is contained in the methods of a class. The general form of a method is :

type name(*parameter-list*) {

body of the method

}

type specifies the data type returned by the method. This can be any valid data type including the class type that you create. If a method does not return a value, its return type must be declared **void**. *name* specifies the name of the method. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are variables which receive values of the arguments passed to the method, when it is called. If a method has no parameters, the parameter list will be empty.

Remember that methods which have a return type other than **void**, should return a value to the calling routine using the **return** statement as follows :

return *value*;

where *value* is the value returned.

We know that we can use methods to access the instance variables defined by the class. Methods define the interface to most classes and therefore the internal data structures remain hidden. Let us modify the above program to add a method to compute the volume.

```
class Volume {
        int length;
        int breadth;
        int height;
void calvol() {          // method to display volume
                System.out.print("Volume is :");
                System.out.println(length * breadth * height);
        }
}
class Demo {
        public static void main(String args[ ]) {
                Volume vol = new Volume();
                int v;
                vol.length = 10;
                vol.breadth = 10;
                vol.height = 10;
                vol.calvol();              // call method to
                                        display volume
        }
}
```

The output of the program would be :

Volume is : 1000

/**vol.calvol()** invokes the method **calvol()** on the object **vol**. When the method is executed the Java run time system transfers control to the code defined inside the volume. When the code is executed control is transferred back to the calling routine and execution continues from the following line of code. Remember that in the method **calvol()** the instance variables are accessed directly without the dot operator. This is because a method is always invoked relative to some object of its own class.

In the above example, we have declared the return type of our method to be **void** which means our method will not return any value to the calling routine. We can modify the above method by computing the volume in the method and making the method return the value to the calling routine. The modified method may be written as :

```
int calvol() {
                    return length * breadth * height;
}
```

In our calling routine we can receive this value in a variable as :

```
v = vol.calvol();
```

where v is a variable of type **int**. Alternatively the volume can directly be printed without an intermediate variable as:

```
System.out.println("Volume is :" + vol.calvol());
```

In this case, vol.calvol() will be called automatically and its value passed to **println()**.

Remember :

- the type of data returned by a method must be compatible with the return type specified by the method.

- The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

Parameterised methods :-

Parameters allow a method to be generalised. A parameterised method can operate on a variety of data and can be used in a number of slightly different situations. A parameter is a variable defined by a method which receives a value when the method is called. An argument is a value that is passed to the method when it is invoked.

We can further modify the above program and add a method which will receive values from the invoking routine as parameters. These can then be used to compute the volume. Here is a revised version of the program where we add a method **setval()**.

```
class Volume
{
                    int length;
```

```
        int breadth;
        int height;
        int calvol()                    // method to compute volume
        {
                return length * breadth * height;
        }
        void setval(int l, int b, int h)
        {
                length = l;
                breadth = b;
                height = h;
        }
}
class Demo
{
        public static void main(String args[ ])
        {
                Volume vol = new Volume();
                vol.setval(10,5,10);        // call set
                                                val to set values
int v = vol.calvol();                       // call method to compute
                volume
                System.out.println("Volume is : " + v);
        }
}
```

Thus the method **setval()** is used to set the dimensions length, breadth and height.

4.3 Check Your Progress.

1. Fill in the blanks.

a) If a method is declared ..... it will not return any value to the calling routine

b) An ..... is a value that is passed to the method when it is invoked.

c) A ..... is a variable defined by a method which receives a value when the method is called.

4.4 CONSTRUCTORS

4.4.1 Constructors

It can be time consuming to initialise all of the variables in a class each time an instance is created. Therefore Java allows objects to initialise themselves when they are created. This automatic initialisation is performed through the use of a constructor. A constructor initialises an object as soon as it is created. It has the same name as the name of the class in which it resides. Constructors do not have a return type, not even **void**. This is because the implicit return type of the class' constructor is the class type itself. Let us modify the above program by adding a constructor to it.

```
class Volume {
        int length;
        int breadth;
        int height;
        Volume() {
            length = 10;
            breadth = 5;
            height = 10;
        }
        int calvol() {          // method to calculate volume
            return length * breadth * height;
        }
            }
class Demo {
        public static void main(String args[ ]) {
            int v;
            Volume vol = new Volume();
            Volume vol2 = new Volume();
            v = vol.calvol();       // call method to
                                    compute volume
            System.out.println("Volume is : " + v);
            v = vol2.calvol();
            System.out.println("Volume is : " + v);
        }
}
```

The output of the program would be :

```
Volume is : 500
Volume is : 500
```

Both the objects **vol** and **vol1** were initialised by the **Volume()** constructor when they were created. The constructor gave the same set of values to vol and vol2 and therefore the volume of both vol and vol2 will be the same.

Thus when you allocate an object with the following general form :

class-var = new *classname*();

when a constructor is not explicitly defined for a class, Java creates a default constructor. However, once you define your own constructor the default constructor is no longer used.

4.4.2 Parameterised constructors :

We saw that the above constructor initialised all the objects with the same values. But actually what is needed in practice is that we should be able to set different initial values. For this purpose, we make use of parameterised constructors and each object gets initialised with the set of values specified in the parameters to its constructor. The following modification in the above program will illustrate :

```
class Volume {
    int length;
    int breadth;
    int height;
    Volume(int l, int b, int h) {
        length = l;
        breadth = b;
        height = h;
    }
    int calvol() {          // method to compute volume
        return length * breadth * height;
    }
        }
class Demo {
    public static void main(String args[ ]) {
        int v;
        Volume vol = new Volume(10,5,10);
        Volume vol2 = new Volume(5,5,5);
        v = vol.calvol();           // call method to
                                    compute volume
        System.out.println("Volume is : " + v);
```

```
                v = vol2.calvol();
                System.out.println("Volume is : " + v);
            }
    }
```

The output of the program would be :

Volume is : 500

Volume is : 125

Remember that the parameters are passed to the **Volume()** constructor when new creates the object. Thus you pass values to the object vol when you create it as : Volume vol = new Volume(10,5,10).

4.4.3 this keyword

this is a keyword that can be used inside any method to refer to the current object i.e **this** is always a reference to the object on which the method was invoked. This is useful when a method needs to refer to the object that invoked it.

4.4.4 Instance Variable hiding

In Java, it is illegal to declare two local variables with the same name within the same or enclosing scopes. However, you can have local variables, including formal parameters to methods which overlap with the names of the class' instance variables. But when a local variable has the same name as the instance variable, then the local variable hides the instance variable. We can use the **this** keyword to refer directly to the object and thus resolve the name space collisions that might occur between the instance variables and the local variables. eg. we can modify Volume() as follows :

```
Volume(int length, int breadth, int height) {
        this.length = length;
        this.breadth = breadth;
        this.height = height;
}
```

Here the parameter names are also length, breadth and height, so we can

4.4 & 4.5 Check Your Progress.

1. Write True or False.

a) Constructors always have a return type.

b) Java handles deallocation automatically.

c) In Java, you can declare two local variables with the same name within the same or enclosing scopes.

d) It is possible that the finalize() method may not be executed at all.

make use of the keyword **this** to access the instance variables which have the same names.

4.5 GARBAGE COLLECTION

In languages like C++, dynamically allocated objects must be manually released by making use of the **delete** operator. Java however handles deallocation automatically. The technique which accomplishes this task is called *garbage collection*. When no references to an object exist, that object is assumed to be no longer needed and the memory occupied by that object can be reclaimed. There is no explicit need to destroy objects. However, remember that garbage collection only occurs sporadically during your program execution. It will not occur just because one or more objects exist which are not used anymore.

finalize() method :

Sometimes an object may be required to perform some specific action when it is destroyed. eg. if it is holding some non-Java resource like a file handle then such resources should be freed before the object is destroyed. For this purpose, Java provides a mechanism called **finalization**. With finalization, you can define specific actions that will occur when an object is just about to be destroyed. You can define a **finalize()** method to specify those actions that must be performed before an object is destroyed. The Java runtime mechanism calls that method whenever it is about to recycle an object of that class.

The general form of the method is :

```
protected void finalize() {
        finalization code
}
```

The **protected** keyword prevents access to **finalize()** by code defined outside its class. **finalize()** is called prior to garbage collection. It is not called when an object goes out of scope. Thus you have no way of knowing when (or even if) the method will be executed. Therefore your program must not depend upon **finalize()** for normal program operation. It must have other means of releasing system resources used by the object.

4.6 OVERLOADING

4.6.1 Overloading Methods

When two or more methods in the same class share the same name, as long as their parameter declarations are different the methods are said to bo *overloaded* and the process is referred to as *method overloading*. Java supports overloading. It is one of the ways that Java implements the concept of polymorphism.

When an overloaded method is invoked, Java uses the type and/or number of arguments as a guide to determine which version of the overloaded method to call. When Java encounters a call to an overloaded

method, it executes that version of the method whose parameters match the arguments used in the call.

Overloading allows related methods to be accessed by the use of a common name. Thus through the application of polymorphism, several names have been reduced to one. When methods are overloaded each method can perform any desired activity. There is no rule stating that overloaded methods must relate to one another. However, in practice you should only overload closely related operations.

The following example demonstrates overloading methods.

```
class Area {
                    void area(int i) {
                        int a = i * i;
                        System.out.println("Area of a square is :" + a);
                    }
                    void area(int i, int j) {
                    int a1 = i * j;
                    System.out.println("Area of a rectangle is :" + a1);
    }
    void area(float r) {
                    double a2 = 3.14 * r * r;
                    System.out.println("Area of a circle is :" + a2);
    }
}
class Over {
    public static void main(String args[]) {
                    Area A = new Area();
                    A.area(10);
                    A.area((float)7.2);
                    A.area(10,5);
    }
}
```

In the above example area() is overloaded thrice, one which takes one integer parameter, second takes two integer parameters and the third takes one float parameter. When you call a method, Java looks for a match between the arguments used to call the method and the parameter list of the method. Upon finding a match, the corresponding method is invoked. Study the output of the above program.

4.6.2 Overloading Constructors

It is also possible to overload constructors. The overloaded constructor is called based upon the parameters specified when **new** is executed at the time of creating objects. Let us write a program to demonstrate constructor overloading :

```
class Area {
    int length, breadth;
    Area(int i) {
        length = breadth = i;
    }
    Area(int i, int j) {
        length = i;
        breadth = j;
    }
    int area() {
        return length * breadth;
    }
}
class OverCons {
    public static void main(String args[]) {
        Area A = new Area(10,5);
        Area A1 = new Area(5);
        System.out.println("The area of square is :" +
    A1.area());
        System.out.println("The area of rectangle is :" +
    A.area());
    }
}
```

Here when the object A is intialised, the constructor with two parameters is used and for initialising object A1, the constructor with a single parameter is used.

4.6 Check Your Progress.

1. Write True or False.

a) It is not possible to overload constructors in Java.

b) Method overloading is one of the ways that Java implements the concept of polymorphism.

c) As a rule overloaded methods must relate to one another.

4.7 USING OBJECTS AS PARAMETERS

Objects can be passed as parameters to methods. One of the most common use of passing objects as parameters involves constructors. Many times it may be required to construct a new object that is initially the same as an existing object. To do this, you simply define a constructor that takes an object as a parameter. The example below illustrates :

```
Class Area {
    int length, breadth;
    Area(Area a) {
                        length = a.length;
                         breadth = a.breadth;
    }
    Area(int i, int j) {
                        length = i;
                        breadth = j;
    }
    int area() {
                        return length * breadth;
    }
}
class ObjParam {
    public static void main(String args[]) {
                        Area A = new Area(10,5);
                        Area A1 = new Area(A);
                        System.out.println("The area is :" + A1.area());
                        System.out.println("The area is :" + A.area());
    }
}
```

When the first object A is allocated, the constructor which takes two parameters is invoked. When creating object A1, object A is passed as parameter, and is used to initialise instance variables of object A1.

4.8 CALL BY VALUE AND CALL BY REFERENCE

The call by value method copies the value of an argument into the formal parameter of the subroutine. Therefore changes made to the parameters of the subroutine have no effect on the argument used to call it. In the call by reference method, a reference to an argument is passed to the parameter. This reference is then used in the subroutine to access the actual argument specified in the call.

This implies that any changes made to the parameter will affect the argument used to call the subroutine. Both call by value and call by reference are used in Java.

In Java, when you pass a simple type to a method, then it is passed by value. Thus changes made to parameters in the method, have no effect on the values of the argument. However, objects are passed as reference. We know that when we create a variable of a class type we are only creating a reference to an object. Therefore when you pass this reference to the method, the parameter that receives it will refer to the same object as that referred to by the argument. Changes made to objects within methods, affect the object that has been used as an argument. The following example will illustrate:

```
Class One {
        int a,b;
        One(int i, int j) {
                a = i;
                b = j;
        }
        void method1(One o) {
                o.a *= 2;
                o.b /= 2;
        }
}
class CallRef {
        public static void main(String args[ ]) {
                One O = new One(10,20);
System.out.println("Values of O.a and O.b before calling method :" + O.a + O.b);
                O.method1(O);
System.out.println("Values of O.a and O.b after calling method :" + O.a + O.b);
        }
}
```

The output of the program will be :

4.7 & 4.8 Check Your Progress.

1. Write True or False.

a) Objects can be passed as parameters to methods.

b) In Java, simple types are passed to a method by reference.

c) Changes made to objects within methods, do not affect the object that has been used as an argument.

Values of O.a and O.b before calling method : 10, 20

Values of O.a and O.b before calling method : 20, 10

Thus you can see that with call by reference changes made to parameters, have affected the arguments.

4.9 RETURNING OBJECTS

A method can return any type of data including the class types that you created. The following example will illustrate :

```
Class One {
                    int a,b;
                    One(int i, int j) {
                        a = i;
                        b = j;
                    }
                    One method1() {
                        One temp = new One(a,b)
                        temp.a *= 2;
                        temp.b /= 2;
                        return temp;
                    }
}
class RetObj {
                    public static void main(String args[ ]) {
                        One O = new One(10,20);
                        System.out.println("Values of a and b for
                    object O :" + O.a + O.b);
                        One O1 = O.method1();
          System.out.println("Values of  a and  b for Object O1 :" +
O1.a + O1.b);
                    }
}
```

The output of the program will be :

Values of O.a and O.b before calling method : 10, 20

Values of O.a and O.b before calling method : 20, 10

Remember that an object will not go out of scope because the method in which it was called terminates. It will continue to exist as long as there is a reference to it somewhere in the program. When there are no references, the object will be reclaimed next time garbage collection takes place.

4.10 RECURSION

Java supports recursion. A method that calls itself is a *recursive* method. We

have already studied recursion in our study of the C programming language. Recall that when writing recursive methods, you must have an **if** statement somewhere to force the method to return without the recursive call being executed. Otherwise the method will never return.

4.11 ACCESS SPECIFIERS

Encapsulation links the data with the code that manipulates the data. Encapsulation also provides another important attribute viz. access control. By introducing access control you can control what parts of the program can access the members of a class and thus prevent misuse. In this section we shall study access control as it applies to classes. An access specifier determines how a member can be accessed. The access specifiers in Java are : **public**, **private** and **protected**. **protected** applies only when inheritance is involved. Java also defines a default access level.

When a member of a class has the specifier **public** associated with it, then such a member can be accessed by any other code in your program. When a class member is specified as **private**, then that member can be accessed by the members of that class alone. Therefore the **main()** method is always declared as **public**, since it is called by code that is outside the program i.e. the Java run time system. When no access specifier is used, then by default the member of a class is **public** within its own package, but cannot be accessed outside the package. A **package** is a grouping of classes. We shall study more about packages later.

An access specifier precedes the rest of a member's type specification. eg.

```
private int i;
public double a;
```

are both examples of declaring variables with their access specifiers. This means that the access specifier begins the declaration statement of the member.

4.12 STATIC KEYWORD

In some situations, it may be necessary to define a class member which will be used independant of any object of the class. To create such a member we make use of the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created and without reference to any object. Both methods and variables can be declared **static**. Thus you can understand why the **main()** method is declared **static**. **main()** is declared **static** because it must be called before any object exists.

Instance variables declared as **static** are essentially global variables. When objects of its class are declared, no copy of static variable is made. Instead all instances of the class share the same static variable.

Methods declared **static** have the following restrictions :

- they can only call other static methods

- they must access only static data, it is illegal to refer to any instance variable inside of a static method.

- they cannot refer to **this** or **super** in any way.

Outside of the class in which they are created static methods and variables can be used independantly, without use of any object. The general form is :

classname-method();

where classname is the name of the class in which the static method is declared.

4.9 to 4.11 Check Your Progress.

1. Write True or False.

a) Java supports recursion.

b) A method can return data of the class type that you created.

2. Match the following.

	Column A	**Column B**
a)	private	i) applies when inheritance is involved
b)	public	ii) can be accessed by the members of that class alone
c)	protected	iii) can be accessed by any other code in your program

4.13 FINAL KEYWORD

When a variable is declared **final**, its contents cannot be modified. You must initialise a **final** variable at the time of its declaration. Usage of **final** is similar to **const** in C/C++. eg.

final PI = 3.141

final MAX = 100;

It is a common coding practice to choose all uppercase identifiers for variables declared as **final**.

A note about Arrays : In Java, arrays are implemented as objects. Arrays have a special attribute which is found in its **length** instance variable. The size of the array i.e. the number of elements that the array can hold, is found in the **length** instance variable. All arrays have this variable and it will always hold the size of the array. Remember that the value of length has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold. The following program demonstrates how to determine the size of the array using the **length** instance variable.

```
class Length {
    public static void main(String args[]) {
                    int a1 = new int[10];
                    int a2 = {2, 4, 6, 8, 10, 12};
                    System.out.println("length of a1 is " + a1.length);
```

```
        System.out.println("length of a2 is " + a2.length);
    }
}
```

The output of the program will be :

length of a1 is 10

length of a 2 is 6

4.12 & 4.13 Check Your Progress.

1. Answer the following.

a) Why is the main() method declared static?

_____

_____

b) What happens when a variable is declared final?

_____

_____

c) What does the length instance variable tell about an array?

_____

_____

4.14 NESTED AND INNER CLASSES

It is possible to define a class within another class; such classes are known as nested classes. If class B is defined within class A, then B is known to A, but not known outside A. A nested class can have access to the members of the class in which it is nested (including private members). However, the enclosing class does not have access to the members of the nested class.

There are two types of nested classes : **static** and **non static**. Static classes have the **static** modifier applied to them, which implies that they must access members of its enclosing class through an object and not directly. Due to this limitation, static nested classes are used rarely.

The most important type of nested class is the **inner** class which is a non static nested class. It has access to all the members of its outer class and can refer to them directly. Thus an **inner** class is fully within the scope of its enclosing class.

Let us study the following illustration of **inner** class.

```
Class Outer {
    int outer_i = 100;
    void test() {
```

```
                Inner inner = new Inner();
                inner.display();
            }
            class Inner {
                void display() {
                  System.out.println("Display member variable of
outer : " + outer_i);
                }
            }
    }
    class InnerOuter {
            public static void main(String args[]) {
                Outer outer = new Outer();
                outer.test();
            }
    }
```

The output of the above program is :

Display member variable of outer : 100

You can see from the above example, that the inner class named **Inner** is defined within the scope of the outer class named **Outer**. **Inner** directly access the variable outer_x. The **display()** method is defined in the inner class and the outer class creates and instance of the inner class to access the method **display()** as **inner.display()**. Remember that no code outside of class **Outer** can access the class **Inner**. Also remember that an inner class can be defined within any block scope. It is not necessary that an inner class be enclosed within the scope of a class only.

4.14 Check Your Progress.

1. Write True or False

a) A nested class cannot have access to the members of the class in which it is nested.

b) An inner class can be defined within any block scope.

4.15 INHERITANCE

4.15.1 Inheritance a Class

We have already studied that using inheritance we can create a general class that defines traits common to a set of related items. This class can then be inherited by other more specific classes. Each of these classes adds those features that are unique to it. A class that is inherited is called the **superclass**.

The class which inherits another class is called a **subclass**. Thus a subclass is a specialised version of a superclass, which inherits all the instance variables and methods defined by the superclass, at the same time adding its own unique elements.

In order to inherit a class we make use of the keyword **extends**. The general form of extending a class is :

class *subclass-name* extends *superclass-name* {

body of the class

}

The following example will demonstrate how to inherit a class.

```
Class A {
    int x, y;
    void display() {
        System.out.println("x and y " + x + y);
    }
}
class B extends A {
    int z;
    void showz() {
        System.out.println("z = " + z);
    }
    void add() {
        System.out.println("x+y+z= " + (x + y + z));
    }
}
class Inherits {
    public static void main(String args[]) {
        A a = new A();
        B b = new B();
        a.x = 10;
        a.y = 5;
        System.out.println("Contents of Superclass A :");
        a.display();      // invoke display on object a
        b.x = 20;         //Subclass can access public
                members of superclass
        b.y = 30;
        b.z = 10;
```

```
        System.out.println("Contents of Subclass ");
        b.display();            // invoke the display
                                   method on object b
        b.showz();
        System.out.println("Call add method from
        Subclass :");
        b.add();        // add method of subclass
    }
}
```

The output of the program will be :

```
Contents of Superclass A :
10              5
Contents of Subclass
x and y : 20            30
z = 10
x+y+z= 60
```

Here you can see that the subclass B includes all the members of its superclass A. Therefore it can access the variables x and y. Note that class A though a superclass of B, is completely indepedant stand alone class.

Remember that you can only specify one superclass for any subclass that you create. Inheritance of multiple superclasses is not possible in Java. (This differs from C++). However, a subclass can be a superclass for another subclass. Thus you can create a hierarchy of inheritance in which a subclass becomes a superclass of another superclass. No class can be a superclass of itself. A superclass can be used to create any number of subclasses. Also remember that a member which has been declared **private** is not accessible by any code outside its class, including the subclasses.

4.15.2 Super

Whenever a subclass needs to refer to its immediate superclass, it can do so by using the keyword **super**. **super** has two general forms. The first one calls the constructor of the superclass. The second one is used to access a member of the superclass that has been hidden by a member of a subclass.

In the first form of **super**, a subclass can call a constructor method defined by its superclass by using the following form :

super(*parameter-list*);

The *parameter-list* specifies any parameters needed by the constructor in the superclass. Remember that **super()** must always be the first statement executed inside a subclass' constructor. The following example will illustrate this use of **super**

```
class A {
        int i;
        int j;
        A(int a, int b)
        {
                i = a;
                j = b;
        }
        ...
        ...
}
class B extends A {
        int k;
        B(int p, int q, int r) {
                super(p,q);
                k = r;
        }
        ...
}
```

Here the constructor for subclass B is called with parameters p, q, r. We make use of **super()** with parameters p and q which initialise i and j. The class A no longer initialises these values itself. Remember that constructors can be overloaded, therefore **super()** can be called using any form defined by the superclass. The one that matches the arguments in number and type will be executed.

An important thing to note is that **super()** always refers to the superclass immediately above the calling class. The general form of **super** in the second method is:

super.*member*

member can be either a method or an instance variable. This form of **super** is applicable when members of the subclass hide the members of the superclass which have the same name in both classes. **super** allows access to the member of the superclass which has the same name as a member of the subclass.

eg. if a variable i is defined in both the superclass and the subclass, then using **super.i** in the subclass, you can refer to the variable i of the superclass.

4.15.3 Multilevel Hierarchy

You can build hierarchies which contain as many levels of

inheritance as you like. eg. if you have three classes A, B and C, then C can be a subclass of B, where B is a subclass of A. In this case, C inherits all the properties of A and B. Remember that all classes can be written in separate files and compiled separately.

In a class hierarchy, constructors are called in their order of derivation, from superclass to subclass. Also remember that **super()** must be the first statement executed in the subclass constructor.

4.15.1 to 4.15.3 Check Your Progress.

1. Fill in the blanks.

a) A class that is inherited is called the .....

b) We make use of the keyword ..... to inherit a class.

c) In a class ..... constructors are called in their order of derivation.

2. What are the two general forms of super?

.....

.....

4.15.4 Method Overriding

It may so happen that a method in a subclass has the same name and type as that of the superclass. In this situation, the method in the subclass is said to override the method in the superclass and the method defined by the superclass will be hidden. Overridden methods in Java are similar to virtual functions in C++. Let us illustrate method overriding with the following example :

```
class A {
                    int i, j;
                    A(int a, int b) {
                        i = a;
                        j = b;
                    }
                    void display() {
                        System.out.println( “i = “ + i + “j = “ + j);
                    }
}
class B extends A {
                    int k;
                    B(int a, int b, int c) {
                        super(a,b);
                        k = c;
                    }
                    void display() {
```

```
                System.out.println("k = " + k);
        }
}
class Override {
        public static void main(String args[ ]) {
                B b = new B(4, 10, 12);
                b.display();
        }
}
```

The output of this program will be :

k = 12

Here, when you invoke the method display() on an object of type B, the version of display() which has been defined in B will be used. This means the version of B will override the version declared in A.

If you wish to access the method of the superclass, you can do so by using the keyword **super**. Modify the above example as shown where you will precede the display() method of B with the word **super**.

```
class B extends A {
        int k;
        B(int a, int b, int c) {
                super(a, b);
                k = c;
        }
        void display() {
                super.display()  // call display of A
                System.out.println("k = " + k);
        }
}
```

The output of the modified program will be :

i = 4 j = 10

k = 12

super.display() calls the superclass version of display().

Remember that method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not then the methods are simply overloaded.

4.15.5 Dynamic Method Dispatch

This is the mechanism by which a call to an overridden function is

resolved at run time rather than compile time. This is how Java implements run time polymorphism. When a method that is overridden is called through a superclass reference, Java determines which version of that method to execute, based upon the object being referred to at the time the call occurs. Thus, this determination is made at run time. Thus, it is the type of the object being referred to and not the type of the reference variable, which determines the particular version of the method to be executed.

4.15.4 to 4.15.5 Check Your Progress.

1. Answer the following.

a) When does a method in the subclass override a method in the superclass?

_____

_____

b) What is meant by dynamic method dispatch?

_____

_____

4.15.6 Using Abstract Classes

Sometimes it is necessary to create a superclass that only defines a generalised form. This generalised form will be shared by all its subclasses and each subclass will fill in its own details. Such a class determines the nature of the methods that the subclass must implement. You can require that a certain method be overridden by subclasses by specifying the **abstract** modifier. The general form to declare an **abstract** method is :

abstract *type name*(*parameter-list*);

Note that no method body is present. Any class which contains one or more **abstract** methods must also be declared **abstract**. There can be no object of an **abstract** class. This means that an abstract class cannot be instantiated with the **new** operator. You cannot declare **abstract** constructors or **abstract static** methods. Any subclass of an abstract class must either implement all of the abstract methods of the superclass, or itself be declared **abstract**. A class is declared **abstract** by writing the **abstract** keyword before the class keyword at the beginning of the class declaration.

The following example will illustrate :

```
abstract class A {
                    abstract void meth();
                    void meth1() {
                        System.out.println("Method in abstract class A");
                    }
}
```

```
class B extends A {
        void meth() {
                System.out.println("Implement the abstract method of
superclass A");
        }
}
class Abstracts {
        public static void main(String args[]) {
                B b = new B();
                b.meth ();
                b.meth1();
        }
}
```

The output of the program would be :

```
Implement the abstract method of superclass A
Method in abstract class A
```

Remember that you cannot create an instance of class A. The method **meth()** which is **abstract** in class A is provided a body in the subclass B.

4.15.7 final

We have already seen one use of **final** to create a named constant. Let us study two more uses of final in this section.

Using final to prevent overriding : To disallow a method from being overridden, specify the **final** modifier at the start of the declaration of the method. This means that methods declared as **final** cannot be overridden. Once a method has been declared **final** in a class, its subclass cannot override it. There can be no method with the same name and type in the subclass. A compile time error will occur in such situation.

Using final to prevent inheritance : If you want to prevent a class from being inherited you precede the class declaration with the word **final**. When a class it declared **final**, all of its methods are also implicitly declared **final**. It is illegal to declare a class as both **abstract** and **final**, because an abstract class is incomplete by itself and requires the subclass to provide implementation to its methods.

4.15.6 to 4.15.7 Check Your Progress.

1. Write True or False.

a) An abstract class can be instantiated with the new operator.

b) A class can be declared as both abstract and final.

2. List two uses of final.

_____

_____

4.16 THE OBEJCT CLASS

Object class is a special class in Java. All the other classes are subclasses of **Object**. **Object** is the superclass of all the classes. This means that a reference variable of type **Object** can refer to an object of any other class. Also remember that since arrays are implemented as classes, a variable of type **Object** can also refer to any array.

The **Object** class defines the following methods :

Method	**Purpose**
Object clone()	Creates a new object that is the same as the object being cloned
boolean Equals(Object object)	Determines whether one object is equal to another
void finalize()	called before an unusual object is recycled
Class getClass()	Obtains the class of an object at run time
int hashCode()	Returns the hash code associated with the invoking object
void notify()	Resumes execution of a thread waiting on the invoking object
void noifyAll()	Resumes execution of all threads waiting on the invoking object
String toString()	Returns a string that describes the object
void wait()	Waits on another thread of execution

Among these, the methods **getClass()**, **notify()**, **notifyAll()** are declared as **final**. Other methods may be overridden. We shall see all these methods in detail as we progress through the study notes.

4.16 Check Your Progress.

1. Fill in the blanks.

a) The ..... class is the superclass of all the classes.

b) The getClass() method is declared as ..... in object.

4.17 SUMMARY

A class is a logical construct which defines the shape and nature of an object.

The variables defined within a class are called instance variables. The code is contained within methods. The variables and the methods within a class are called as members of the class. The new operator dynamically allocates memory for an object and returns a reference to it.

Constructors : Java allows objects to initialise themselves when they are created. This automatic initialisation is performed through the use of a constructor. A constructor has the same name as the name of the class in which it resides. Constructors do not have a return type.

Garbage collection : Java handles deallocation automatically. The technique which accomplishes this task is called garbage collection.

finalize() method : You can define a finalize() method to specify those actions that must be performed before an object is destroyed.

The call by value method copies the value of an argument into the formal parameter of the subroutine. In the call by reference method, a reference to an argument is passed to the parameter.

Class declared with in a class called nested class.

There are two types of nested classes : static and non static.

4.18 CHECK YOUR PROGRESS - ANSWERS

4.1 & 4.2

1. a) True
b) False
c) True
d) False

2. a) new
b) class, instance

4.3

1. a) void
b) argument
c) parameter

4.4 & 4.5

1. a) False
b) True
c) False
d) True

4.6

1. a) False

b) True

c) False

4.7 & 4.8

1. a) True

b) False

c) False

4.9 to 4.11

1. a) True

b) True

2. a) ii)

b) iii)

c) (i)

4.12 & 4.13

1. a) main() method is declared as static because when a member is declared static, it can be accessed before any objects of its class are created and main() has to be called before any object exists.

b) When a variable is declared final its contents cannot be modified.

c) The size of the array i.e. the number of elements that the array can hold is found in the length instance variable of an array.

4.14

1. a) False

b) True

4.15.1 to 4.15.3

1. a) superclass

b) extends

c) hierarchy

2. The two general forms of super are :

super(*parameter-list*);

super.member

4.15.4 & 4.15.5

1. a) When a method in a subclass has the same name and type as that of the superclass the method in the subclass overrides the method in the superclass and the method defined by the superclass will be hidden.

b) Dynamic method dispatch is the mechanism by which Java implements run time polymorphism whereby a call to an overridden function is resolved at run time rather than compile time.

4.15.6 & 4.15.7

1. a) False

b) False

2. Two uses of final : to prevent overriding, to prevent inheritance.

4.16

1. a) object

b) final

4.19 QUESTIONS FOR SELF - STUDY

1. What is a class? Describe the general form of declaring a class.
2. What are methods? What is the general form of declaring a method?
3. **Write short notes on :**
   a) Constructors
   b) Parameterised methods
   c) Call by value and call by reference
   d) Access specifiers in Java
   e) The final keyword
   f) Method overriding
   g) Abstract class
4. What is overloading?
5. What do you understand by garbage collection? Describe in brief the finalize() method.
6. Describe the use of the static keyword in Java.
7. Write in brief about nested and inner classes.
8. Describe what is meant by inheritance. What do you understand by multilevel hierarchy?
9. What is meant by dynamic method dispatch?
10. What is the Object class? Write any two methods defined in the Object class.

4.20 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

❑ ❑ ❑

Notes

Notes