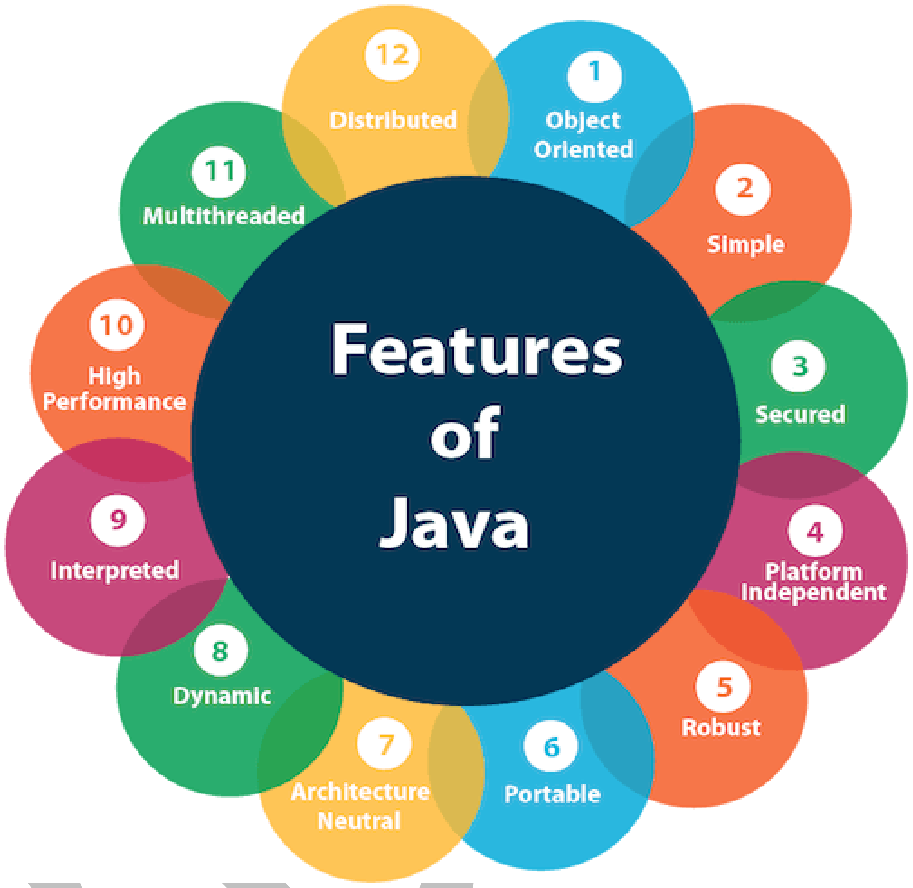
Unit I Long answers

Q) Explain about features of JAVA

The prime reason behind creation of Java was to bring portability and security feature into a computer language. Beside these two major features, there are many other features that are also important:



1) Simple

Java is easy to learn and its syntax is quite simple, clean and easy to understand.
The confusing and ambiguous concepts of C++ are either left out in Java or they have been re-implemented in a cleaner way.
Eg : Pointers and Operator Overloading are not there in java but were an important part of C++.

2) Object Oriented

In java everything is an Object which has some data and behaviour.
Object-oriented means we organize our software as a combination of different types of objects
Java programs are written using classes.

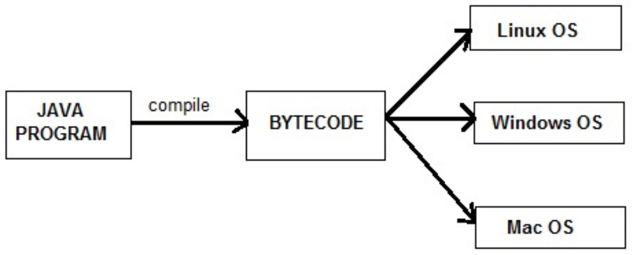
3) Robust

Robust simply means strong. Java is robust because:

- It uses strong memory management.

- No pointers. So, no security problems.
- There is automatic garbage collection
- There are exception handling and the type checking mechanism
-

4) Platform Independent
Unlike other programming languages such as C, C++ etc which are compiled into platform specific machines. Java is guaranteed to be write-once, run-anywhere language.
Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc.
Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once Run



5) Secure
Java is best known for its security. With Java, we can develop virus-free applications. Java is secured because:

- **No explicit pointers**
- **Java Programs run inside a virtual machine sandbox**
- **Bytecode verifier**

6) Multi Threading
Java multithreading feature makes it possible to write program that can do many tasks simultaneously.
This design feature allows the developers to construct interactive applications that can run smoothly.

7) Architectural Neutral
Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.
In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

8) Portable

Compiler generates bytecodes, which have nothing to do with a particular computer architecture, hence a Java program is easy to intrepret on any machine.

9) Both compiled & interpreted
The compilation step allows for code optimization and makes JVM to generate portable code. During interpretation step, Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid since the linking is an incremental and light-weight process.

10) High Performance
Java enables high performance with the use of just-in-time compiler.

11) Distributed
Java allows users to create distributed applications on the Internet such as RMI,EJB.
This feature allows us to access files from any machine on the Internet.

12) Dynamic
Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Q) Explain about Conditional statements in detail.

When we need to execute a set of statements based on a condition then we need to use **control flow statements or conditional statements.**

These are various types conditional statements in java:

1) Simple **if** Statement

2) **if – else** Statement

3) **Nested if-else** statement

4) **else – if Ladder**

5) **switch**statement

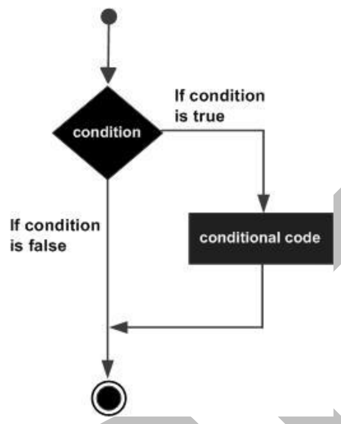
6) Conditional Expression Operator

(1) Simple "if" statement:

The "if" statement is a powerful decision making statement and is used to control the flow of execution of statements.

Syntax:

```
if (Condition or test expression)

{

Statements;

}
```



(2) "if-else" Statement:

```
if (Condition )

 {

      /*true block (or) if block */

      Statements;

 }

else

{

     /* false block (or) else block */

     Statements;

}
```

Example

```
int time = 20;
if (time < 18)
```

```
{
 System.out.println("Good day.");
}

else

{
 System.out.println("Good evening.");
}
```

(3) Nested "if–else" Statement:

- Using of one *if-else* statement in another *if-else* statement is called as *nested if-else* control statement.

Syntax:

```
if (Condition1)

{

    if (Condition2)

        {

          Statement -1;

        }

   else

    {

        Statement -2;

    }

}

else

{

    if (Condition3)
    {
        Statement -3;
    }
    else
    {
         Statement -4;
    }
}
```

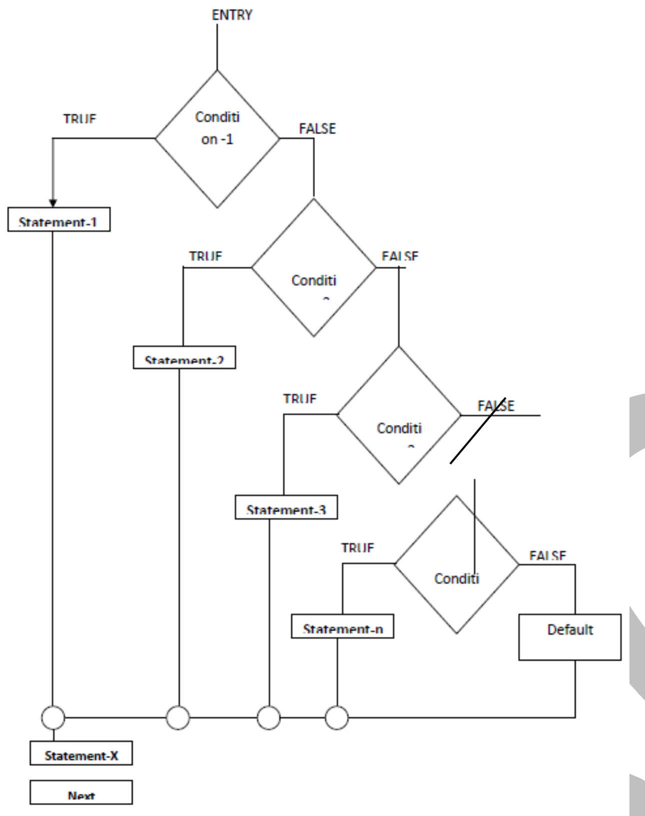
(4) The "else – if" Ladder:

- A multipath decision is a chain of *if* s in which the statement associated with each *else* is an *if*.
- Hence it forms a ladder called *else–if* ladder.

Syntax:

```
if (Condition -1)
    Statement -1;
else if ( Condition -2)
    Statement -2;
else if ( Condition -3)
    Statement -3;
          :
            :
               :
else if (Condition –n)
    Statement –n;
```

```
else
   default statement;
```



(5) The "switch-case" Statement:

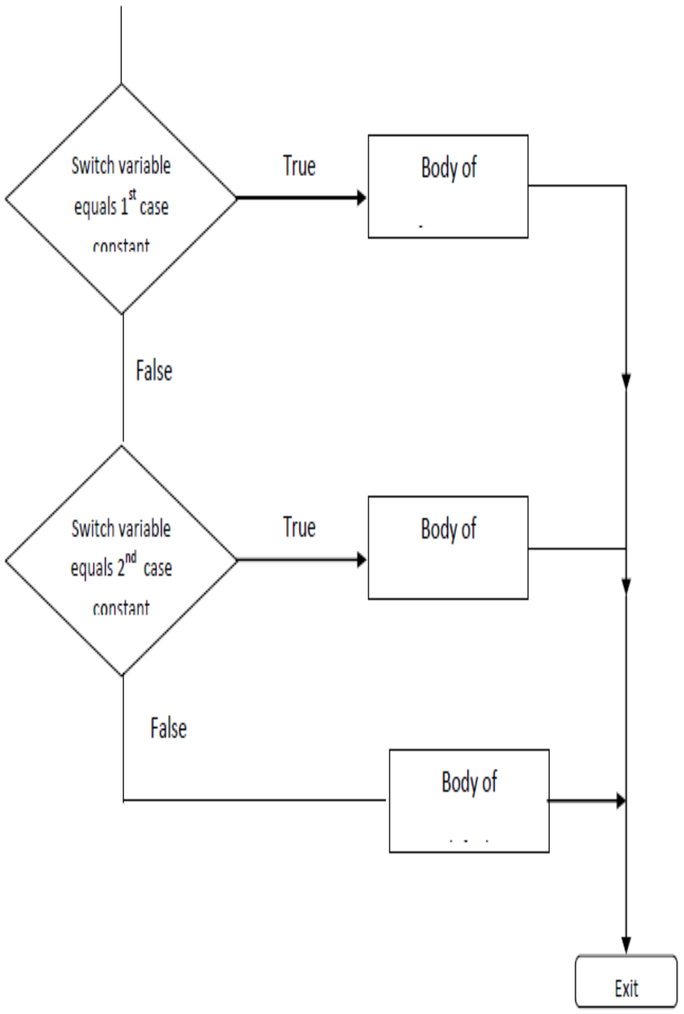
- The *switch* statement is a multi-way branch statement.
- The *switch* statement evaluates expression and then looks for its value among the *case* constants.
- If the value matches with *case* constant, then that particular *case* statement is executed. If no one *case* constant not matched then *default* is executed.
- Here *switch, case* and *default* are keywords.
- Every *case* statement terminates with colon "**:**".

Syntax:

```
switch(expression)
{
  case  Constantvalue-1: Block -1;
```

```
                    break;
 case  Constantvalue-2: Block -2;
                    break;

         _ _ _ _ _ _ _ _ _

         _ _ _ _ _ _ _ _ _


 case  Constantvalue-n: Block -n;
                    break;


 default:  default – block;
 }
```

Flow Chart:


Switch variable equals 1st case constant
True
Body of
False
Switch variable equals 2nd case constant
True
Body of
False
Body of
Exit

(6) Conditional Operator [?:] :

1. It is represented by ? :
2. It takes on 3 Arguments.

Syntax:

Expression1 ? Expression2 : Expression3

where,

Expression1 is Condition

Expression2 is statement followed if Condition is True

Expression3 is statement followed if Condition is False

Qn) Explain about Loop Control Statements in detail.

Loop: A loop is defined as a block of statements which are repeatedly executed for certain number of times.

Types of loops ;

1. for loop
2. while loop
3. do…while loop

1) The "for" loop:

The *for* loop statement comprises of 3 actions.

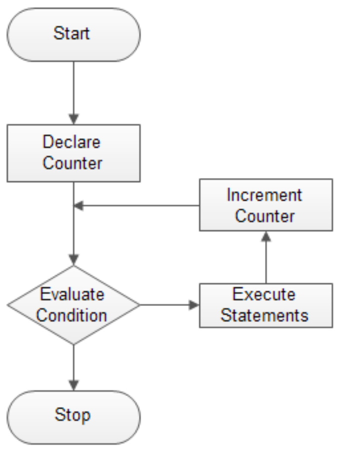
The 3 actions are

"initialize expression", "Test Condition expression" and "updation expression"

- The expressions are separated by Semi-Colons (**;**).
- The loop variable should be assigned with a starting and final value.
- Each time the updated value is checked by the loop itself. Increment / Decrement is the numerical value added or subtracted to the variable in each round of the loop.

Syntax:

```
for(initialize expression; test condition; updation )
 {
   Statements;
 }
```



```
class ForExample

{

public static void main(String[] args)

{

   for(int i=1;i<=5;i++)

   {

      System.out.println(i);

   }

}

    }
```

(2) The " while " loop:

The simplest of all the looping structures.

Syntax:

Initialization Expression;

while(Test Condition)

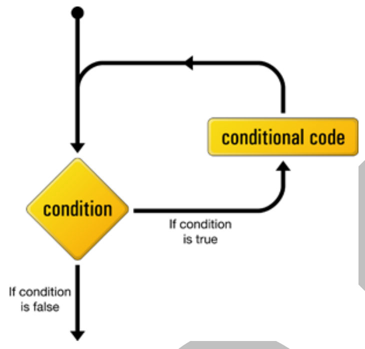
{

Body of the loop

Updation Expression;

}

- The ***while*** is an entry – controlled loop statement.
- The test condition is evaluated and if the condition is true, then the body of the loop is executed.
- The execution process is repeated until the test condition becomes false and the control is transferred out of the loop.
- On exit, the program continues with the statement immediately after the body of the loop.
- The body of the loop may have one or more statements.
- The braces are needed only if the body contains two or more statements.



Example :

```
class WhileExample

{

public static void main(String[] args)

{

   int  i=1;

   while(i<=5)

    {
        System.out.println(i);

        i++;

    }

}

}
```

(3) The " do-while " loop:

- In do-while, the condition is checked at the end of the loop.
- The do-while loop will execute at least one time even if the condition is false initially.
- The do-while loop executes until the condition becomes false.

Syntax:

Initialization Expression;

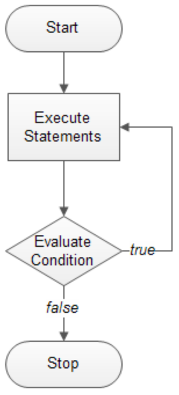
do

{

Body of the loop

Updation Expression;

} while (Test Condition);



class DoWhileExample

```
{
public static void main(String[] args)
{
   int  i=1;
   do
   {
      System.out.println(i);
      i++;
   }while(i<=5);
}
}
```

Q) Explain about branching statements in Java in detail.

Java doesn't support goto statement as in some older languages(C, C++). However, it supports other ways to jump from one statement to another.

There are 2 broad types of branching(jump) statements in Java- break , continue.

Various subtypes :-

- break
- labeled break
- continue
- labeled continue

break Statement :

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

The Java *break* is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

Example

```
for (int i = 1; i <= 5; i++)

{
 if (i == 4)

  {
   break;
  }
 System.out.println(i);
}
```

Output:-

```
1

2

3
```

Labeled break Statement :

A simple break statement can jump out of only the inner loop.

This limitation can be overcome by using a labeled break statement.

We can use break statement with a label. With a labeled break , we can break any loop whether it is outer loop or inner loop.

```
class LBreak

{

  public static void main(String[] args)

  {

        aa:  for(int i=1;i<=3;i++)

               {

                    for(int j=1;j<=3;j++)
                    {
                       if(i==2&&j==2){
                           break aa;
                    }
                    System.out.println(i+" "+j);
                }
          }
     }
}
```

Output:

```
1   1
1   2
1   3
2   1
```

continue Statement :

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

It continues the current flow of the program and skips the remaining code at the specified condition.

In case of an inner loop, it continues the inner loop only.

This example skips the value of 4:

Example

```
for (int i = 1; i <= 5; i++)
```

```
{
 if (i == 4)

 {
   continue;
 }
 System.out.println(i);
}
```

Output:-

```
1

2

3

5
```

<u>Labeled continue Statement :</u>

We can use continute statement with a label.

So, we can continue any loop in Java now whether it is outer loop or inner.

```
class LContinue

{

   public static void main(String[] args)

  {

          aa:  for(int i=1;i<=3;i++)

                {

                     for(int j=1;j<=3;j++)

                     {

                         if(i==2&&j==2){

                              continue aa;
                }
                System.out.println(i+" "+j);
            }
        }
    }
}
```

Output:

```
1  1
1  2
1  3
2  1
3  1
3  2
3  3
```

Q) Explain about Classes & Objects in Java.
OR
Explain about class declaration and object creation in detail.

Classes & Objects are the basic building blocks of OOP languages.

<u>Class :</u>

Definition : - A class is a blueprint or prototype that defines the variables and methods common to all objects of it.

A class is a user-defined datatype.

Class is the fundamental unit of Java programming.

Class declaration : To be simple, a class is declared using the keyword ***class*** followed by a class name.

A detailed **syntax** for declaring a class is :

```
[modifiers]   class ClassName [extends SuperClassName] [ implements InterfaceNames]
{
        // variable declarations
        // method definitions
}
```

Modifiers - declare whether the class is public, private, protected, abstract, or final

ClassName - sets the name of the class we are declaring

SuperClassName - the name of the super class name

InterfaceNames - a comma-separated list of interfaces implemented by this class

Only the keyword ***class*** and ClassName are mandatory. The items within [ ] are optional.

Class Body :

The class contains two sections : variable declarations and method declarations.

Both are defined inside the class.

The **variables** of the class describe the ***state*** of objects of the class and the **methods** describe the ***behavior*** of the objects.

A class can have 3 types of variables : Instance variables

Class variables

Local variables

Instance variables : A class can have many objects and each object has its own set of instance variables.

The instance variables are created when the objects are created.

If a variable of one object is changed, it does not affect the variable of another object.

Q) Explain about Constructors, types of constructors, and constructor overloading.

Definition : Constructors provide a mechanism for automatically initializing the values of an object as soon as the object is created. A **constructor** is a special method of a class that is automatically executed when an object the class is created. It is used to initialize the variables of the objects.

Characteristics of Constructors:-

- They have the same name as the class name they reside in.
- Their syntax is same as method.
- They have no return type, not even void.
- They do not have any modifiers such as private, static, final, abstract, etc.

Example :

```
class Rectangle
{
        int    length,  width;

        Rectangle()
        {
                length = 10;
                width = 6;
        }
}
```

Types of Constructors :

1. Default / no-argument constructor
2. Parameterized constructor

1.Default / no-argument constructor :-

A constructor that does not take any arguments / parameters is called a default constructor.

Eg.

```
        Rectangle()
        {
                length = 10;
                width = 6;
        }
```

We create an object that executes the default constructor as follows :

```
        Rectangle   r  =  new   Rectangle (   );
```

Here, length of object r is 10 and width of object r is 6.

If the constructor is explicitly defined within the class as shown above, it is known as **explicit constructor**.

If there is no explicit constructor in the class, then Java compiler automatically creates a default constructor as soon as the object is instantiated. This is known as **implicit constructor**. This constructor has empty body like--

```
        Rectangle()
        {
        }
```

Implicit constructor initializes the variables to their default values.

<u>Complete program :-</u>

```
class Rectangle
{
        int length,width;
        Rectangle()
        {
                length = 10;
                width = 6;
        }
        int area( )
        {
                return  length * width ;
        }
        public static void main(String args[ ])
        {
                Rectanle  r1 = new  Rectangle( );
                Rectanle  r2 = new  Rectangle( );
                System.out.println("Area=" + r1.area( ));
                System.out.println("Area=" + r2.area( ));
        }
}
```

Output :

```
Area = 60
Area = 60
```

2. Parameterized constructor :-

Just like methods , arguments can be passed to the constructors in order to initialize the instance variables of an object.

A constructor that takes arguments / parameters is called a parameterized constructor. The instance variables of the object are initialized to the values of the parameters when the object is created.

Eg.

```
        Rectangle(int  len,  int  wid )
        {
                length = len ;
                width = wid ;
        }
```

We create an object that executes the parameterized constructor as follows :

```
        Rectangle   r  =  new   Rectangle ( 5, 2  );
```

Here, length of object r is 5 and width of object r is 2.

Complete program :-

```
class Rectangle
{
        int length,width;
        Rectangle(int  len,  int  wid )
        {
                length = len ;
                width = wid ;
        }
        int area( )
        {
                return  length * width ;
        }
        public static void main(String args[ ])
        {
                Rectanle  r1 = new  Rectangle( 5, 2 );
                Rectanle  r2 = new  Rectangle( 15, 3 );
                System.out.println("Area=" + r1.area( ));
                System.out.println("Area=" + r2.area( ));
        }
}
```

Output : -

```
Area = 10
Area = 45
```

Constructor Overloading :

Just like methods, constructors can also be overloaded.

A class can have more than one constructor but with different parameter types and different number of parameters. This phenomenon is known as "**constructor overloading**".

Example :

```
class Rectangle
{
        int length, width;

        Rectangle()
        {
                length = 10;
                width = 6;
        }
        Rectangle(int  len,  int  wid )
        {
                length = len ;
                width = wid ;
        }
        int area( )
        {
            return  length * width ;
        }
        public static void main(String args[ ])
        {
            Rectanle  r1 = new  Rectangle( 5, 2 );
            Rectanle  r2 = new  Rectangle(  );
            System.out.println(“Area=” + r1.area( ));
            System.out.println(“Area=” + r2.area( ));
        }
}
```

Output : -

```
Area = 10

Area = 60
```

Qn) Explain about the types of variables in Java.

Types of Variables

There are three types of variables in java:

- instance variables
- class / static variables
- local variables

Instance Variables :

- They are declared in a class, but outside all the methods
- They are created for every object of the class.
- They are created when an object is created and destroyed when the object is destroyed.
- They represent an object's state.
- Access modifiers (private, public, protected) can be given.
- They have default values.
- They can be accessed using the object name and dot (**.**) operator.

Example :

```
class  Demo

{

  int x=10;

  public static void main(String args[])

  {

    Demo  obj = new Demo( );

    System.out.println(obj.x);

  }

}
```

Output :-

```
10
```

Class / static Variables :

- A static or class variable is created using the keyword “static”.
- It is declared outside all the methods but within the class.
- A static variable is common to all the instances (or objects) of the class.
- It is a class level variable.
- A single copy of static variable is created and shared among all the instances of the class.
- It can be accessed before any objects of its class are created.
- It can be accessed with the classname and dot(**.**) operator without object name.

Example :

```
class StaticDemo

{

  static int x=10;
  public static void main(String args[])
  {
        System.out.println(StaticDemo.x);
  }
}
```

Output :-

```
10
```

Local Variables :

Local variables are declared in methods, constructors, or blocks.

We can use the variable only within that method.

Local variables of one method cannot be accessed in other methods of the class.

Local variables are created when the method, constructor or block is entered and will be destroyed once the method, constructor, or block exits.

```
class LocalDemo
{
  int a=10, b=5;
  void swap()
  {
     int temp;
     temp = a;
     a = b;
```

```
        b = temp;

    }

    public static void main(String args[])

    {

            LocalDemo obj = new LocalDemo();

            System.out.println(obj.a + "   " + obj.b);

            obj.swap();

            System.out.println(obj.a + "    " + obj.b);

    }

}
```

Output :-

```
10     5

5      10
```

Qn) Explain about instance methods and static methods.

OR

Explain the difference between instance methods & static methods.

Instance methods

- They are also known as non-static methods.
- It does not have the "*static" keyword* before the method name.
- We have to create an object to access the instance method.
- They can access static methods and static variables without creating an object.
- They can be overridden.
- They may take parameters and return a value or void.

Syntax :

```
 type  methodName(parameter list)
 {
    // method body
 }
```

Example :

```
class MethodDemo
{
  int x=10;
  void incr( )
  {
      x++;
   }
  public static void main(String args[])
  {
    MethodDemo  obj=new  MethodDemo( );
    obj.incr( );
    System.out.println(obj.x);
  }
}
```

Output :-

```
11
```

Static methods

- They have the "*static" keyword* before the method name.
- It is called using the class (className.methodName).
- They can access only static variables.
- They can't access instance variables.
- They can call other static methods. But they can't call instance methods.
- They can not be overridden.
- They may take parameters and return a value or void.

Syntax :

```
static  type  methodName(parameter list)

{

   // method body

}
```

Example :

```
class StaticDemo

{

  static int x=10;

  static void incr( )

  {

       x++;

   }

  public static void main(String args[])

  {

     StaticDemo.incr( );

    System.out.println(StaticDemo.x);
```

```
      }

    }
```

Output :-

```
11
```

Qn) Explain about "static" keyword.

static Keyword :

In Java, static keyword is used to create the following :

- static variables
- static blocks
- static methods
- static class

static Variables :

- A static or class variable is created using the keyword "static".
- It is declared outside all the methods but within the class.
- A static variable is common to all the instances (or objects) of the class.
- It is a class level variable.
- A single copy of static variable is created and shared among all the instances of the class.
- It can be accessed before any objects of its class are created.
- It can be accessed with the classname and dot(**.**) operator without object name.

Example :

```
class StaticDemo

{

  static int x=10;
  public static void main(String args[])
  {
        System.out.println(StaticDemo.x);
    }
}
```

Output :-

```
10
```

<u>static Methods :</u>

- A static method begins with the keyword “static”.
- It belongs to the class and not to the object.
- A static method can access only static data. It can not access non-static data (instance variables)
- A static method can be accessed directly by the class name and doesn’t need any object
- A static method can call only other static methods and cannot call a non-static method.

Example :

```
class StaticDemo
{
  static int x=10;
  static void incr( )
  {
       x++;
   }
  public static void main(String args[])
  {
     StaticDemo.incr( );
    System.out.println(StaticDemo.x);
  }  }
```

Output :-

```
11
```

static Block :

- It is used to initialize the static data members.
- It is executed before the main method.

Example :

```
public class Demo {
 static int a;
 static int b;

static {
   a = 10;
   b = 20;
 }
 public static void main(String args[])
{
   System.out.println("Value of a = " + a);
   System.out.println("Value of b = " + b);
}
}
```

Output :-

```
Value of a = 10

Value of b = 20
```

static Class :

- A static class is created inside another class.
- It can be accessed by outer class name.

Example :-

```
class Outer{

  static int x=10;

  static class Inner

  {

    static void print()

    {

         System.out.println(x);

    }

  }

  public static void main(String args[])

  {

    Outer.Inner.print();

  }
}
```

Output :-

```
10
```

Qn) Define Array. Explain One – Dimensional & Two-Dimensional arrays with examples.

OR

Define Array. Explain how to create them in Java.

There are situations where we want to store a group of similar type of values in a variable. It can be achieved by the data structure known as array.

Definition : An array is a memory space that can store multiple values of same data type in contiguous (i.e. continuous) locations. The multiple element s can be accessed with a common name.

Eg: a set of ages of group of students

Types of arrays : - There are two types of arrays : One-dimensional

Two-dimensional

One-dimensional Arrays:

In a one-dimensional array, a single index is used. The indexes start from 0 and end with n-1 if the array size is n.

Creation of Arrays: There are 3 steps in creating an Array :

Declaring an array

Creating memory locations

Initializing / assigning values

Declaring an array : There are two ways to declare an array :

```
type   arrayname[  ];

type[ ]   arrayname;
```

Here, type can be a basic data type or classname. arrayname is an identifier i.e., programmer-defined.

```
Eg:   int  age[ ];

      int [  ]   age;
```

Creating memory locations : In this step, we need to allocate memory to store array elements. We allocate memory by specifying the array size(or length) with the new operator.

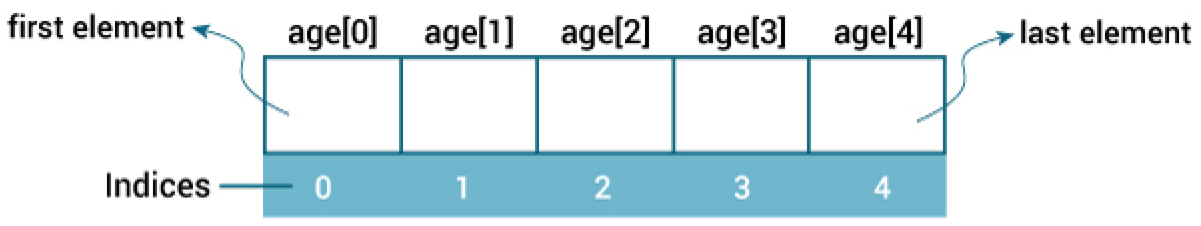
```
arrayname  =  new  type  [  size ] ;
```

The size of the array once defined cannot be changed during the execution.

```
Eg:   age  =  new  int [ 5 ];
```

Both declaration & creation can be combined and written as follows :

```
type[  ]    arrayname  =  new  type  [  size ] ;

type   arrayname[  ]  =  new  type  [  size ] ;
```



Array age **of length 5**

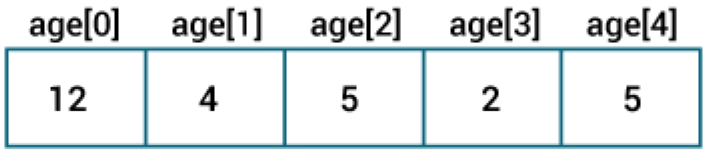
Initializing / assigning values to an array : We can initialize an array during declaration.

```
type[ ]  arrayname = { value1, value2, . . .  .,  valuen };
```

```
int[] age = {12, 4, 5, 2, 5};
```

Above statement creates an array and initializes it during declaration.

The length of the array is determined by the number of values provided which are separated by commas. In our example, the length of `age` array is 5.



We can also assign values to an array as follows :

    arrayname [index] = value ;

eg :

```
int[] age = new int[5];

age[ 0 ] =12;

age[ 1 ] =4;

age[ 2 ] =5;

age[ 3 ] =2;

age[ 4 ] =5;
```

Array Index

Individual array element is accessed using an index as follows :

arrayname [index]

Array Length

To find out how many elements an array has, use the `length` property as follows :

```
arrayname . length
```

Example

```
int[ ] age = (12, 4, 5, 2, 5 };
System.out.println(age.length);
// Outputs 5
```

Loop Through an Array using for loop :

We can loop through the array elements with the `for` loop, and use the `length` property.

The following example outputs all elements in the age array:

Example

```
int[] age = {12, 4, 5, 2, 5};

for (int i= 0;i< age.length;i++)

 {
   System.out.println(age[i]);
}
```

Loop Through an Array with For-Each

There is also a "**for-each**" loop, which is used exclusively to loop through elements in arrays. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

Syntax

```
for (type variable : arrayname)
{

     ...
}
```

Example

```
int[] age = {12, 4, 5, 2, 5};
```

```
for (int item:age)

 {
  System.out.println(item);
}
```

Multidimensional Array in Java

In multidimensional arrays, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

```
type[ ][ ] arrayname;
       (or)
type arrayname[ ][ ];
```

Example to instantiate Multidimensional Array in Java

```
int[ ][ ]  arr=new int[3][3];    //3 row and 3 column
         (or)
int  arr[ ][ ]=new int[3][3];    //3 row and 3 column
```

Example to initialize Multidimensional Array in Java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;

arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;

arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of two-dimensional Java Array

```
class Demo2d
{
    public static void main(String args[])
    {
        //declaring and initializing 2D array
        int arr[][]={   {1,2,3},    {2,4,5},    {4,4,5}   };
       //printing 2D array
       for(int i=0;i<3;i++)
       {
                for(int j=0;j<3;j++)
                {
                     System.out.print(arr[i][j]+" ");
                }
                System.out.println();
        }
    }
}
```

Output:

```
1 2 3
2 4 5
4 4 5
```

*****   ******   ******   *******   *******

Program for Addition of 2 Matrices in Java

```
class AddMatrix
{
        public static void main(String args[])
        {
            //creating two matrices
            int a[][]={{1,3,4},{3,4,5}};
            int b[][]={{1,3,4},{3,4,5}};

            int c[][]=new int[2][3];

            for(int i=0;i<2;i++)
            {
                    for(int j=0;j<3;j++)
                    {
                            c[i][j]=a[i][j]+b[i][j];
                            System.out.print(c[i][j]+" ");
                    }
                    System.out.println();//new line
            }
       }
}
```

Output:

```
2 6 8
6 8 10
```

*******   ******   ******   *******   *********

Program for Multiplication of 2 Matrices in Java

```
class MatrixMul
{
            public static void main(String args[])
            {
                        //creating two matrices
                        int a[][]={{1,1,1},{2,2,2},{3,3,3}};
                        int b[][]={{1,1,1},{2,2,2},{3,3,3}};

                        int c[][]=new int[3][3];

                        for(int i=0;i<3;i++)
                        {
                                    for(int j=0;j<3;j++)
                                    {
                                                c[i][j]=0;
                                                for(int k=0;k<3;k++)
                                                {
                                                            c[i][j]+=a[i][k]*b[k][j];
                                                }
                                                System.out.print(c[i][j]+" ");
                                    }
                                    System.out.println();
                        }
            }
    }
```

Output:

```
6     6    6
12   12   12
18   18   18
```

Passing Array to a Method in Java

We can pass the array to method so that we can reuse the same logic on any array.

Let's see the simple example to print an array using a method.

```
class  Printarray
{

   static void print(int arr[])
   {
           for(int i=0;i<arr.length;i++)
                   System.out.println(arr[i]);
    }

   public static void main(String args[])
  {
           int a[]={33,3,4,5};
           print(a);
  }
}
```

Output:

```
33
3
4
5
```

*******   ******   ******   *******   *********

Returning Array from the Method

We can also return an array from the method in Java.

```
class ReturnArray
{
           static int[] get()
           {
                   int  x[ ] = new int[]{10,30,50,90,60};
                   return x;
            }
```

```
     public static void main(String args[])
    {
            int arr[]=get();
            for(int i=0;i<arr.length;i++)
                    System.out.println(arr[i]);
    }
}
```

Output:

```
10
30
50
90
60
```

Q) Explain about Nested Classes and types in detail.

A class defined within another class is known as Nested class.

Syntax:

```
class Outer
{
    //class Outer members

    class Inner
    {
        //class Inner members
    }
}
```

Advantages of Nested Class:

1. It is a way of logically grouping classes that are only used in one place.
2. It increases encapsulation.
3. It can lead to more readable and maintainable code.

Types of Nested Classes :

- Non-static nested classes / Inner classes
- Static nested classes
- Local inner classes
- Anonymous classes

Non-static nested classes / Inner classes:-

- It is created outside the methods within outer class.
- It is the member of the outer class.
- It can be created only within the scope of Outer class.
- It can access all variables and methods of Outer class including its private data members and methods and may refer to them directly.
- But Outer class cannot directly access members of Inner class.

Example :-

```
class Outer
{
 public void display()
 {
  Inner in=new Inner();
  in.show();
 }
 class Inner
 {
  public void show()
  {
   System.out.println("Inside inner");
  }
 }
 public static void main(String[] args)
 {
  Outer ot = new Outer();
  ot.display();
 }
 }
```

Output :-

Inside inner

Static nested classes :-

- If the nested class has the keyword "static" before the keyword "class", then it is called as **static nested class**.
- It can access static variables of outer class.
- It cannot access non-static (instance) variables and instance methods.
- It can be accessed by outer class name.

Example :-

```
class  Outer
{
   static int x=30;

   static class Inner
   {
        void show()
        {
                  System.out.println("x = "+x);
        }
   }
   public static void main(String args[])
   {
        Outer.Inner obj=new Outer.Inner();
        obj.show();
   }
}
```

Output:

```
x = 30
```

Local Inner Classes :-

- A class created inside a method is called local inner class.
- To invoke the methods of local inner class, we must instantiate it inside the method.

Example :

class Outer

```
{
 int x=30;
 void display()
 {
  class Local
  {
   void show()
   {
       System.out.println(x);}
   }

   Local lobj=new Local();
   lobj.show();
 }
 public static void main(String args[])
  {
    Outer obj=new Outer();
    obj.display();
 }
}
```

Output:

```
x = 30
```

<u>Anonymous Inner Classes :</u>

- It is a class that has no name.
- It should be used if we have to override method of class or interface.
- Java Anonymous inner class can be created by two ways:

    1. Class (may be abstract or concrete).
    2. Interface

Q) Define Inheritance. Explain the types of inheritance with examples.

Definition :- **Inheritance** is a mechanism in which one object acquires all the properties and behavior of another object.

- It is an important concept of OOPs.
- The idea behind inheritance is that we can create new classes from the existing classes.
- When we inherit, we can reuse methods and variables of the parent class.
- Moreover, we can add new methods and variables in the child class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Use of inheritance

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability i.e the same methods and variables defined in a super class can be used in the subclass.

Terms used in Inheritance

- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Syntax

```
class   Subclassname   extends   Superclassname
{
        //methods and variables
}
```

The **extends keyword** indicates that we are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Example :

```
class Vehicle
{
   ......
}
class Car extends Vehicle
```

```
{
   .......    //extends the property of vehicle class
}
```

Here,

- **Vehicle** is super class of **Car**.
- **Car** is sub class of **Vehicle**.
- Car IS-A Vehicle.

Types of Inheritance

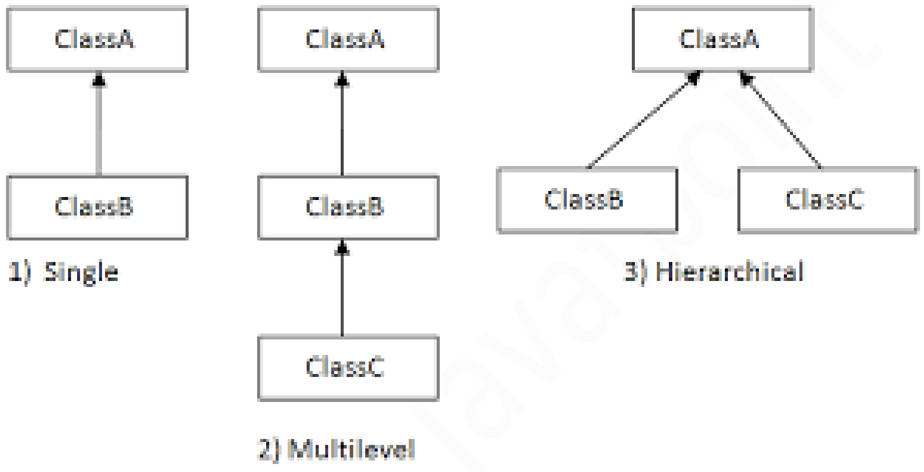
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

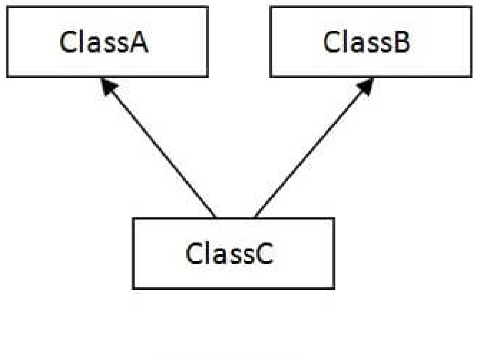
1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance

NOTE: Multiple inheritance is not supported in java

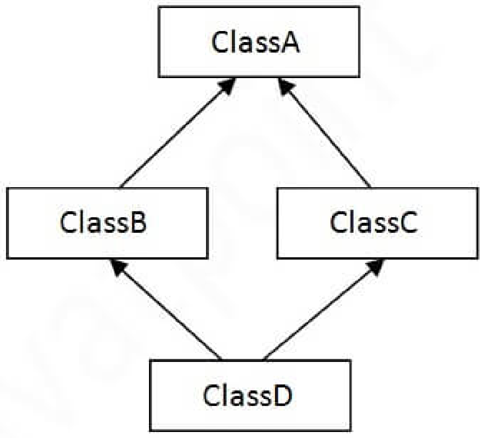
In java programming, multiple and hybrid inheritance is supported through interface only.

4. Multiple Inheritance
5. Hybrid Inheritance





4) Multiple



5) Hybrid

1. Single Inheritance

A single class is derived from another single class. So, there is only one superclass and only one subclass.

```
class A {

   public void methodA()

   {

     System.out.println("class A method");

   }

}

class B extends A {

     public void methodB()

     {

       System.out.println("class B method");

     }
}

class SI {

   public static void main(String args[]) {

      B bo = new B();

      bo.methodA();

      bo.methodB();

  }

}
```

Output :-

```
class A method

class B method
```

2. Multilevel Inheritance

A derived class id created from another derived class and can have any number of levels.

```
class A {

   public void methodA()

   {

      System.out.println("class A method");

   }

}

class B extends A {

     public void methodB()
     {

       System.out.println("class B method");

     }

}

class C extends B  {

     public void methodC()

     {

       System.out.println("class C method");

     }

}

class ML {

   public static void main(String args[]) {

      C co = new C();

      co.methodA(); //calling grand parent class method

      co.methodB(); //calling parent class method

      co.methodC(); //calling local method

  }

}
```

<u>Output :-</u>

```
class A method

class B method
class C method
```

3. Hierarchical Inheritance

There are more than one derived classes which get created from one single base class.

```
class A {
   public void methodA()
   {
      System.out.println("class A method");
   }
}
class Y extends A  {
     public void methodY()
     {
       System.out.println("class Y method");
     }
}
class Z extends A {
   public void methodZ()
   {
     System.out.println("class Z method");
   }
}
```

```
class HI {

   public static void main(String args[]) {

      Y yo = new Y();

      yo.methodA();

      yo.methodY();

      Z zo = new Z();

      zo.methodA();

      zo.methodZ();

  }

}
```

Output :-

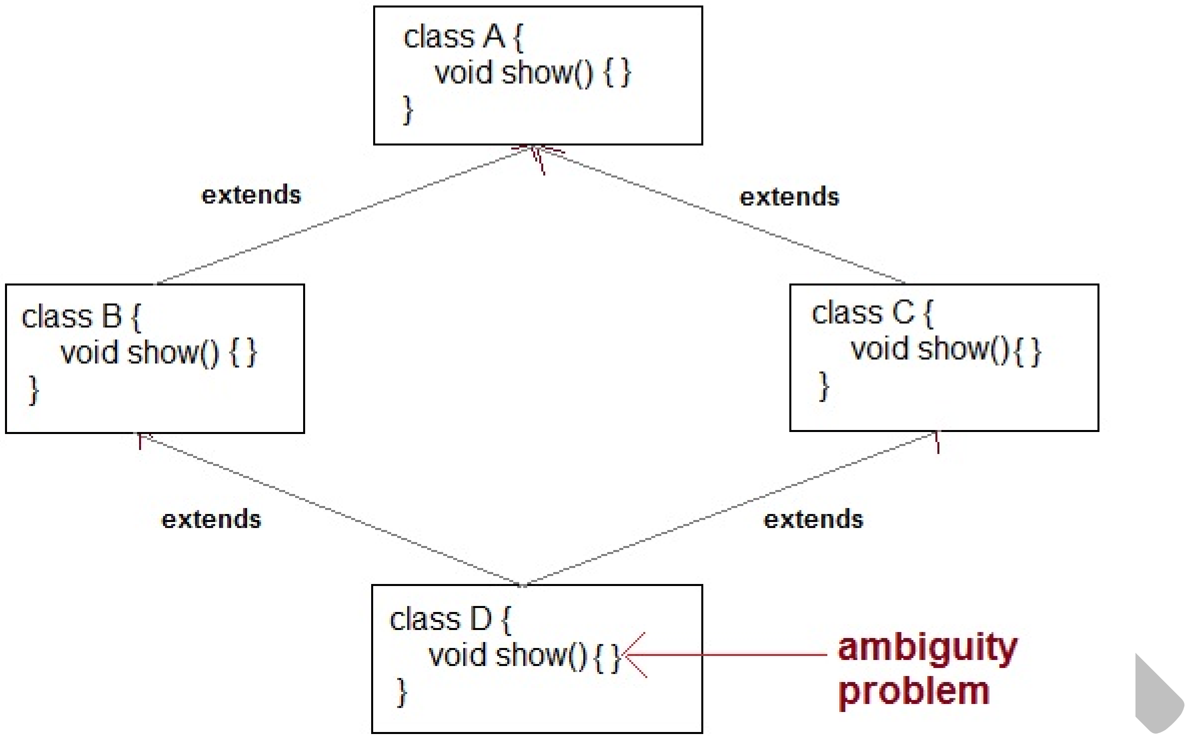
```
class A method

class Y method

class A method

class Z method
```

4. Multiple Inheritance

One derived class is created from more than one base class.

Why multiple inheritance is not supported in Java?

- To remove ambiguity.
- To provide more maintainable and clear design.



Java solves this ambiguity situation by the use of interfaces .

In Java, multiple inheritance is achieved by using interfaces.

```
interface I1 {

     public void m1();

}

interface I2 {

     public void m2();

}

class C implements I1, I2 {

     public void m1()

     {
        System.out.println("I1 m1 called");

     }

    public void m2()

     {

          System.out.println("I2 m2 called");

     }

     public void m3()

     {

          System.out.println("C m3 called");

     }

}

class MultipleInh {

     public static void main(String args[]) {

          C co = new C();

          co.m1();

          co.m2();

          co.m3();

     }

}
```

Output :-

```
I1 m1 called
```

```
I2 m2 called

C m3 called
```

5. Hybrid Inheritance

It is a combination of **Single** and **Multiple inheritance.**

It can also be achieved in a same way as multiple inheritance by using interfaces.

```
interface  I1 {

   public void m1();

}

interface I2 extends I1 {

     public void m2();

}

interface I3 extends I1 {

     public void m3();

}

class C implements I2, I3 {

     public void m1()

     {

        System.out.println("m1 called");

     }

     public void m2()

     {
            System.out.println("m2 called");
        }
        public void m3()
        {
            System.out.println("m3 called");
        }
        public void m4()
        {
            System.out.println("m4 called");
        }
}
class HybInh {
        public static void main(String args[]) {
            C co = new C();
            co.m1();
            co.m2();
            co.m3();
            co.m4();
        }
}
```

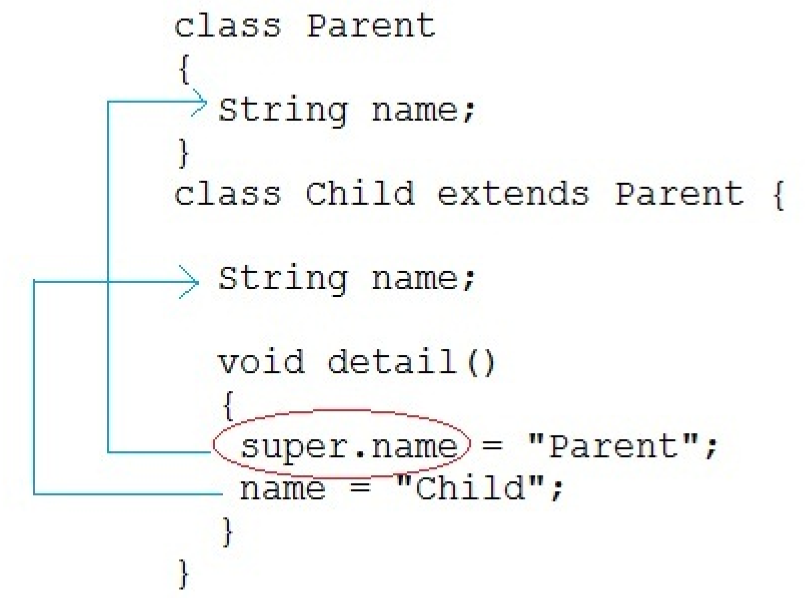
<u>Output :-</u>

```
m1 called

m2 called

m3 called

m4 called
```

Q) Explain about the "super" Keyword.

In Java, super keyword is used to refer to immediate parent class of a child class.

In other words **super** keyword is used by a subclass whenever it needs to refer to its immediate super class.



```
class Parent
{
  String name;
}
class Child extends Parent {

  String name;

  void detail()
  {
   super.name = "Parent";
   name = "Child";
  }
}
```

1. Example of Child class referring Parent class property using super keyword

In this example, we will focus on accessing the parent class property or variables.

```
class Parent
{
   String name;


}
public class Child extends Parent {
   String name;
   public void details()
   {
       super.name = "Parent";  //refers to parent class member
       name = "Child";
       System.out.println(super.name+" and "+name);
   }
   public static void main(String[] args)
   {
       Child cobj = new Child();
       cobj.details();
   }
}
```

Output:

```
Parent and Child
```

2. ***Example of Child class referring Parent class methods using super keyword***

In this example, we will focus on accessing the parent class methods.

```
class Parent
{
    String name;
    public void details()
    {
        name = "Parent";
        System.out.println(name);
    }
}
public class Child extends Parent {
    String name;
    public void details()
    {
        super.details();        //calling Parent class details() method
        name = "Child";
        System.out.println(name);
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.details();
    }
}
```

Output:

```
Parent
Child
```

3. Example of Child class calling Parent class constructor using super keyword

In this example, we will focus on accessing the parent class constructor.

```
class Parent
{
    String name;

    public Parent(String n)
    {
        name = n;
    }


}
public class Child extends Parent {
    String name;

```

```
   public Child(String n1, String n2)
   {

      super(n1);       //passing argument to parent class constructor
      this.name = n2;
   }
   public void details()
   {
      System.out.println(super.name+" and "+name);
   }
    public static void main(String[] args)
   {
      Child cobj = new Child("Parent","Child");
      cobj.details();
   }
}
```

Output:

```
Parent and Child
```

Note: When calling the parent class constructor from the child class using super keyword, super keyword should always be the first line in the method/constructor of the child class.

Q) Explain about final Keyword.

The **final keyword** is used to restrict the user.

The final keyword can be applied with the following:

1. variable
2. method
3. class

1. <u>final Variable:-</u>

If we make any variable as final, we cannot change the value of final variable(It will be constant).

Example :

```
class Demo

{

    final int x=10;   //final variable

    void setx()

   {

         x=4;     //  error : we  cannot change final variable

   }

   public static void main(String args[])

   {

            Demo obj=new  Demo();

            obj.setx();

   }

}
```

Output:-

```
    Compile Time Error
```

2. **final Method:-**
   If we make any method as final, we cannot override it.

Example :

```
    class A
```

```
{
  final void show()
  {
    System.out.println("class A show");
  }
}
class B extends A
{
        void show()       //final method cannot be overridden
        {
         System.out.println("class B show");
    }
    public static void main(String args[])
    {
         B bo = new B();
         bo.show();
    }
}
```

Output:-

```
Compile Time Error
```

3. <u>final Class:-</u>

If we make any class as final, we cannot extend it.

Example :

```
final  class A

{

}

class B extends A   //  final class cannot be extended

{

        void show()

        {

          System.out.println("class B show");

        }

    public static void main(String args[])

    {

         B bo = new B();

         bo.show();

    }

}
```

Output:-

```
Compile Time Error
```

Q) Explain about abstract class and abstract method.

A class which is declared as abstract is known as an **abstract class**.

Abstract class contains one or more abstract methods.

It needs to be extended and its abstract methods need to be implemented.

Example of abstract class

```
abstract class Animal
{
        ……
}
```

Some points about abstract class:

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods .

Abstract Method :

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
abstract void printStatus();    //no method body and abstract
```

Example :

```
abstract class Animal
{
  abstract void sound();
}

class Dog extends Animal
{

  void sound()
  {
          System.out.println("Woof");
  }
   public static void main(String args[])
  {
            Animal obj = new Dog();
            obj.sound();
  }
}
```

Output:

```
Woof
```

Q) Explain about Interfaces in detail.

The interface in Java is *a mechanism to achieve abstraction*.

An interface contains only static constants and abstract methods.

It is a completely "**abstract class**" that is used to group related methods with empty bodies.

An interface must be "implemented" by another class with the implements keyword.

Points about Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, we must define all its methods
- Interface methods are by default abstract and public
- Interface variables are by default public, static and final
- An interface cannot contain a constructor

Use of interfaces:

There are mainly three reasons to use interface.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Syntax for declaring an interface:

```
interface  InterfaceName

{
    // declare constant variables

    // declare methods that abstract by default.

}
```

Example :

```
interface Shape

{

    void draw();

}
```

Syntax for implementing an interface:

```
class  ClassName  implements  InterfaceName

{

      // interface method definition

}
```

Sample Program:

```
interface Bank

{

   float rateOfInterest();

}

class SBI implements Bank

{

     public float rateOfInterest()

     {
            return 9.15f;
    }
}
class AB implements Bank
{
    public float rateOfInterest()
    {
            return 9.7f;
    }
}

class  Demo
{
   public static void main(String[] args)
   {
            Bank b=new SBI();
            System.out.println("ROI: "+b.rateOfInterest());
   }
}
```

Output:

```
ROI: 9.15
```

Q) What is the difference between Interfaces & Abstract classes ?

Interface	Abstract Class

Multiple inheritance possible; a class can inherit any number of interfaces.	Multiple inheritance not possible; a class can inherit only one class.
A class uses **implements** keyword to inherit an interface.	**extends** keyword is used to inherit a class
By default, all methods are public & abstract.	Methods need explicit use of public, static.
By default, all variables are public, static, final	Variables need to be declared as public, static, final
Have no implementation at all	Can have partial implementation
All methods need to be overridden	Only abstract methods need to be overridden
Do not have constructors	Can have constructors
Methods cannot be static	Concrete methods can be static