UNIT 4 : CLASS IN JAVA

UNIT STRUCTURE

4.1 Learning Objectives
4.2 Introduction
4.3 Class Fundamentals
    4.3.1 Declaring and Defining a Class
    4.3.2 Adding Methods to a Class
    4.3.3 Creating Objects
    4.3.4 Accessing Class Members
4.4 Constructor
    4.4.1 Default and Copy Constructor
    4.4.2 Constructor Overloading
4.5 Passing Arguments to Methods
4.6 Recursive Method
4.7 Modifiers
4.8 Let Us Sum Up
4.9 Further Readings
4.10 Answers to Check Your Progress
4.11 Model Questions

4.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- write your own classes in Java
- access class members in Java
- declare objects and pass arguments to methods
- use recursive methods in Java
- describe and use constructors, copy constructor etc.

4.2 INTRODUCTION

We have already learnt the concepts of tokens, data types, variables, constants, decision and control statements, operators etc. along with their types as they relate to Java language.

Anything we wish to represent in a Java program must be encapsulated in a class. Classes are the building blocks of a Java application. In this unit, we will explore the object-oriented aspects of Java. We will learn about the concept of classes in Java, how to access class members, how to create instance of classses etc. Some other important concepts like super classes, contructors etc. are also covered in this unit. At the end of this unit, we will acquiant you with the concept of recursion. In the next unit, we will discuss about arrays, strings and vectors.

4.3 CLASS FUNDAMENTALS

Classes provide a convenient way for grouping data items and functions. A class can contain basically data and methods or functions. In case of Java, the data items are termed as ***fields*** and the functions are termed as ***methods***.

Class serves as a blueprint for making class *instances*, which are runtime objects that implement the class structure. Thus, an ***object*** is defined to be an instance of a class. An object consists of data called ***attributes*** and methods called ***behaviors***. An attribute is a feature of the object, something the object "has." A behavior is something the object "does".

4.3.1 Declaring and Defining a Class

A class in Java is declared using the ***class*** keyword. A source code file in Java can contain exactly one public class, and the name of the file must match the name of the public class with a ***. java*** extension.

We can declare more than one class in a . java file, but at

most one of the classes can be declared as *public*. The name of the source code file must match the name of the public class. If there are no public classes in the source code file, the name of the file is arbitrary. The general form of a **class** definition is as follows:

```
class classname
    {
        type instanceVariable1;
        type instanceVariable2;
        ..............
        ..............
        type instanceVariableN;
        type methodname1(parameterList)
        {
            // body of method
        }
        type methodname2(parameterList)
        {
            // body of method
        }
        ..............
        ..............
        type methodnameN(parameterList)
        {
            // body of method
        }
    }
```

Here, we can see that, there is no semicolon after closing brace of class in Java. But in case of C++ language, class definition ends with a semicolon. The data or variables, defined within a class are called ***fields*** or ***instance variables*** as each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique

from the data for another. The ***methods*** are used for manipulating the instance variables(data items) contained in the class. The methods and variables defined within a class are collectively called *members* of the class. A class may contain three kinds of members: *fields*, *methods* and *constructors*. Constructor specify how the objects are to be created.

Java classes do not need to have a ***main( )*** method. The general form of a class does not specify a ***main( )*** method. We only specify one if that class is the starting point for our program. The fields and methods of a class appear within the curly brackets of the class declaration.

A class may not contain anything inside the curly brace { and }. Thus we can have empty classes in Java. An empty class cannot do anything as it does not contain any properties. But it is possible to create objects of such classes. The structure of an empty class is as follows:

```
class classname
{
}
```

Let us consider another class *Triangle* which has two float type instance variables *base* and *height*.

```
class Triangle
{
        float base;          //instance variable base
        float height;        //instance variable height
}
```

Until object of *Triangle* class is created, there is no storage space that has been created in the memory for these two variables *base* and *height*. When an object of *Triangle* class is created, memory will be allocated for each of these two fields.

4.3.2 Adding Methods to a Class

A ***method*** is a sequence of declarations and executable statements encapsulated together like independent mini program. For the manipulation of data fields inside a class we have to add methods into the class. Method contains local variable declarations and other Java statements that are executed when the method is invoked.

- The name of the method, which can be any valid identifier
- Return value (i.e., the type of the value the method returns)
- List of parameters, which appears within parentheses
- Definition of the method (i.e., the body of the method)

A Java application must contain a main() method whose syntax looks like this :

```
public static void main(String args[ ])
{
        // body;
}
```

The meaning of the keywords *public*, *static* etc. are already outlined in *Unit 1*. *void* indicates that the *main()* method does not return any value.

In Java, the *definition* (often referred to as the *body*) of a method must appear within the curly brackets that follow the method declaration. For example, let us add two methods in the *Triangle* class.

```
class Triangle
{
        float base, height;
        void readData(float b, float h)    //definition of method
        {
                base=b;
                height=h;
        }
        float findArea( )          //definition of another method
```

```
        {
                float area = 0.5*(base *height);
                return area;
        }
}
```

In the above code, the *Triangle* class demonstrates how to add methods to a class. In the above class there are two methods, ***readData( )*** and ***findArea( )***. The method ***readData()*** has a return type of ***void*** because it does not return any value. We pass two float type values to the method which are then assigned to the instance variables *base* and *height*. The method ***findArea()*** calculates area of the triangle and returns the result. Here, we can directly use ***base*** and ***height*** inside the method *readData()* and *findArea()*.

Let us consider another method ***checkPrime()*** for the illustration of method. The following program tests a boolean method that checks its arguments for primality. The main() method prints those integers for which the checkPrime() returns true:

*//**Program 4.1 :*** checkPrime.java

```
class checkPrime
{
      public static void main(String [ ] args)
      {
             for(int i=0;i<50;i++)
             if(isPrime(i))
                    System.out.print(i+ " ");
             System.out.println();
      }
      static boolean isPrime(int n)      //static method
      {
             if(n<2)
                    return false;
             if(n==2)
```

```
                return true;
            if(n%2==0)
                return false;
            for(int j=3;j<=Math.sqrt(n);j=j+2)
                if(n%j==0)
            return false;
                return true;
        }
}
```

In the program, *Math* is a predefined class in Java and *sqrt()* is a predefined method. The output will give the prime numbers from 2 to 47 as follows:

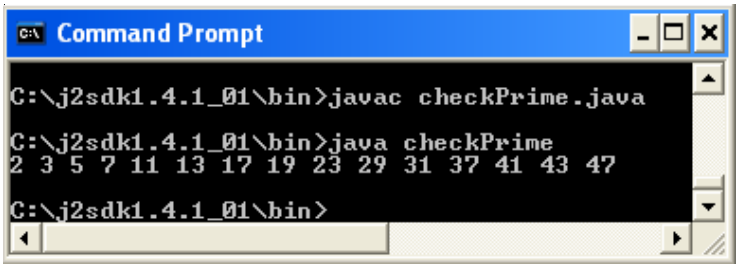


Figure 4.1: Output of program 4.1

4.3.3 Creating Objects

It is important to remember that a class declaration only creates a template; it does not create an actual object. Java allocates storage for an object when we create it with the ***new*** operator. Creating an object is also referred to as instanting an object. The new operator creates an object of the specified class and returns a reference to that object.

Thus, the preceding code does not cause any objects of type ***Triangle*** to come into existence. To create a **Triangle** object, we will use a statement like the following:

Triangle t1 = new Triangle( ); //t1 is an object of Triangle class

Here, the *Triangle( )* is the default constructor of the class. The above statement can also be written by splitting it into two

statements. This can be written as follows :

Triangle t1; // declares variable t1 to hold the object reference

t1 = new Triangle(); //assigns the object reference to the
//variable t1

Again, each time we create an instance of a class, we are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Triangle** object will contain its own copies of the instance variables **base** and **height**.

4.3.4 Accessing Class Members

Now we can say that when an object is created, each object contains its own set of data and methods. After creating objects of a class, we should assign values to these data member in order to use them in our program. Data and methods cannot be accessible directly outside the class. For this, we have to use concerned object and the *dot(.)* operator. The dot operator links the name of the object with the name of an data member or method. The syntax of accessing class member is as follows:

objectName.instanceVariableName ;

objectName.methodName(parameterList);

For example, suppose we have created **t1** object with the following statement

Triangle t1 = new Triangle( );

Now to assign the value 20.5 to the **base** variable of **t1**, we would use the following statement:

t1.base = 20.5;

This statement tells the compiler to assign the copy of *base* that is contained within the t1 object the value of 20.5. Here is a complete program that uses the **Triangle** class.

//Program 4.2 : Area.java

```
class Triangle
```

```
{
    double base, height;
    void readData(double b, double h)  //definition of method
    {
        base=b;
        height=h;
    }
    double findArea( )      //definition of another method
    {
        double a = 0.5*(base *height);
        return a;
    }
}
class Area                  //class with the main() method
{
    public static void main(String args[ ])
    {
        double area1, area2;
        Triangle t1=new Triangle();
        Triangle t2=new Triangle();
        t1.base=20.5;
        t1.height=15.5;
        area1= t1.base  * t1.height;
        t2.readData(30.0,15.0);
        area2=t2.findArea();
        System.out.println("Area of the first triangle is =  "+area1);
        System.out.println("\nArea of the second triangle is =  "+area2);
    }
}
```

The *Triangle* class has no *main()* method. So it cannot be executed as Java program. We need a separate class that does have a *main()* method. In the above program, the class *Area* contains the *main()* method. The name of the file where we write the program should

have the name ***Area.java*** . If we execute the program with the above data, the output will look like:

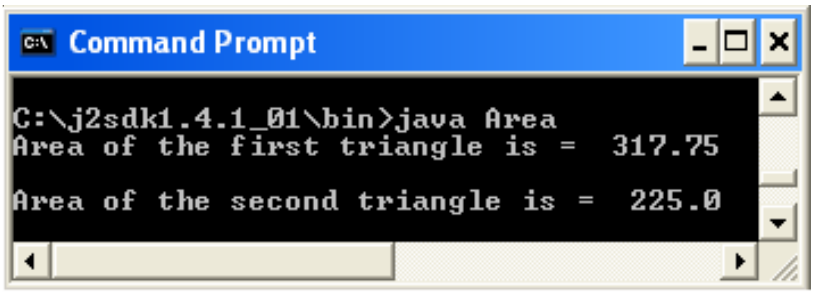


Figure 4.2 : Output of program 4.2

If we write these two classes ***Triangle*** and ***Area*** into two separate files namely ***Triangle.java*** and ***Area.java***, then these two files should be saved in the same folder in the computer. To compile ***Area.java*** program, we must first compile the ***Triangle*** class as:

javac Triangle.java

This will create the bytecode file ***Triangle.class*** in the same folder. Now, we can compile and execute the ***Area.java*** file with the following statements: **javac Area.java**

java Area

If we look at the contents of the folder, we will find four files: the source code and bytecode for Triangle and the source code and bytecode for Area. The statements

Triangle t1=new Triangle();

Triangle t2=new Triangle();

will create two Triangle objects t1 and t2 in memory, which reserves memory for two base fields, two height fields, two readData() methods and two findArea() methods. They are distinguished by which reference we use. Each of the fields is initialized in both objects using the dot operator with the following statements:

t1.base=20.5;

t1.height=15.5;

The Triangle reference t1 points to the Triangle object whose base is 20.5 and height is 15.5. The base and height field of t2 object

is initialized to 30.0 and 15.0 with the statement :

t2.readData(30.0,15.0);

To compute the area of the triangle t1, we can use any one of the following statements:

area1= t1.base * t1.height;

area1=t1.findArea();

Similarly, for t2, we can use

area2=t2.base * t2.height;

area2=t2.findArea();

4.4 CONSTRUCTOR

It would be simpler and more concise to initialize an object when it is created. Java supports constructors that enable an object to initialize itself when it is created. This section provides a brief introduction about the **constructor** and how constructors are overloaded in Java.

Constructor is always called by ***new*** operator. Constructors are declared just like a method, except that the constructor does not have any return type. The name of the constructor must be the same as the class name where it is declared. It is called when a new class instance is created, which gives the class an opportunity to set up the object for use. Constructors, like other methods, can accept arguments and can be overloaded but they cannot be inherited. For example, let us consider the same *Triangle* class. We can replace the *readData()* method by a constructor. This can be accomplished as follows:

*//**Program 4.3 :*** TriangleArea.java

```
class Triangle
{
   double base, height;
   Triangle(double b, double h)   //constructor with two arguments
      {
            base=b;
            height=h;
```

```
    }
  double findArea( ) //definition of method findArea()
  {
    double area = 0.5*(base *height);
    return area;
  }
}
class TriangleArea   //class with the main() method
{
  public static void main(String args[ ])
  {
    Triangle t1=new Triangle(20.5, 15.5); // call of constructor
      double area1= t1.findArea();
     System.out.println("Area of the triangle is =  "+area1);
  }
}
```

The output of the program will be:

Area of the triangle is 317.75

The *Triangle* class contains a single constructor. We can recognize a constructor because its declaration uses the same name as the class and it has no return type. The constructor in the Triangle class takes two arguments. The statement Triangle t1= new Triangle(20.5, 15.5); provides 20.5 and 15.5 as values for those arguments.

4.4.1 Default and Copy Constructor

There are two special kinds of constrcutors that a class may have: a *default constructor* and a *copy constructor*.

The ***default constructor*** is what gets called whenever we create an object by calling its constructor with no arguments. It initializes all instance variables to default value(zero for numeric types, null for object references, and false for booleans). There may be only one default constructor in a class.

The ***copy constructor*** is a constructor whose only parameter is a reference to an object of the same class to which the constructor belongs. It is called copy constructor as it is used to duplicate an existing object of the class.

Let us consider another example to illustrate these two constructors:

//Program 4.4 : CircleArea.java

```
class Circle
{
     double radius;
     Circle()            //default constructor
     {
          radius =10.0;
     }
     Circle(Circle c)    //copy constructor
     {
          radius = c.radius;
     }
     double findArea( )  //for calculation of area
     {
          double area = 3.14 *radius *radius;
          return area;
     }
}
class CircleArea         //class with the main() method
{
     public static void main(String args[ ])
     {
     double p_area,q_area;
     Circle p = new Circle();   //invokes the default constructor
      Circle q = new Circle(p);    //invokes copy constructor
          p_area= p.findArea();
          q_area =q.findArea();
```

```
        System.out.println("Area of circle p =  "+p_area);
        System.out.println("\nArea of circle q =  "+q_area);
        }
}
```

The output of the above program will be like this:

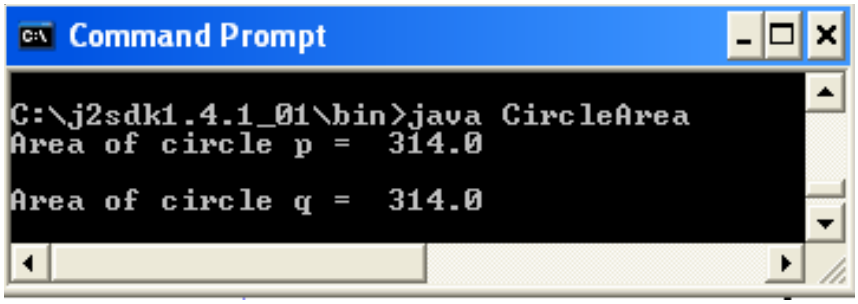


Figure 4.3: Output program for Copy constructor

The statement **Circle p = new Circle();** invokes the *default constructor* and it creates a Circle object **p** with radius 10.0. The statement Circle **q = new Circle(p);** invokes the *copy constructor* and it also creates another Circle object q with the same radius value 10.0. The object p and q are two separate but equal objects. In the output we can see that the area of both the circles are same. All fields in a class will automatically be initialized with their type's default values unless the constructor explitcitly uses other values.

4.4.2 Constructor Overloading

Overloading of constructors means multiple constructors in a single class. *Program 4.5* is an example of constructor overloading as there are two constructors in that program. Constructor can be overloaded provided they should have different arguments because Java compiler differentiates constructors on the basis of arguments passed in the constructor.

Let us consider the following *Rectangle* class for the demonstration of constructor overloading. After you going through it, the concept of constructor overloading will be more clear to you. In the example (Program 4.5), we have written four constructors each

having different arguments types.

//Program 4.5 : ConstructorOverloading.java

```
class Rectangle
{
    int l, b;    // for int type length and breadth
    float p, q; // for float type length and breadth
    Rectangle(int x, int y)   // constructor 1
        {
            l = x;
            b = y;
        }
        int first() {  //method
            return(l * b);
        }
    Rectangle(int x) { //constructor 2
            l = x;
            b = x;
        }
        int second() { //method
            return(l * b);
        }
    Rectangle(float x) {      //constructor 3
            p = x;
            q = x;
        }
        float third()
        {
            return(p * q);
        }
       Rectangle(float x, float y) { //constructor 3
        p = x;
        q = y;
    }
```

```
    float fourth()  {
        return(p * q);
    }
}
class ConstructorOverloading  {
    public static void main(String args[ ])    {
Rectangle r1=new Rectangle(2,4);
int area1=r1.first();
System.out.println(" Area in first constructor : " + area1);
Rectangle r2=new Rectangle(5);
int area2=r2.second();
System.out.println("\nArea in second constructor: " + area2);
Rectangle r3=new Rectangle(2.0f);
float area3=r3.third();
System.out.println("\nArea in third constructor: " + area3);
Rectangle r4=new Rectangle(3.0f,2.0f);
float area4=r4.fourth();
System.out.println("\nArea in fourth constructor: " + area4);
    }
}
```

The output will be like this:

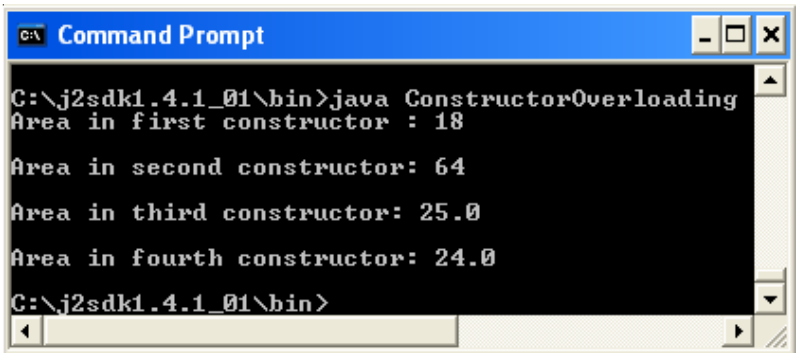


Figure 4.4: Output of Constructor Overloading Example

Constructor can also invoke other constructors using ***this*** and ***super*** keywords. We will discuss the first case here this keyword. A constructor can invoke another, overloaded constructor in its class using the reference ***this()*** with appropriate arguments to select the

desired constructor. If a constructor calls another constructor, it must do so as its first statement: ***this()*** calls another constructor in same class. The following program segment shows the use of 'this' keyword.

```
class Triangle
{
    double base, height;
    Triangle(double b,double h) {
        base = b;
        height=h;
    }
    Triangle(double b)
    {
        this(b, 20.50 );
    }
}
```

Here, class Triangle have two contructors. The first, accepts arguments specifying the triangles's base and height. The second constructor takes just the base as an argument and call the first constructor with a default value 20.50 for height. The advantage of this approach is that a single constructor may call other constructors. It must appear as the first statement in the second constructor. It should be remembered that we can invoke a second constructor only as the first statement of another constructor.

4.5 PASSING ARGUMENTS TO METHODS

In Java, we can pass argument of any valid Java data type into a method similar to functions in other programming languages like C, C++ etc. This includes simple data types such as characters, integers, floats, doubles and boolean as well as derived types such as arrays, objects etc. Arguments provide information to the method from outside the scope of the method. A method in Java always specifies a return type. The returned value can be a primitive type, a reference type, or *void* , which indicates no return

value.

The declaration for a method or a constructor declares the number and the type of the arguments for that method or constructor.

For example, let us consider the following method that computes the area of a rectangle:

```
int computeArea(int width, int height)      //method header
{
      int area;       // area is a local variable
      area = width * height;
      return area;
}
```

Here, the first *int* indicates that the method returns an integer. The name of the method is "computeArea" and it has two integer parameters- *length* and *breadth*. The *body* of the method starts with the left brace, "{" and end with the right brace, "}". The method may be called using the following syntax:

```
int a= r1.computeArea(16,8);      //method call
```

- **Formal and Actual parameter :** The parameter used to define a method is called formal parameter. In the above example, *width* and *height* are the formal parameters. We use the term *actual parameters* to refer to variables in the method call. They are called "actual" because they determine the actual values that are sent to the method.

  Let us consider what happens when we pass arguments to a method. In Java, all primitive data types (e.g., int, char, float) are *passed by value*. The reference types (i.e., any kind of object, including arrays and strings) are used through references. An important distinction is that the references themselves (the pointers to these objects) are actually primitive types and are passed by value too.

- **Pass-by-Value :** *Pass-by-value* means that when we call a method, a copy of the *value* of each actual parameter is passed to the method. We can change that copy inside the method, but this will have no

effect on the actual parameter. Unlike many other languages, Java has no mechanism for changing the value of an actual parameter. Java has eight primitive data types. Let us consider the following example for the demonstration of *pass-by-value* :

//Program 4.6 : PassPrimitiveByValue.java

```
class PassPrimitiveByValue
{
     public static void main(String[ ] args)
     {
        int p = 3;
        System.out.println("Before calling passValue, p = " + p);
         passValue(p);    //call passValue() with p as argument
        System.out.println("\nAfter calling passValue, p = " + p);
     }
     public static void passValue(int p)//passValue() definition to
     {                    //change the value of parameter
          p = 10;
     }
}
```

When we run this program, the output will be :

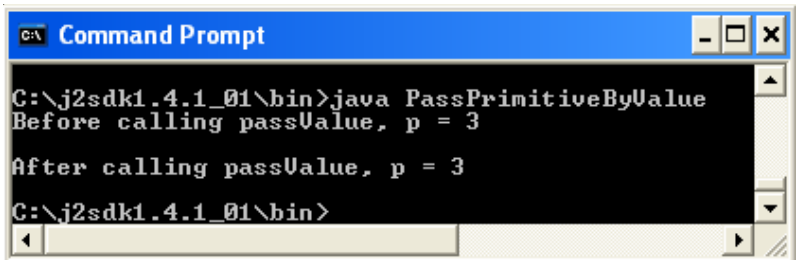


Figure 4.5: Output of example for pass by value

Here, we can see that the outputs are same before and after calling the method ***passValue()***. When Java calls a method, it makes a copy of its actual parameters' values and sends the copies to the method where they become the values of the formal parameters. Then when the method returns, those copies are discarded and the actual parameters have remained unchanged.

Passing variables by value affords the programmer some safety. Methods cannot unintentionally modify a variable that is outside of its scope. However, we often want a method to be able to modify one or more of its arguments. In the ***passValue()*** method, the caller wants the method to change the value through its arguments. However, the method cannot modify its arguments, and, furthermore, a method can only return one value through its return value. So, it is necessary to learn how a method can return more than one value, or have an effect (modify some value) outside of its scope.

To allow a method to modify a argument, we must pass in an object. Objects in Java are also passed by value; however, the value of an object is a reference. So, the effect is that the object is ***passed in by reference***. When passing an argument by reference, the method gets a reference to the object. A reference to an object is the address of the object in memory. Now, the local variable within the method is referring to the same memory location as the variable within the caller.

For example, if a parameter to a method is an object reference. We can manipulate the object in any way, but we cannot make the reference refer to a different object.

//Program 4.7 : Record.java (demonstration of passing object to method)

```
class  Record
{
   int  roll;
   String  name;
   static void tryObject(Record r)//parameter is an object
   {                              //reference
      r.roll = 1;
      r.name = "Anuj";
   }
   public static void main(String [] args)
   {
      Record obj = new Record(); //object obj of Record class
      obj.roll = 2;
```

```
        obj.name = "Rahul";
        System.out.println("Before calling tryObject(),the record is:"+
            obj.name + "  " + obj.roll);
        tryObject(obj);          //method call
        System.out.println("After calling tryObject(), the record is: " +
            obj.name + "  " + obj.roll);
    }
}
```

In the output, we observe different records before and after calling the method.



Figure 4.6 : Output of program 4.7

The first print statement displays "**Rahul 2**". The second print statement displays "**Anuj 1**". Thus the object has been changed in this case.

The reference to *obj* is the parameter to the method, so the method cannot be used to change that reference; i.e., it cannot make *obj* reference a different Record. But the method can use the reference to perform any allowed operation on the Record that it *already* references.

It is often not good programming style to change the values of instance variables outside an object. Normally, the object would have a method to set the values of its instance variables.

4.6 RECURSIVE METHOD

Java supports recursion. Recursion is the process of calling a method by itself. A method that calls itself is said to be ***recursive method***.

One of the suitable examples of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N, for example, 4 factorial is 1×2×3×4, or 24. Here is how a factorial can be computed using a recursive method.

//Program 4.8 : findFactorial.java

```
class findFactorial
{
   public static void main(String [] args)
   {
      for(int i=1;i<=10;i++)
      System.out.println("Factorial("+ i +") =" + fact(i)); //method call
   }
   static long fact(int n)//method definition
   {
      if(n<2)
         return 1;
      return n*fact(n-1);  //Recursive call
   }
}
```

The output will display the factorial of 1 to 10 as follows:

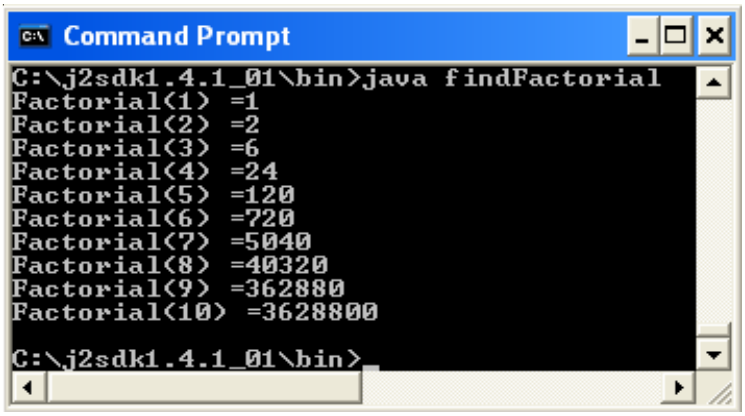


Figure 4.7: Demostration of recursion

When **fact()** is called with an argument of 1, the function returns 1; otherwise it returns the product of **n* fact(n-1)**. To evaluate this expression, **fact()** is called **n-1** times.



CHECK YOUR PROGRESS

Q1. Fill in the blanks :

i) In Java, data items are called _____.

ii) An _____ is defined to be the instance of a class.

iii) The extention of java file name should be _____.

iv) Class containg the ***main( )*** method shold have the _____ name with the . file name with .java extension.

v) Java allocates storage for an object with the _____ operator.

vi) The _____ operator links the name of the object with the name of an instance variable or method.

vii) Contructors have no _____.

viii) _____ calls another constructor in same class.

ix) In Java, all primitive data types are _____ to a method.

x) _____ is the process of defining something in terms of itself

xi) Objects in Java are also passed _____.

xii) A copy of the value of each _____ parameter is passed to the method in case of pass by value.

xiii) A class can have only _____ default constructor.

xiv) A _____ constructor is a constructor that replicates an existing object.

xv) Constructors are used to _____ an object of a class.

Q2. Write the method definition that implements the power function:

static double Power (double x, int n)

The method should return the value of x raised to the power n. For example Power(2.0,-3) would return 2^{-3}=0.125.

Q3. What is a constructor signature?

4.8MODIFIERS

Modifiers are keywords that are used to those definitions to change meaning. To use a modifier, we include its keyword in the definition of a class, method, or variable etc. The Java language has a wide variety of modifiers as listed in Table 4.1.

Some of the modifiers listed in the Table 4.1 are used to set access levels in the declaration of classes, fields, methods, constrctors and local variables etc. These are termed as ***access control modifiers*** which include :

private (Visible to the class only)

public (Visible to all)

protected (Visible to the package and all subclasses).

If none of these three is specified, i.e., the default, then the entity (class, field, constructor or method) has *package access*, which means that it can be accessed from any class in the same package.

- ***private :*** Private access modifier is the most restrictive access level. Variables, methods and constructors that are declared private can only be accessed within the declared class itself. Class and interfaces cannot be private.
- ***public :*** A class, method, constructor, interface etc. declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any other class. However, if the public class we are trying to access is in a different package, then the public class still need to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses. For example, The *main()* method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class. Thus we write:

```
public static void main(String[ ] arguments)
{
    // statements
}
```

- ***protected :*** Variables, methods and constructors which are declared *protected* in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

  To achieve another functionality Java provides a number of ***non-access modifiers*** which include *static, final, abstract, synchronized, volatile.*

- ***static :*** The modifier *static i*s used to specify that a method is a class method. Without it, the method is an instant method. An *instance method* is a method that can be invoked only when it is bound to an object of the class. A class method (also called static method) is a method that is invoked without being bound to any specific object of the class.

- ***final :*** The modifier *final* has differnent role depending upon which kind of entity it modifies. If it modifies a class, it indicates that class cannot have subclasses. If it modifies a field or local variable, it means that the variable must be initialized and cannot be changed (i.e., it is a constant).

We will learn how to use the modifiers like *abstract*, *transient*, *volatile*, *native*, *synthonized* etc. while describing different programs through out this course.

Table 4.1 : List of Modifiers

Class Modifier	**Meaning**
abstract	The class cannot be instantiated i.e., we cannot create objects of that class.
final	The class cannot be extended.
public	Its members can be accessed from any other class.
Field Modifier	**Meaning**
private	It is accessible only from within its own class.
protected	It is accessible only from within its own class and its extensions.
public	It is accessible from all classes.
final	It must be initialized and cannot be changed
transient	It is not part of persistent state of an object
volatile	It may be modified by asychronous threads
Method Modifier	**Meaning**
final	It cannot be overridden in class extensions
native	Its body is implemented in another programming language
private	It is accessible only from within its own class.
protected	It is accessible only from within its own class and its extensions.
public	It is accessible from all classes.
static	It has no implicit argument.
synchronized	It must be locked before it can be invoked by a thread.
volatile	It may be modified by asynchronous threads.
Constructor Modifier	**Meaning**
private	It is accessible.
protected	It is accessible only from within its own class and it extensions.
public	It is accessible from all classes.
Local Variable Modifier	**Meaning**
final	It must be initialized and cannot be changed.



CHECK YOUR PROGRESS

Q4. What is the difference between public member and private member of a class?

Q5. What is the difference between class method and an instance method.



4.8 LET US SUM UP

- Java programs are organized by classes, which specify the behaviour of of objects, which control the actions of the program.
- Classes, objects and methods are basic components used in Java programming. A class can contain methods, fields, initialization codes etc. It serves as a blueprint for making class instances, which are runtime objects that implement the class structure.
- A Java program is a collection of one or more text files that define Java classes, atleast one of which is public and contains a *main()* method that has this specific form:

```
public static void main(String[ ] args)
{
      //statements
}
```

- A class definition specifies the variables and methods that are members of the class.
- Each class must be saved in a file with the same name as the class, and with the extension .java.
- A method is a sequence of declaration and executable statements that performs some individual task just like functions in C/C++.
- Method definition have some basic parts like *return type, name of the method, parameter list (*optional), *return statement* (if the method returns a value). If the method is not returning any value then the

return type will be *void*.

- A method is said to be *recursive* when it calls itself.
- In Java, we can pass arguments to a method. Parameters which appear in the method call are *actual* parameters and the parameters which appear in the method definition are termed as *formal* parameters.
- The primitive types (e.g., byte, short, int, long, char, float, double, boolean) are passed by value to a method.
- Reference to an object can also be passed to a method as argument.
- Constructors are used to create new objects. Constructors must have the same name as that of the class when it is declared. It has no return type. Constructors are invoked with the *new* operator The *this* keyword calls another constructor in same class.
- Constructor with no argument is termed as *default constructor*. It is special as because if there is no other constructor in the class, the compiler will automatically define a public default constructor and initializes all fields to their type's default values.
- The *copy constructor* is one whose only parameter is a reference to an object of the same class to which the constructor belongs. It is usually used to duplicate an existing object of the class.
- When there are more than one constructor in a class then it is said to be constructor *overloading*. The name of all constructors should be same with the class name to which they belong but they should have different types and number of arguments.
- We can define one class based on another. This is called class *inheritance*. The base class is called a *superclass* and the derived class is called a *subclass*. A superclass can also be a subclass of another superclass.
- The first statement in the body of a constructor for a subclass should call a constructor for the superclass. If it does not, the compiler will insert a call for the default constructor for the superclass.
- Modifiers are some keywords which are used in the declaration of classes, fields, methods, constructors local variables etc. The use

of different types of modifiers like *public, private, protected, final, abstract, static* etc. are described in this unit.



4.9 FURTHER READINGS

1) Balagurusamy, E. (2007). Programming With Java: A Primer 3E.

2) Schildt, H. (2000). C/C++ Programmer's Reference. McGraw-Hill, Inc..

3) Deitel, P., & Deitel, H. (2011). Java How to program. Prentice Hall Press.



4.10 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : i) fields; ii) instance; iii) java; iv) same; v) new; vi) dot; vii) return type; viii) this(); ix) passed by value; x) Recursion; xi) by value; xii) actual; xiii) one; xiv) copy; xv) initialize

Ans. to Q. No. 2 : static double power(double x, int n)

```
{
    int i;
    double p=1.0;
    for(i=0; i<n; i++)
        p = p*x;
    for(i=0; i<-n; i++)
        p=p/x;
    return p;
}
```

Ans. to Q. No. 3 : A constructor signature consists of the constructor's name, number of parameters and type of parameter. The constructor signature must be unique when constructors are overloaded.

Ans. to Q. No. 4 : A public class member can be accessed from methods of other classes. A private class member can be accessed only from methods of the class.

Ans. to Q. No. 5 : A ***class method*** is declared *static* and is invoked using

the class name. For example,

double p = Math.abs(q);

invokes the class ***abs()*** that is defined in the ***Math*** class. An ***instance method*** is declared without the static modifier and is invoked using the name of the object to which it is bound. For example,

double p = **obj**. nextDouble();

invokes the class method *nextDouble()* that is defined in the *Random* class and is bound to the object *obj* which is an instance of that class.



4.11 MODEL QUESTIONS

Q1. What is a class? How does it accomplish data hiding?

Q2. What are objects? How are they created?

Q3. What is a constructor? How do we invoke a constructor? What are its special properties?

Q4. What is recursive method? Explain with example.

Q5. Discuss different levels of access protection available in Java.

Q6. Design a class to represent a bank account. Include the following data members: *Annount number, Depositor name, Account type, Balance amount in the account* .

Methods : To assign initial value

To deposit an amount

To withdraw an amount after checking balance

To display depositor name and balance.

*****